Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 253 (2009) 3–14

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Virtual-Machine Abstraction and Optimization Techniques

Stefan Brunthaler[1]

Institute of Computer Languages
Vienna University of Technology
Vienna, Austria

Abstract

Many users and companies alike feel uncomfortable with execution performance of interpreters, often also dismissing their use for specific projects. Specifically virtual machines whose abstraction level is higher than that of the native machine they run on, have performance issues. Several common existing optimization techniques fail to deliver their full potential on such machines. This paper presents an explanation for this situation and provides hints on possible alternative optimization techniques, which could very well provide substantially higher speedups.

Keywords: Interpreter, Virtual-Machine Abstraction, Optimization Techniques.

1 Motivation

1000 : 10 : 1. These are the slowdown-ratios of an inefficient interpreter, when compared to an efficient interpreter, and finally to an optimizing native code compiler. Many interpreters were not conceived with any specific performance goals in mind, but rather striving for other goals of interpreters, among them portability, and ease of implementation. This also means that there is a huge benefit in optimizing an interpreter before taking the necessary steps to convert the tool chain to a compiler. There are common optimization techniques for interpreters, e.g. threaded code [2],[5],[8], superinstructions [9], and switching to a register based architecture [20]. The mentioned body of work provides careful analyses and in-depth treatment of performance characteristics, implementation details.

Those optimization techniques, however, have one thing in common: their *basic assumption* is that interpretation's most costly operation is instruction dispatch, i.e., in getting from one bytecode instruction to its successor. While this assumption

[1] Email: `brunthaler@complang.tuwien.ac.at`

1571-0661 © 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.11.011

is certainly true for the interpreters of languages analyzed in the corresponding papers, e.g. Forth, Java, and OCaml, our recent results indicate that it is specifically not true for the interpreter of the Python programming language. We find that this correlates with a difference in the virtual machine abstraction levels between their corresponding interpreters.

In virtual machines where the abstraction level is very low, i.e., essentially a 1 : 1 correspondence between bytecode and native machine code, the basic assumption of dispatch being the most costly operation within an interpreter is valid. Members of this class are the interpreters of Forth, Java, and OCaml, among others. Contrary to those, interpreters that provide a high abstraction level do not support this assumption. The interpreters of Python, Perl, and Ruby belong here. Additionally, we analyze the interpreter of Lua, which is somewhere in between both classes.

In their conclusion, Piumarta and Riccardi [15] suppose the following: "*The expected benefits of our technique are related to the average semantic content of a bytecode. We would expect languages such as Tcl and Perl, which have relatively high-level opcodes, to benefit less from macroization. Interpreters with a more RISC-like opcode set will benefit more — since the cost of dispatch is more significant when compared to the cost of executing the body of each bytecode.*" Our work shows, whether their expectations turn out to be correct, and we make explicit what is only implicitly indicated by their remark. Specifically we contribute:

- We categorize some virtual machines according to their abstraction level. We show which characteristics we consider for classifying interpreters, and provide hints regarding other programming languages than those discussed.
- We subject optimization techniques to that categorization and analyze their potential benefits with respect to their class. This serves as a guideline for a) implementers, which can select a set of suitable optimization techniques for their interpreter, and b) researchers which can categorize other optimization techniques according to our classification.

2 Categorization of Interpreters

To get a big picture on the execution profile of the Python interpreter, we collected 9 million samples of instruction execution times running the `pystone` benchmark on a modified version of the Python 3.0rc1 interpreter, which samples CPU cycles. We sampled Operation Execution, Dispatch and Whole Loop costs. Operation Execution contains all cycle costs for the first machine instruction in operation implementation until the last. Dispatch costs contain the number of cycles spent for getting from one operation to another, e.g. in a switch-statement from one case to another. Python's interpreter, however, does not dispatch directly from one bytecode to another, but maintains some common code section which is conditionally executed before dispatching to the next instruction. To account for that special case, we measured so called Whole Loop costs, which measure the CPU cycles from the first and last instructions within the main loop.

Based on extensive previous work, [2],[5],[8],[9],[20], we expected that instruc-

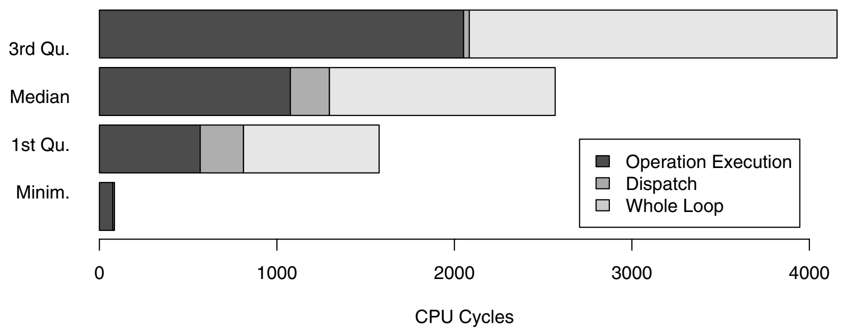


Fig. 1. CPU cycles per code section for Minimum, 1st quartile, median, and 3rd quartile measures.

tion dispatch would be the most costly interpreter activity for Python's virtual machine, too. Figure 1 shows our results obtained by examining CPU cycles for the Python 3.0rc1 interpreter, running on a Pentium 4, 3 GHz, with Xubuntu 8.04.

Section	Min	1st Quartile	Median	3rd Quartile
Op-Execution	75	568	1076	2052
Dispatch	84	812	1296	2084
Whole Loop	84	1576	2568	4156

Table 1
Minimum, 1st quartile, median, and 3rd quartile values for CPU cycles per code section. All values here are *inclusive*, i.e., values for *Dispatch* include *Operation Execution*, and *Whole Loop* includes values of *Dispatch*, and *Operation Execution* respectively.

Our results do not support the assumption of instruction dispatch being the most costly operation for the Python interpreter, actually operation execution is. For a detailed explanation of why this is, we present a comparative example of one instruction implementation of the interpreters of Java, OCaml, Python, and Lua in Section 2.1:

Java, according to the latest Java Language Specification [10], the Java instruction set consists of 205 operations, including reserved opcodes. Instructions are typed for the following primitive types: integers, longs, floats, doubles, and addresses. In our example we take a look at the Sable VM, version 1.13.

OCaml, is a derivative of ML enriched with object oriented elements [13]. Version 3.11.0 contains 146 instructions. Among those are regular stack manipulation instructions, complemented by instructions to manipulate the environment, which is needed for function application, and evaluation respectively. Additionally, it contains direct support for integer operations, which are however not documented in the corresponding documentation [3].

Python, is a multi-paradigm dynamically typed programming language, that enables hybrid object-oriented/imperative programming with functional programming elements [18]. It has 93 instructions in Python 3.0rc1. Most of its operations support ad-hoc polymorphism, e.g. `BINARY_ADD` concatenates string operands, but does numerical addition on numerical ones [17].

Lua, is somewhat similar to Python according to its characteristics, a multi-paradigm programming language that allows functional, imperative and object-oriented (based on prototypes) programming techniques. It includes a lightweight—it has just 38 instructions—and fast execution environment based on a register architecture [16].

It is worth noting that our results are not restricted to those programming languages only. Actually, we conjecture that this is true for the interpreters of programming languages with similar characteristics, i.e., for the Python case this also includes Perl [21], and Ruby [14].

2.1 Categorization based on the comparative addition example

Our classification scheme requires the assessment of the abstraction level of several virtual machines interpreting different languages. In order to do so, we take a representative bytecode instruction present in all our candidates and analyze their implementations. This enables us to show important differences in bytecode implementation, and in consequence allows us to classify them accordingly.

The representative instruction we use for demonstration is integer addition, e.g. for Java we take a look at `IADD`, for OCaml we show `ADDINT`, for Python `BINARY_ADD`, and for Lua we inspect `OP_ADD`. With the notable exception of Lua, all our candidates use a stack architecture, i.e., they need to pop their operands off the corresponding stack, and push their result onto it before continuing execution. In Lua's register architecture, operand-registers and result-registers are encoded in the instruction.

We have highlighted the relevant implementation points by using a bold font, and use arrows for additional clarity.

```
case SVM_INSTRUCTION_IADD:
    {
         jint value1 = stack[stack_size - 2].jint;
         jint value2 = stack[--stack_size].jint;
         stack[stack_size - 1].jint= value1 + value2;
         break;
    }
```

Fig. 2. Implementation of Java's integer addition operation, `IADD` in Sable VM v1.13.

Figures 2 and 3 share an interesting characteristic. They do not implement addition on the virtual machine level, but express the bytecode addition by leveraging the addition used by the compiler, i.e., expose the addition of the native

```
Instruct(ADDINT):
    accu = (value)((intnat) accu + (intnat) *sp++ - 1);
    Next;
```

Fig. 3. Implementation of integer addition in OCaml v3.11.0.

machine. Consequently, the virtual machine addition is expressed using a single native machine instruction. This constitutes our class of a *low abstraction level virtual machines*, where the interpreter is only a thin layer above a real machine.

```
BINARY_ADD:
    w = POP();
    v = TOP();
    if (PyUnicode_CheckExact(v) &&
        PyUnicode_CheckExact(w)) {
        x= unicode_concatenate(v, w, f, next_instr);
        goto skip_decref_vx;
    }
    else {
        x= PyNumber_Add(v, w);
    }
    Py_DECREF(v);
  skip_decref_vx:
    Py_DECREF(w);
    SET_TOP(x);
    if (x == NULL) continue;
    break;
```

Fig. 4. Implementation of integer addition in Python 3.0rc1.

Python's case (cf. Figure 4) shows a contrary picture: the upper arrow shows that `BINARY_ADD` does unicode string concatenation on string operands by calling `unicode_concatenate`. On non-string operands it calls `PyNumber_Add`, which implements dynamic typing and chooses the matching operation based on operand types, indicated by the lower arrow. In our integer example, the control flow would be: `PyNumber_Add`, `binary_op`, and finally `long_add`. If, however the operands were float, or complex types, then `binary_op` would have diverted to `float_add`, or `complex_add` respectively (cf. Figure 5).

Aside from this ad-hoc polymorphism, the addition in Python 3.0 has an additional feature: it allows for unbounded range mathematics for integers, i.e., it is not restricted by native machine boundaries in any way. As a direct consequence, the original Python add instruction cannot be directly mapped onto one native machine instruction in the interpreter. This constitutes our second class, namely *high abstraction level virtual machines.*

Our classes are by no means completely separated and disjoint, since interpreters

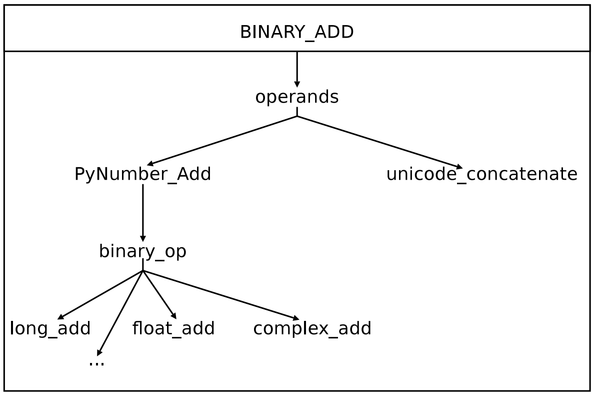
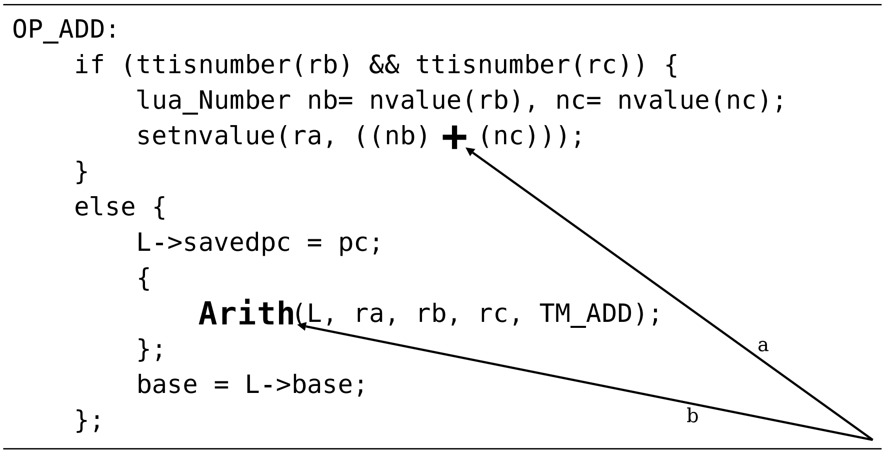


Fig. 5. Ad-hoc polymorphism in Python's `BINARY_ADD` instruction.



```
OP_ADD:
    if (ttisnumber(rb) && ttisnumber(rc)) {
        lua_Number nb= nvalue(rb), nc= nvalue(nc);
        setnvalue(ra, ((nb) + (nc)));
    }
    else {
        L->savedpc = pc;
        {
            Arith(L, ra, rb, rc, TM_ADD);
        };
        base = L->base;
    };
```

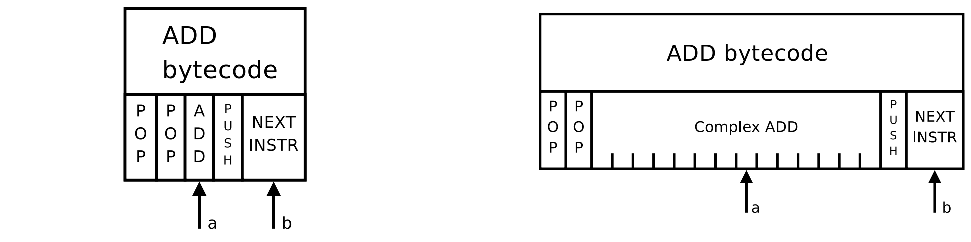
Fig. 6. Implementation of integer addition in Lua 5.1.4.

can be members of both classes, having some subset of instructions belong into one set, and a separate subset of instructions to the other. This is exemplified in Lua (cf. Figure 6) in which addition has characteristics of both classes.

In Lua, if operand types are numeric it delegates the actual addition implementation to the compiler, and therefore to the machine (cf. Figure 6 arrow a). No distinction between float, double, long, and integers is necessary, because Lua uses double as its default numeric type. So far, this would indicate a low abstraction level. However, if operand types are non-numeric, Lua's implementation delegates to the `Arith` (cf. Figure 6, arrow b) function, which tries to convert these operand types into a numeric representation that can be added, e.g. if given a string operand which holds a non-ambiguous numerical value, it would extract this value and continue with regular addition. This "operand-polymorphism" is often found in other programming languages, too—e.g. in Perl—and constitutes a high abstraction level instruction.

2.2 Comparison of Low and High Abstraction Level Virtual Machines

The previous section introduces our two classes of interpreters, namely:



(a) Low Abstraction Level Virtual Machine (b) High Abstraction Level Virtual Machine

Fig. 7. Illustration of Virtual Machine Abstraction Levels. Important are the different ratios of $a : b$ which affects the relative optimization potential of various optimization techniques.

- Low abstraction level, where operation implementation can be directly translated to a few native machine instructions. Figure 7(a) shows the implementation of the interpreter's add instruction, and how the actual add is realized using a single machine add instruction.
- High abstraction level, where operation implementation requires *significantly* more native machine instructions than for low abstraction level. Analogous to Figure 7(a), Figure 7(b) shows the relative impact of implementing a complex add. Frequent characteristics for high abstraction level are:
  - *Ad-Hoc Polymorphism*: a) either polymorphic operations are selected for concrete tuples of operand types, e.g. in Python, or b) operand-type coercion into compatible types for a given operation implementation, e.g. in Lua or Perl.
  - *Complex Operation Implementation*: In Python's case, this complexity directly maps to unbounded integer range mathematics for numeric operations, or full unicode support at the interpreter level.

3 Optimizations for Low Abstraction Level Interpreters

The well known techniques for interpreter optimization focus on reduction of instruction dispatch cost. As shown in Figure 7(a), virtual machines with low abstraction level are particularly well suited for those techniques, since dispatch often is their most expensive operation. Threaded code [5] reduces the instructions necessary for branching to the next bytecode implementation. The regular switch dispatch technique requires 9-10 instructions, whereas e.g. direct threaded code needs only 3-4 instructions for dispatch [6], with only one indirect branch. Superinstructions [9] substitute frequent blocks of bytecodes into a separate bytecode, i.e., they eliminate the instruction-dispatch costs between the first and last element of the replaced block.

Recent advances in register based virtual machines [20], however, suggest a complete architectural switch from a stack-based interpreter architecture to a register based model. This model decreases instruction dispatches by eliminating a large number of stack manipulation operations, i.e., the frequent LOAD/STORE operations that surround the actual operation. The paper reports that 47% of Java bytecode instructions could be removed, at the expense of growing code size of about 25%.

Table 2 shows a list of achievable speedups for low abstraction level interpreters.

Optimization Technique	Speedup Factor	Reference
Threaded Code (compared to switch dispatch interpreter)	up to 2.02	[8]
Superinstructions (compared to threaded code interpreter)	up to 2.45	[7]
Replication + Superinstructions (compared to threaded code interpreter)	up to 3.17	[7]
Register vs. Stack Architecture (both using switch dispatch)	1.323 avg	[20]
Register vs. Stack Architecture (both using threaded code)	1.265 avg	[20]

Table 2
Reference of reported speedup factors for several techniques.

For high abstraction level virtual machines, these speedups are not nearly as high. Vitale and Abdelrahman [22] actually report that applying their optimization technique to Tcl has *negative* performance impacts on some of their benchmarks, because of instruction cache misses due to complex operation implementation leading to excessive code growth—the main characteristic we use to identify high abstraction level virtual machines.

This, however, does not mean that these techniques are irrelevant for virtual machines with a high abstraction level. Actually, quite the opposite is true: once techniques for optimizing the high abstraction level are implemented, the ratio of operation-execution vs. instruction-dispatch (as indicated by the arrows a and b in Figures 7(a) and 7(b)) has favorably changed their optimization potential. Therefore, our categorization merely provides an ordering of relative merits of various optimization techniques, such that considerable deviations in expected/documented vs. actually measured speedups are not stunningly surprising anymore.

4 Optimizations for High Abstraction Level Interpreters

Figure 7(b) shows that many machine instructions are necessary for realizing the high abstraction level of an interpreter instruction. Therefore we are interested in cutting down the costs here, since they provide the greatest speedup potential. In this situation it makes sense to provide a reminder as to what characteristics constitute our classification into the class of high abstraction level. As already mentioned earlier in the addition example, there are two answers to that question:

a) ad-hoc polymorphism

b) complex operation implementation (unbounded range mathematics)

Hence there are two issues to deal with. In the first case (a), a look at the history of programming languages provides valuable insights. We are trying to find programming languages with similar characteristics like Python's but having more efficient execution environments. Smalltalk fits the bill, and specifically SELF is a prominent derivative which offered an efficient execution engine back in the early 90s. Among the various optimizations in SELF, specifically type feedback in combination with inline caching seems particularly matching our first problem. Hölzle and Ungar [12] report performance speedups by a factor of 1.7 using type feedback, and give advice that languages having generic operators are ideally suited for optimization with type feedback. Application of type feedback requires that for a pair of operand-types the target of the actually selected operation implementation is cached, such that consecutive calls can directly jump there, when operand-types match the cached pair (cf. Figure 8).

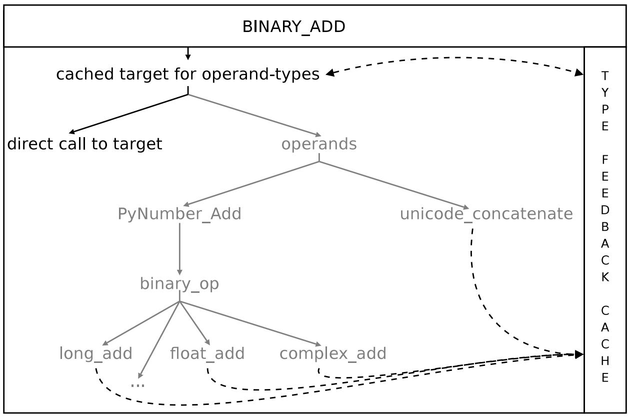


Fig. 8. Type feedback for Python's BINARY_ADD instruction. The gray-colored part is the system default look-up routine from Figure 5. The dashed arrows represent querying the type feedback cache (upper bidirectional arrow), and updating the cache with new target addresses after running through the system look-up routine (lower unidirectional arrows).

In the second case (b), the same technique can be applied. This requires that the actual operation implementation would be sub-structured to the following steps:

(i) Try to use the fastest possible native machine method

(ii) If i fails/overflows, apply unbounded range software algorithm

By encoding this information into separate types—e.g. int, long, arbitrary—, the type feedback infrastructure of our first problem (a) can be reused. In such a case, a positive check against machine boundaries and overflow errors, calling downwards the chain of most-general implementations and updating the cache for subsequent calls is necessary.

Aside from these optimization techniques, we want to mention a subtle issue that comes up when comparing high abstraction level instructions with low abstraction level instructions. When we compare the addition example of Java and Python, we find that JVM's integer addition bytecodes, `IADD` and `LADD`, are bound by a maximum range of representable numbers—32-bit for integers, and 64-bit for long integers respectively—whereas Python's `BINARY_ADD` implementation is not. Since the JVM does not offer unbounded range mathematics at the virtual machine level, it is necessary to leverage library functionality—in our case java.math.BigInteger—in order to have a 1 : 1 correspondence between the integer addition of both languages. In Java's case a call to `IADD`, or `LADD` for that matter, would be substituted by a invocation of a software algorithm for unbounded range mathematics of java.math.BigInteger—probably similar to the one implemented for Python's `BINARY_ADD`, or `long_add` respectively. This implies that even though a similar algorithm might be used, their difference in implementation level is significant: in Java we need a library, which generates multiple bytecodes for implementation of the unbounded range addition, whereas the Python compiler still emits just a single `BINARY_ADD` instruction.

Consequently, a high abstraction level sometimes can be considered as an optimization technique itself, since it can save a considerable amount of emitted low abstraction level instructions—we could probably say that this is a derivative, or special case, of the superinstruction optimization technique [9]. In conclusion, this example also illuminates that the instruction set architecture is also of significant importance for the performance of virtual machines—probably we can also reuse the analogy of hardware machine instruction set architecture, by recognizing the terms RISC and CISC in context with our classification, e.g. low and high abstraction level interpreters.

5 Related Work

Romer et al. [19] provide an analysis for interpreter systems. Their objective was to find out whether interpreters would benefit from hardware support. However, they conclude that interpreters share no similarities, and therefore hardware support was not meaningful. Among other measurements, they collected average native instructions per bytecode (Section 3 of their paper). This is a sort of a black-box view on our classification scheme based on comparable source code examples from bytecode implementations. Finally, there is no link to optimization techniques, too.

Contrary to Romer et al. [19], Ertl and Gregg [8] found that at least within the subset of efficient interpreters, hardware support in the form of branch target buffers would significantly improve performance for the indirect branch costs incurred in operation dispatch. Their in-depth analysis by means of simulation of a simple MIPS CPU found that the indirect branching behavior of interpreters is a major cause for slowdowns. Another important result of Ertl and Gregg is that a Prolog implementation, the Warren Abstract Machine based YAP [4], is very efficient, too: This implies that there is no immanent performance penalty associated with dynamically typed programming languages. Their class of efficient interpreters maps perfectly well to our category of low abstraction level virtual machines.

Adding to their set of efficient interpreters, they also provide results for the interpreters of Perl, and Xlisp—both of which achieve results that do not fit within the picture of efficient interpreters. This is where we introduce the concept of high abstraction level interpreters, and how it correlates to optimization techniques.

Interestingly, for Xlisp Ertl and Gregg mention the following: "*We examined the [Xlisp] code and found that most dynamically executed indirect branches do not choose the next operation to execute, but are switches over the type tags on objects. Most objects are the same type, so the switches are quite predictable.*" This directly translates to our situation with high abstraction level interpreters (Section 4).

In his dissertation, Hölzle also notes that a problem for an efficient SELF interpreter would be the abstract bytecode encoding of SELF [11], with a point in case on the `send` bytecode, which is reused for several different things. Interestingly, Hölzle observes, that instruction set architecture plays a very important role for the virtual machine, and conjectures that a carefully chosen bytecode instruction set could very well rival his results with a native code compiler.

6 Conclusions

We introduced the classes of high and low abstraction levels for interpreters, and categorized some interpreted systems into their corresponding classes. Using them, we subjected various known optimization techniques for their relative optimization potential. Techniques that achieve very good speedups on low abstraction level interpreters do not achieve the same results for high abstraction level virtual machines. The reason for this is that the ratio of native instructions needed for operation execution vs. the native instructions needed for dispatch, and therefore their relative costs changes. In low abstraction level interpreters the ratio usually is $1 : n$, i.e., many operations can be implemented using just one machine instruction, but dispatch requires n instructions, which varies according to the dispatch technique applied, and is costly because of its branching behavior. Quite contrary for high abstraction level interpreters: here operation execution usually consumes much more native instructions than dispatch does, which lessens the implied dispatch penalties.

Our classes are not mutually exclusive, an interpreter can have both, low and high abstraction level instructions. For our classes of high abstraction level interpreters, exemplified by Python and Lua—but conjectured to be true for Perl and Ruby, too—type feedback looks particularly promising. When faced with other programming languages but the same situation, i.e., a discrepancy in expected and reported speedups for low abstraction level techniques, our mileage may vary. In such a situation, only detailed analysis of an interpreter's execution profile can tell us where most time is spent and which techniques are most promising with regard to optimization potential.

In closing, we want to mention that our objectives are to demonstrate the relative optimization potential for different abstraction levels between an interpreter's virtual machine instruction set and the native machine it runs on.

Acknowledgement

I want to thank Jens Knoop and Anton Ertl for their feedback and comments on earlier versions of this paper, which helped to clarify issues and to improve the presentation. I would also like to thank the anonymous reviewers for their helpful comments and remarks.

References

[1] "IVME '04: Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04)," ACM, New York, NY, USA, 2004, General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

[2] Bell, J. R., *Threaded code*, Communications of the ACM **16** (1973), pp. 370–372.

[3] Botlan, D. L. and A. Schmitt, "The OCaml System–Implementation," (2001), http://pauillac.inria.fr/~lebotlan/docaml_html/english/.

[4] CRACS, *Yap–Yet Another Prolog*, http://www.dcc.fc.up.pt/~vsc/Yap/.

[5] Ertl, M. A., *Threaded code variations and optimizations*, in: *EuroForth*, TU Wien, Vienna, Austria, 2001, pp. 49–55.

[6] Ertl, M. A. and D. Gregg, *The behavior of efficient virtual machine interpreters on modern architectures*, in: *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference on Parallel Processing* (2001), pp. 403–412.

[7] Ertl, M. A. and D. Gregg, *Optimizing indirect branch prediction accuracy in virtual machine interpreters*, in: *SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)* (2003), pp. 278–288.

[8] Ertl, M. A. and D. Gregg, *The structure and performance of efficient interpreters*, Journal of Instruction-Level Parallelism **5** (2003).

[9] Ertl, M. A. and D. Gregg, *Combining stack caching with dynamic superinstructions*, [1], pp. 7–14, General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

[10] Gosling, J., B. Joy, G. Steele and G. Bracha, "Java Language Specification, Second Edition: The Java Series," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] Hölzle, U., "Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming," Ph.D. thesis, Stanford, CA, USA (1995).

[12] Hölzle, U. and D. Ungar, *Optimizing dynamically-dispatched calls with run-time type feedback*, in: *SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, 1994, pp. 326–336.

[13] INRIA, *OCaml*, http://caml.inria.fr.

[14] Matsumoto, Y., *Ruby*, http://ruby-lang.org.

[15] Piumarta, I. and F. Riccardi, *Optimizing direct threaded code by selective inlining*, in: *SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI '98)* (1998), pp. 291–300.

[16] PUC-Rio, *Lua*, http://www.lua.org.

[17] Python Software Foundation, *Python*, http://www.python.org.

[18] python.org, "Python v3.0 documentation," (2008), http://docs.python.org/3.0/.

[19] Romer, T. H., D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad and H. M. Levy, *The structure and performance of interpreters*, in: *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (1996), pp. 150–159.

[20] Shi, Y., K. Casey, M. A. Ertl and D. Gregg, *Virtual machine showdown: Stack versus registers*, ACM Transactions on Architecture and Code Optimization **4** (2008), pp. 1–36.

[21] The Perl Foundation, *Perl*, http://www.perl.org.

[22] Vitale, B. and T. S. Abdelrahman, *Catenation and specialization for Tcl virtual machine performance*, [1], pp. 42–50, General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.