View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Inline Caching Meets Quickening

Stefan Brunthaler

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
brunthaler@complang.tuwien.ac.at

Abstract. Inline caches effectively eliminate the overhead implied by dynamic typing. Yet, inline caching is mostly used in code generated by just-in-time compilers. We present efficient implementation techniques for using inline caches without dynamic translation, thus enabling future interpreter implementers to use this important optimization technique—we report speedups of up to a factor of 1.71—without the additional implementation and maintenance costs incurred by using a just-in-time compiler.

1 Motivation

Many of the currently popular programming language interpreters execute without dynamic code generation. The reason for this lies in their origins: many of these languages were implemented by single developers, who maintained their—often extensive—standard libraries, too. Since implementing just-in-time compilers is prohibitively expensive in terms of additional complexity and increasing required maintenance efforts, it is usually not considered to be a viable implementation option. Perl, Python, and Ruby are among the most popular of these programming languages that live without a dynamic compilation subsystem, but, nevertheless, seem to be major drivers behind many of the advances in the Internet's evolution.

In 2001, Ertl and Gregg [8] found that there are certain optimization techniques for interpreters, e.g., threaded code[1], [1,7] that cause them to perform at least an order of magnitude better than others. While interpreters using the threaded code optimization, such as the OCaml and Forth interpreters performed within a slowdown factor of up to 10 when compared with an optimizing native code compiler, other interpreters, such as the Perl and Xlisp interpreters, which were not using similar optimization techniques performed considerably worse: a slowdown of an additional factor of 100 when compared with efficient interpreters [8]. These striking results provided the motivation for us to investigate

[1] Please note that in the context of this paper, threaded code does not carry its usual meaning related to multi-threaded programming; it refers exclusively to a technique that reduces the overheads in interpreter instruction dispatch: instead of using the well-known switch-based dispatch technique, a threaded code interpreter uses an indirect jump to the next instruction.

© Springer-Verlag Berlin Heidelberg 2010

the case for inefficient interpreters more closely. Our analysis of the Python 3.0 interpreter [2] indicates that due to its nature, applying these promising optimization techniques results in a comparatively lower speedup than would be expected based on the reported figures. Vitale and Abdelrahman report cases where optimizing the interpreter's dispatch overhead in the Tcl interpreter actually results in a slowdown [19].

This is due to the differing abstraction levels of the respective interpreters: while the Java virtual machine [17] reuses much of the native machine for operation implementation—i.e., it is a low abstraction-level virtual machine—, the interpreters of Perl, Python, and Ruby have a much more complex operation implementation, which requires often significantly more native machine instructions; a characteristic of high abstraction-level interpreters. In consequence, optimization techniques that focus on minimizing the overhead in dispatching virtual machine instructions have a varying optimization potential with regard to the abstraction level of the underlying interpreter. In low abstraction-level virtual machines the overhead in instruction dispatch is big, therefore using threaded code is particularly effective, resulting in reported speedups of up to a factor of 2.02 [8]. On the other hand, however, the same techniques achieve a much lower speedup in high abstraction-level interpreters: the people implementing the Python interpreter report varying average speedups of about 20% in benchmarks, and significantly less (about 7%-8%) when running the Django[2] template benchmarks—a real world application.

Upon further examination of the operation implementation in the Python 3.0 interpreter, we find that there is substantial overhead caused by its dynamic typing—a finding that was true for Smalltalk systems more than 25 years ago. In 1984, however, Deutsch and Schiffman [6] published their seminal work on the "Efficient Implementation of the Smalltalk-80 System." Its major contributions were dynamic translation and inline caching. Subsequent research efforts on dynamic translation resulted in nowadays high performance just-in-time compilers, such as the Java Virtual Machine [17]. Via polymorphic inline caches and type feedback [13], inline caching became an important optimization technique for eliminating the overhead in dynamic typing. Unfortunately, inline caches are most often used together with dynamic translation. This paper presents our results on using *efficient* inline caching without dynamic translation in the Python 3.1 interpreter.

Our contributions are:

- We present a simple schema for efficiently using inline caching without dynamic translation. We describe a different instruction encoding that is required by our schema (Section 2), as well as our implementation of profiling to keep memory requirements imposed by our new instruction encoding at a minimum (Section 2.1).
- We introduce a more efficient inline caching technique using instruction set extension (Section 3) with quickening (Section 3.1).

[2] Django is a popular Python Web application development framework.

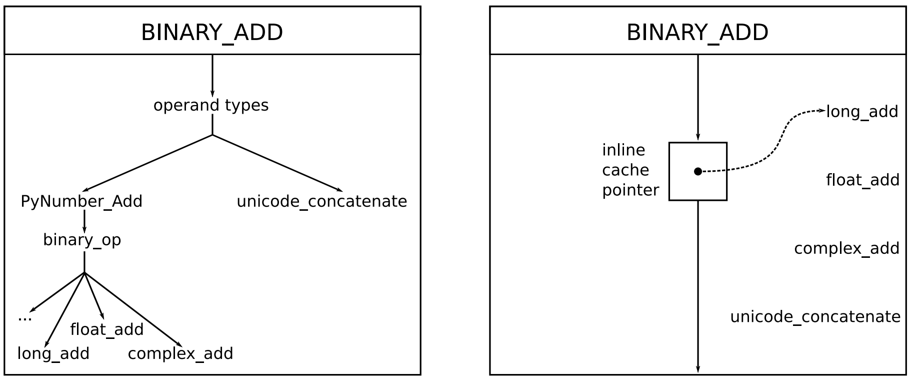
– We provide detailed performance figures on how our schemes compare with respect to the standard Python 3.1 distribution on modern processors (Section 4). Our advanced technique achieves a speedup of up to a factor of 1.71. Using a combination of inline caching and threaded code results in a speedup of up to a factor of 1.92.

2 Basic Inline Caching without Dynamic Translation

In dynamically typed programming language implementations, the selection of the actual operation implementation for any given instruction depends on the actual types of its operands. We call the function that has the operand-types as its domain, and the addresses of the corresponding operation implementations as its range, the *system default look-up routine*. In 1984, Deutsch and Schiffman describe their original version of inline caching [6]. During their optimization efforts on the Smalltalk-80 system, they observe a "*dynamic locality of type usage*": for any specific occurrence of an instruction within a given function, the types of the actual operands for that instruction are very likely to remain constant across multiple interpretations of that specific instruction occurrence. Consequently, the result of calling the system default look-up routine remains constant with exactly the same likelihood. Therefore, using this observation, Deutsch and Schiffman use their dynamic translation scheme to rewrite native call instructions from calling the system default look-up routine to directly calling the result of invoking the system default look-up routine. This inline caching effectively eliminates the overhead in dynamic typing. As with every caching technique, however, we need a strategy for detecting when the cache is invalid and what we should do when this occurs. For detecting an invalid inline cache entry, the interpreter ensures that the optimized call depends upon the current class operand—its *receiver*—having the same address as it did when the cache element was initialized. If that condition does not hold, the interpreter calls the system default look-up routine instead and subsequently places its result in the inline cache element.

With the notable exception of rewriting native instructions, the previous paragraph does not indicate any prerequisites towards a dynamic translation schema. In fact, several method look-up caches—most often hash-tables—have been used in purely interpretative systems in order to cache a target address for a set of given instruction operands. If the look-up cache contains a valid target address, the interpreter uses an indirect branch instruction to call it. The premise is that using an indirect branch is less expensive than calling the system default look-up routine. In case of a cache-miss, the interpreter calls the system default look-up routine and places its returned address in the cache. Thus, using an indirect call instruction (i.e., function pointers in C) eliminates the necessity of having a dynamic translator at all.

Still, using hash-table based techniques is relatively expensive: you need to deal with hashing in order to efficiently retrieve the keys, with collisions when placing an element in the hash table, etc. However, we show that we can



(a) Ad-hoc polymorphism in Python 3.x. (b) Basic inline caching `long_add`.

Fig. 1. Illustration of our basic inline caching technique compared to the standard Python 3.1 ad-hoc polymorphism

completely eliminate the need for look-up caches, too. A just-in-time compiler generates dedicated native machine instructions for a given sequence of bytecodes, for example a function body. Subsequently, the just-in-time compiler optimizes these native machine instructions, incorporating information obtained during previous invocations of that sequence. In a sense, the just-in-time compiler is generating more efficient derivatives of the interpreter instructions it actually used to "derive" the native machine code from. For example, the inline caching optimization allows the just-in-time compiler to leverage the "*dynamic locality of type usage*" by short-fusing the corresponding call instruction. Fortunately, we can project this information back to the purely interpretative level: by storing an additional machine word for every instruction within a sequence of bytecodes, we can lift the observed locality to the interpreter level. Consequently, we obtain a dedicated inline cache pointer for every interpreter instruction, i.e., instead of having immutable interpreter operation implementations, this abstraction allows us to think of specific instruction *instances*. At the expense of additional memory, this gives us a more efficient inline caching technique that is more in tune with the original technique of Deutsch and Schiffman [6], too.

Figure 1(a) shows how the Python 3.1 interpreter resolves the ad-hoc polymorphism in the `BINARY_ADD` instruction. Here, an inline cache pointer would store the addresses of the leaf functions, i.e., either one of `long_add`, `float_add`, `complex_add`, and `unicode_concatenate`, and therefore an indirect jump circumvents the system default look-up path (cf. Figure 1(b)). The functions at the nodes which dominate the leaves need to update the inline cache element. In our example (cf. Figure 1), the `binary_op` function needs to update the inline cache pointer to `long_add`. If there is no such dominating function (cf. Figure 1(a), right branch to `unicode_concatenate`), we have to introduce an auxiliary function that mirrors the operation implementation and acts as a dedicated system default look-up routine for that instruction.

Even though Deutsch and Schiffman [6] report that the "*inline cache is effective about 95% of the time*," we need to account for the remaining 5% that invalidate the cache. We change the implementation of the leaf functions to check whether their operands have their expected types. In case we have a cache miss, say we called `long_add` with `float` operands for example, a call to `PyNumber_Add` will correct that mistake and properly update the inline cache with the new information along the way; continuing our example, the address of the `float_add` function would be placed in the inline cache.

```
PyObject *long_add(PyObject *v, PyObject *w) {
    if (!(PyLong_Check(v) && PyLong_Check(w)))
        return PyNumber_Add(v, w);

    /* remaining implementation unchanged */
    ...
}
```

Finally, we present our implementation of using inline cache pointers in the context of the Python 3.1 interpreter (cf. Figure 2). It is worth noting, however, that we actually do not require any specific Python internals. Our technique can in fact be applied in general to many other interpreters as well.

```
TARGET(BINARY_SUBTRACT)                  TARGET(BINARY_SUBTRACT)
    w = POP();                               w = POP();
    v = TOP();                               v = TOP();
    x = PyNumber_Subtract(v, w); <->         x = (*ic_ptr)(v, w);
    Py_DECREF(v);                            Py_DECREF(v);
    Py_DECREF(w);                            Py_DECREF(w);
    SET_TOP(x);                              SET_TOP(x);
    if (x != NULL) DISPATCH();               if (x != NULL) DISPATCH();
    break;                                   break;
```

Fig. 2. Implementation of basic inline caching technique

Figure 2 shows how we replace the call to the system default look-up routine (in this case `PyNumber_Subtract`) by an indirect call instruction using the `ic_ptr` pointer variable. Aside from this change, we use this example to introduce some Python internals.

First, we shed some light on the meaning of `TARGET`: it is a pre-processor macro definition, which the Python implementation uses to allow for conditional compilation of whether the interpreter should use a switch-based dispatch technique or the more efficient threaded-code based dispatch technique. When configured with the switch based technique, this will expand to a C `case` statement, while configuration for threaded code leads to an expansion with an additional label (using a `TARGET_` prefix; for example, `TARGET_BINARY_SUBTRACT`) and additional instruction decoding.

Similarly, DISPATCH is a pre-processor macro definition, too. Depending on the selected dispatch method, it will either expand to a continue statement or a table based look-up of the address of the next bytecode implementation.

POP, and TOP are macro definitions for accessing the operand stack, and finally, Py_DECREF is a macro for decrementing the reference count of the argument.

The Python interpreter has a conditional instruction format: if an instruction has an operand, the two consecutive bytes contain its value. Otherwise, the next byte contains the next instruction. Hence, two instructions in the array of bytecodes are not necessarily adjacent. This irregular instruction form complicates not only instruction decoding (cf. Figure 3), but updating of the inline cache pointers, too.

```
opcode = *ip++;
oparg = 0;
if (opcode >= HAVE_ARGUMENT)
    oparg = (ip+= 2, (ip[-1]<<8) + ip[-2])
```

Fig. 3. Complex decoding of irregular instruction format

Our implementation removes this obstacle by encoding the instruction opcode and its argument into one machine word, and using the adjacent machine word for the inline cache pointer. Thus, all instructions have even offsets, while the corresponding inline cache pointers have odd offsets (cf. Figure 4).

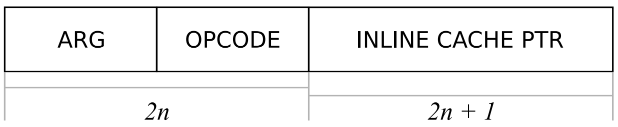


Fig. 4. Changed instruction format

In addition to being a more efficiently decode-able instruction format, this enables us to easily update the inline cache pointer for any instruction without having any expensive global references to that instruction. One minor change is still necessary, however: since we have eliminated the argument bytes from our representation, jumps within the bytecode contain invalid offsets—therefore we relocate these offsets to their new destinations. Our previous work provides more details concerning the relocation process [3].

A downside of this basic technique is that it requires significantly more memory space: instead of just one byte for the instruction opcode, this technique requires two machine words per instruction. One way to circumvent the additional memory requirements would be to add an inline cache word only to instructions that benefit from inline caching. While this is certainly possible, this is a sub optimal solution: as we have outlined above, this would invariably lead to an irregular instruction encoding with its presented downsides. Moreover, as we will see in later sections, many instructions will actually benefit from inline caching, which supports our argument against this approach.

Another way to compensate for this additional memory requirements is to use profiling. Using the classic optimization approach of trading space for time, profiling allows us to limit exuberant space requirements by actively monitoring execution and selectively choosing which parts to optimize for maximum payoff. The following section outlines our simple, low-overhead profiling implementation.

2.1 Implementation of Profiling

We implement the presented basic inline caching technique in a separate version of the interpreter dispatch routine and use profiling to decide which version handles the current activation/call. Choosing which routine should be invoked per default among those two routines is an important choice. We rename the original default interpreter routine and use the optimized version as the default call target. The rationale behind this choice is that the additional call overhead for calling the system default interpreter routine for all the infrequent cases is negligible, but for the frequently used pieces of code it is not.

We add a counter field to the code object (`PyCodeObject`) that we increment at the beginning of the system default interpreter dispatch routine. Once this counter reaches a definable threshold of currently 500 activations, we stop diverting to the system default interpreter routine and continue execution using the optimized interpreter routine. However, call frequency is but one indicator of heavy activity on which we base optimization decision. Our second indicator is the total amount of already executed instructions within the current interpreter dispatch loop. Since counting every execution would be expensive, we approximate the previously executed instructions and switch over to the optimized dispatch routine if our approximation reaches a threshold of 10,000 instructions. This captures code that is infrequently called but executes a lot of interpreter instructions while it is active and therefore leaves a considerable footprint on the overall execution costs.

Our approximation of the number of executed instructions is based on Python's own "instruction counter": approximately every 100 instructions (configurable in a counter variable called `_Py_CheckInterval`) the Python interpreter does various things like handling pending (system) calls. We add code that counts these occurrences. Once we reach a configurable threshold of 15,000 instructions, we transfer the currently active execution from the system default interpreter routine to the optimized interpreter routine. For the optimized interpreter routine to resume interpretation, we need to capture the current execution state of the system default interpreter routine. The stack pointer is one part of the current execution state and can be invariably used by both interpreter routines. However, the other part of the current execution state involves the instruction pointer, which needs to be relocated to correspond to the new instruction encoding. Other instruction-pointer dependent data, for example the encoding of blocks of instructions for exceptions and loops need to be relocated, too. After we finish these relocating operations, we can safely invoke the optimized dispatch routine, which will create and initialize the improved instruction encoding and resume execution.

Summing up, this describes a basic and simple, yet more efficient version of an inline caching technique for interpreters without dynamic translation. Combined with a simple profiling technique the increased memory requirements implied by this technique can be kept at a minimum. On average, the optimized interpreter requires an additional 90 KB of memory on a 32bit Intel Atom CPU, and an additional 122 KB of memory on a 64bit PowerPC 970 system.

3 Instruction-Set Extension

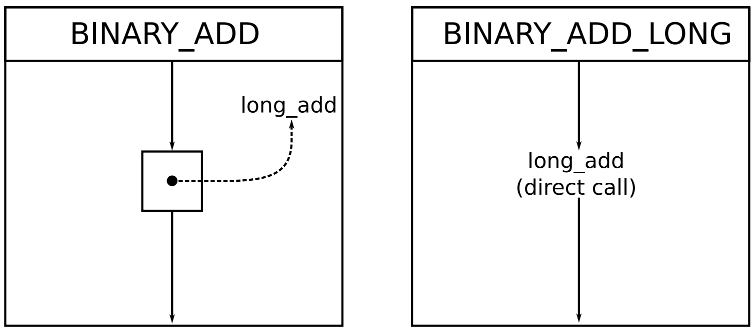
Our basic inline caching technique from Section 2 introduces an additional indirect branch for every instruction that uses an inline cache. Though this indirect branch is certainly almost always cheaper than calling the system default look-up routine, we can improve on that situation and remove this additional indirect branch completely.

The new instruction format enables us to accommodate a lot more instructions than the original one used in Python 3.1: instead of just one byte, the new instruction format encodes the opcode part in a half-word, i.e., it enables our interpreter to implement many more instructions in common 32 bit architectures (2^{16} instead of 2^{8}). Although a 64 bit architecture could implement 2^{32} instructions, for practical reasons and limited instruction cache size it is unrealistic to even approach the limit of 2^{16} interpreter instructions.

The original inline caching technique requires us to rewrite a call instruction target. In an interpreter without a dynamic translator this equals rewriting an interpreter instruction; from the most generic instance to a more specific derivative. Figure 5 shows how we can eliminate the inline cache pointer all together by using a specialized instruction that directly calls the `long_add` function.

Rewriting virtual machine instructions is a well known technique. In the Java virtual machine, this technique is called "quick instructions" [17]. Usually, *quickening* implies the specialization of an instruction towards an operand *value*, whereas our interpretation of that technique uses specialization with respect to the *result* of the system default look-up routine—which is a function of the operand types instead of their values. Another significant difference between the well known application of quickening in the Java virtual machine and our technique lies in the quickening frequency: the JVM uses quickening for initialization purposes, i.e., the actual quickening of a bytecode instruction happens only once for each occurrence. Aside from the initial quickening based on the results of resolving dynamic types of the actual operands, our technique requires re-quickening whenever the inline cache is invalid, viz., in the remaining 5% of cases when the "dynamic locality of type usage" does not hold.

Though quickening can be done regardless of the instruction format of the interpreter, we implement our rewriting technique based on the basic inline caching scheme of Section 2. Our primary motivation for keeping the new instruction format is that it enables us to accommodate much more optimized derivatives: if we would restrict ourselves to the 255 instructions representable using the original Python bytecode, we would have to carefully choose which derivatives to



(a) Basic inline caching `long_add`. (b) New instruction with direct call.

Fig. 5. Instruction-set extension illustrated for operands having `long` type

implement based on some pre-defined profiling results. As an added benefit, our new instruction format allows for much more effective decoding of instructions.

The quickening approach is somewhat the opposite of what we described in the previous section. Which approach performs better depends on the underlying native machine hardware. Since the quickening approach increases the code size of the interpreter dispatch loop, this may cause instruction-cache miss penalties on architectures with small instruction caches. For such architectures with small instruction caches, the basic technique of Section 2 might actually perform better because it causes fewer instruction cache misses. In fact, using the basic technique enables us to coalesce multiple instruction implementations if their implementations differ only in the original direct call to the operation implementation—which is precisely the case for all binary operations in the Python implementation. On modern desktop and server hardware, which offers larger instruction cache sizes, however, the quickening approach is clearly preferable, since it completely removes the overheads caused by dynamic typing. Figuratively speaking, both techniques are opposite ends on the same spectrum, and the actual choice of implementation technique largely depends on direct evaluation on the target hardware. Finally, both approaches are not mutually exclusive: it turns out that a combination of both approaches enables us to implement optimizations that could not be realized using either one of the techniques exclusively (cf. Subsection "Optimizations based on combining both approaches" in the next Section 3.1).

3.1 Inline Caching via Quickening

In Python, each type is a C `struct` that contains a list of function pointers that can be used on instances of that type. This list of function pointers contains, among others, the following three sub-structures (which are C `struct`s themselves) that allow a type implementer to provide *context-dependent* functions for use on instances of that type:

1. Scalar/numeric context: this context captures the application of binary arithmetical and logical operators to operands of a given type. Examples include: add, subtract, multiply, power, floor, logical and, logical or, logical xor, etc.
2. List context: this context captures the use of an instance of a type in list context, for example list concatenation, containment, length, repetition (i.e., operation of a list and a scalar).
3. Map context: this context captures the use of an instance of a type in map context. Operations include the assignment of keys to values in a map, the fetching of values given a key in the map, and the length of the map.

Table 1. Types with context-dependent functions

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

For each of the types in Table 1, we determine whether it implements a specific scalar-/list-/map-context dependent function. For use in the scalar/numeric context, each type has a sub-structure named `tp_as_number`, which contains a list of pointers to the actual implementations, for example the `nb_add` member points to the implementation of the binary addition for that type. If we want to call, for example, the `long_add` function of Python's unbounded range integer type `PyLong_Type`, the "path" would be: `PyLong_Type.tp_as_number->nb_add`. For the list context, we use the sub-structure `tp_as_sequence`; for the map context the corresponding `tp_as_mapping` identifier.

We have a short Python program in a pre-compile step that generates the necessary opcode definitions and operation implementations (cf. Figure 6) for several types. Currently, the program generates 77 optimized derivatives for several bytecode instructions, amounting to 637 lines of code (as measured by `sloccount`). These specialized instructions are generated in a separate file, which we include in the optimized interpreter dispatch loop. Consequently, the interpreter's dispatch loop implementation remains largely untouched by this optimization.

```
TARGET(INCA_LONG_ADD)
    w = POP();
    v = TOP();
    x = PyLong_Type.
        tp_as_number->nb_add(v, w);
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    if (x != NULL) DISPATCH();
    break;
```

```
if (v == PyFloat_Type.
         tp_as_number->nb_add)
    *ip= INCA_FLOAT_ADD;
```

Fig. 6. Examples for code we generate: the *left* side shows an optimized derivative instruction for integer addition; the *right* side shows a quickening example

Apart from the generation of the derivatives themselves, we need to take care of the actual quickening, i.e., the actual rewriting of the instructions, too. In the previous Section 2, we already explained that we need to instrument suitable places to update the inline cache pointer. Our implementation has a function named `PyEval_SetCurCacheElement` that updates the inline cache. Since this function already takes care of updating the inline cache pointer of the current instruction, adding code that rewrites the opcode of the current instruction is easy. However, manually adding code for checking the 77 inline cache target functions is tedious, error-prone, and unnecessary: since our small generator has all the information necessary for creating the actual operation implementations, it already has the information necessary for creating the checks, too: the mapping of bytecode instructions to the addresses of operation implementation functions. Consequently, we extend the generator to provide the C statements necessary for quickening the instructions (cf. Figure 6, right column). By including these checks in the `PyEval_SetCurCacheElement` function, we can actually re-use the cache-miss strategy of the previous approach, too.

The types in Table 1 represent the most basic primitives of the Python language, i.e., they are not defined in modules of the standard library but represent the "core" of the language. Depending on the programs to be run in the interpreter, providing different instruction derivatives might be beneficial; without statistical evidence of real-world usage, however, it seems only natural to promote the most basic primitives to their dedicated instructions. In order to select which operations to implement, it is necessary to know which type implements which operations. One approach would be to parse the C code containing the type definitions and their bindings for several operations, represented as C `struct`'s. While certainly possible, our approach is much simpler: using the `gdb` [18] debugger, one can inspect data structures at runtime.

The format `gdb` uses when printing this information is already very close to Python data structure definitions. For example, Figure 7 displays the output of `gdb`'s print command with the Python floating point type structure (`PyFloat_Type`) as its argument on the left side, and the corresponding Python data structure definition on the right side. As we can see, converting `gdb`'s

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

Fig. 7. Example of the gdb output on the left side, and the corresponding Python data structure definition on the right side

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

Inline Caching the Iteration Instruction: Python has a dedicated instruction for iteration, `FOR_ITER`. This instruction expects the top-of-stack element to be an instance of an iterator—as is the result of executing the `GET_ITER` instruction. The `FOR_ITER` instruction takes the iterator top-of-stack element, and pushes the result of executing this iterator onto the operand stack, such that the result becomes the new top-of-stack element and the iterator object the second object on the operand stack. Executing the iterator means to invoke the function pointed to by the `tp_iternext` function pointer of its type structure, with the actual iterator instance as an argument (cf. the concrete implementation below). This dependency on the type of the top-of-stack element is similar to the application of inline caches in the previous section. The main difference between the two cases is that the previous section requires the selection of a matching operation implementation based on actual operand types, whereas the iteration instruction does not. Nevertheless, the same optimization can be applied in this case, too.

```
TARGET(FOR_ITER)
    v = TOP();
    x = (*v->ob_type->tp_iternext)(v);
    if (x != NULL) {
        PUSH(x);
        DISPATCH();
    }
    /* remaining implementation omitted */
```

There is a set of dedicated types for use with this construct, and we have extracted 15 additional instructions totaling 124 lines of code that replace the indirect call of the standard Python 3.1 implementation with a specialized derivative: for example the iterator over a range object, `PyRangeIter_Type`:

```
TARGET(FOR_ITER_RANGEITER)
    v = TOP();
    x = PyRangeIter_Type.tp_iternext(v);
    /* unchanged body */
```

Our previously mentioned generator takes care of generating the 15 optimized derivatives, too. However, having to compare against the 77 optimized derivatives for inline-cached operations in the previous section (using the extended `PyEval_SetCurCacheElement` function) when we only have to compare among the 15 specialized iteration instructions is not necessary at all. Therefore our generator generates the necessary C code that we include in the original `FOR_ITER` instruction to rewrite its occurrence to an optimized derivative, e.g., replace the current `FOR_ITER` instruction by the corresponding `FOR_ITER_RANGEITER` instruction.

Inline Caching the Call Instruction: The optimization of the `CALL_FUNCTION` instruction requires the most work. In his dissertation, Hölzle already observed the importance of instruction set design with a case in point on the `send` bytecode in the SELF interpreter, which he mentions to being too abstract for efficient interpretation [11]. The same observation holds true for the Python interpreter, too: there are only a few bytecodes for calling a function, and the compiler generates `CALL_FUNCTION` instructions most often. The same bytecode is used for multiple call targets:

– Python functions: calling convention requires special handling of parameter lists depending on the number of arguments passed to the target function.
– Python methods: because of dynamic binding, and taking care of the instance reference, this requires special handling, too.
– C functions: proper Python support requires C functions to be able to take named and/or variable arguments in a similar manner to Python functions/methods. Since C does not natively support both in the same uniform manner as Python handles them, we have to supply dedicated code to “fix up” argument handling accordingly.

Since we cannot provide inline caching variants for every possible combination of call types and the corresponding number of arguments, we decided to optimize frequently occurring combinations (cf. Table 2 for details).

We provide our own version of an instrumented standard look-up routine to rewrite the instructions if the matching number of arguments and type of calls occur. For the remaining, unoptimized calls, we provide another call instruction that uses the default look-up routine without our instrumentation code. Therefore, our instrumentation penalties occur only during the first invocation.

Unfortunately, because of their non-trivial nature, our generator cannot automatically generate the implementations for the optimized call instructions, i.e., we manually created the optimized implementations for each combination of Table 2. This, however, is much less complicated than it may seem at first: the inherent complexity in the operation implementation comes from the call-target type, i.e., whether a call target is a C function or a Python method (cf. rows in Table 2). The combinations within one row, i.e., the support for different numbers of arguments, are trivial—the separate implementations for a given type are almost identical replicas and the only difference comes from argument handling. In consequence, the actual implementation effort lies in the initial implementation of separate instructions for the four basic call targets with the remaining variations being just simple copies. Similarly to the previous inline caching implementations, we keep the inline cached call instruction implementations in a separate file which we include in the optimized interpreter dispatch loop. Our implementation of the optimized call instructions needs 491 lines of code.

Table 2. Specialized `CALL_FUNCTION` instructions

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

Optimizations Based on Combining Both Approaches: There are several instructions in Python that deal with look-ups in environments. These environments are represented by Python dictionaries, i.e., hash tables which contain bindings for given identifiers. Similar to many other stack frame layouts, a Python stack frame object (`PyFrameObject`) holds references to several environments. For example, the `LOAD_GLOBAL` instruction implements a precedence-based look-up procedure: first there is a look-up using the stack frame's reference to the hash table holding globally visible identifier-to-object bindings (`f->f_globals`). If no binding for a given identifier was found, Python implements a second look-up attempt using the stack frame's reference to the hash table holding entries for built-in objects (`f->f_builtins`):

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

Now, hash-table look-up using complex objects is an expensive operation, since the computation of hash keys for those objects and the list traversal to find and retrieve the actual object are necessary. If we can ensure that no destructive calls, i.e., calls invalidating an inline cached version, occur during the execution, we can cache the resulting object in our inline cache element of Section 2 and rewrite the instruction to a faster version:

```
TARGET(FAST_LOAD_GLOBAL)
    Py_INCREF(*ic_ptr);
    PUSH(*ic_ptr);
    DISPATCH();
```

For our current implementation we assume that optimized bytecode that does not contain any STORE_GLOBAL instructions is safe to use that optimization. Therefore, we do not quicken LOAD_GLOBAL instructions in the presence of STORE_GLOBAL instructions within the same optimized bytecode sequence. This naive assumption misses some optimization potential and is not generally applicable for all Python programs. Particularly Python programs that make heavy use of global variables will not be properly interpreted, since this straightforward invalidation mechanism is too aggressive for this type of programs. Yet, for our set of benchmarks, in addition to preliminary benchmarks using the Django web application framework—a comparatively big real world application—, using this naive assumption yields correct results. First analysis results indicate that these programs do not make heavy use of potentially unsafe STORE_GLOBAL instructions. We conjecture that this is due to the common programming practice of avoiding the use of global variables at all costs.

However, we present a sound and generally applicable invalidation mechanism that requires us to add a flag to each object. Since Python uses reference counting, we could re-use the reference count field for that purpose. Whenever we quicken a LOAD_GLOBAL instruction, we set the flag for this object and require the FAST_LOAD_GLOBAL implementation to check whether this flag is set. If the flag is set, we continue to execute the fast path in the FAST_LOAD_GLOBAL implementation; otherwise we jump to the implementation of the LOAD_GLOBAL instruction, which would re-quicken the instruction accordingly. This is however just one part of the invalidation scheme. The other part requires us to change the STORE_GLOBAL implementation: it retrieves the currently held object first and resets the flag before updating the corresponding slot with the new reference. The same optimization applies to the LOAD_NAME instruction.

Quickening and the Comparison Instruction: Depending on its operand value, Python's COMPARE_OP instruction chooses which comparison relation it is going to use. It calls the cmp_outcome function which implements comparator selection using a switch statement:

```
static PyObject *
cmp_outcome(int op, PyObject *v, PyObject *w) {
    int res = 0;
    switch (op) {
      case PyCmp_IS:     res = (v == w); break;
      case PyCmp_IS_NOT: res = (v != w); break;
      case PyCmp_IN:     res = PySequence_Contains(w, v);
                         if (res < 0) return NULL;
                         break;
      case PyCmp_NOT_IN: res = PySequence_Contains(w, v);
                         if (res < 0) return NULL;
                         res = !res;
                         break;
      case PyCmp_EXC_MATCH:
        /* more complex implementation omitted! */
```

We eliminate this switch statement for the four topmost cases by promoting them to dedicated interpreter instructions: `COMPARE_OP_IS`, `COMPARE_OP_IS_NOT`, `COMPARE_OP_IN`, `COMPARE_OP_NOT_IN`. This is somewhat similar to an optimization technique that is described by Allen Wirfs-Brock's article on design decisions for a Smalltalk implementation [16], where he argues that it might be more efficient for an interpreter to pre-generate instructions for every (frequent) pair of (opcode, oparg). Since the operand is constant for any specific instance of the `COMPARE_OP` instruction, we assign the proper dedicated instruction when creating and initializing our optimized instruction encoding. Hence, our implementation does not do quickening in this scenario—however, an implementation without our instruction format might need quickening to implement this. Furthermore, the four topmost cases are only simple comparison operations, more complex cases require the evaluation of an operand-type dependent comparison function. This function might very well be inline cached, similar to our previously presented scenarios. Please note, however, that our current implementation does not do this kind of advanced inline caching for the `COMPARE_OP` instruction.

4 Evaluation

We used several benchmarks from the computer language shootout game [9]. Since the adoption of Python 3.x is rather slow in the community, we cannot give more suitable benchmarks of well known Python applications, such as Zope, Django, and twisted. We ran our benchmarks on the following system configurations:

– Intel i7 920 with 2.6 GHz, running Linux 2.6.28-15 and `gcc` version 4.3.3. (Please note that we have turned off Intel's Turbo Boost Technology to have a common hardware baseline performance without the additional variances immanently introduced by it [14].)

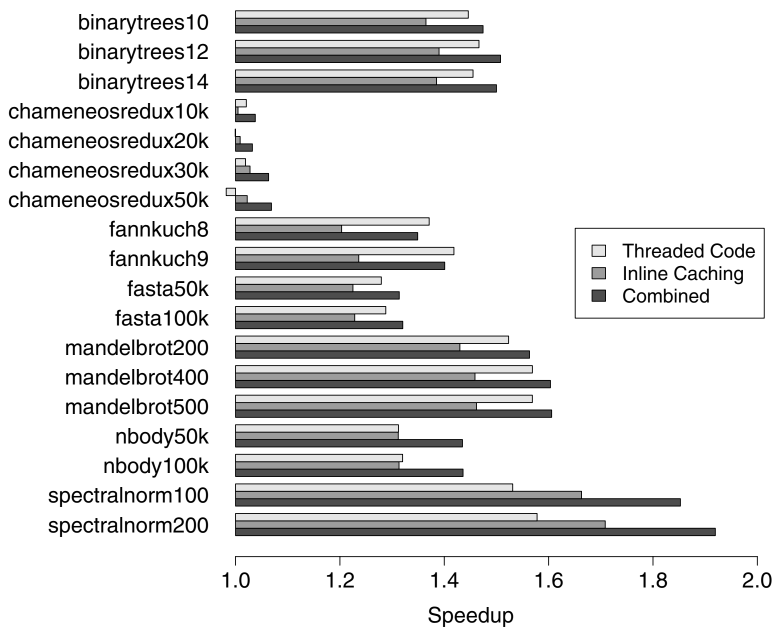


Fig. 8. Achievable speedups on various benchmarks on Intel i7 CPU

- Intel Atom N270 with 1.6 GHz, running Linux 2.6.28-18 and `gcc` version 4.3.3.
- IBM PowerPC 970 with 2.0 GHz, running Linux 2.6.18-4 and `gcc` version 4.1.2.

We used a modified version of the `nanobench` program of the computer language shootout game [9] to measure the running times of each benchmark program. The `nanobench` program uses the UNIX `getrusage` system call to collect usage data, e.g. the elapsed user and system times as well as memory usage of a process. We use the sum of both timing results, i.e., elapsed user and system time as the basis for our benchmarks. In order to account for proper measurement and cache effects, we ran each program 50 successive times and the reported data represent arithmetic averages over those repetitions.

Figures 8, 9, and 10 contain our evaluation results. We calculated the speedup by normalizing against the standard Python 3.1 distribution with threaded code and inline caching optimizations turned off. The labels indicate the name of the benchmark and its command line argument combined into one symbolic identifier. The measured inline caching technique represents the technique of Section 3.1 with the modified instruction format of Section 2. Therefore our inline caching interpreter employs all optimizations described in the previous sections. In particular, we used the naive invalidation mechanism described for optimization techniques that rely on a combination of both inline caching techniques.

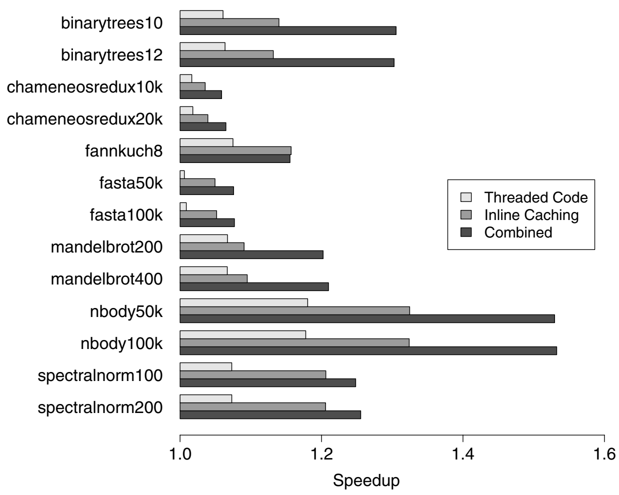


Fig. 9. Achievable speedups on various benchmarks on Intel Atom CPU

Since using our outlined correct invalidation mechanism consists of adding code that just checks the state of the flag, we conjecture that using the generally applicable invalidation mechanism yields similar results. Due to overly long running times on the Intel Atom platform, the range of arguments for the set of chosen benchmarks was slightly reduced for our evaluation (cf. Figure 9).

With the exception of the fannkuch benchmark, the combined approach of using threaded code with inline caching is always faster. On the Intel i7 920 (cf. Figure 8), the range of possible speedups is wide: from moderate improvements ($< 10\%$) in the case of the chameneosredux benchmark, to substantial improvements ($\geq 10\%$ and $< 60\%$) for the binarytrees, fannkuch, fasta, mandelbrot, and nbody benchmarks, and finally up to impressive speedups ($\geq 60\%$) in the spectralnorm benchmark.

The Intel Atom CPU has lower overall speedup potential (cf. Figure 9), but demonstrates that our inline caching technique performs better than the threaded code optimization on all of our benchmarks. The PowerPC CPU presents similar findings (cf. Figure 10): substantial speedups on most benchmarks, with the exception of the chameneosredux benchmark.

Interestingly, when comparing all three results we become aware of the variance of speedup achievable on the spectralnorm benchmark: while achieving the highest speedup of all benchmarks on the Intel i7, it does not achieve the full potential on the other CPUs: it is en par with the nbody benchmark on the PowerPC 970, but noticeably lower on the Atom CPU. Further investigation is necessary to uncover the cause of this effect. In addition to this, the comparison of the benchmark results on the Intel i7 and the PowerPC shows that there is one benchmark for each CPU where inline caching alone and a combination with threaded code is actually slower than a threaded-code only version (cf. fannkuch benchmark in Figure 8, and mandelbrot benchmark in Figure 10). Intuitively,

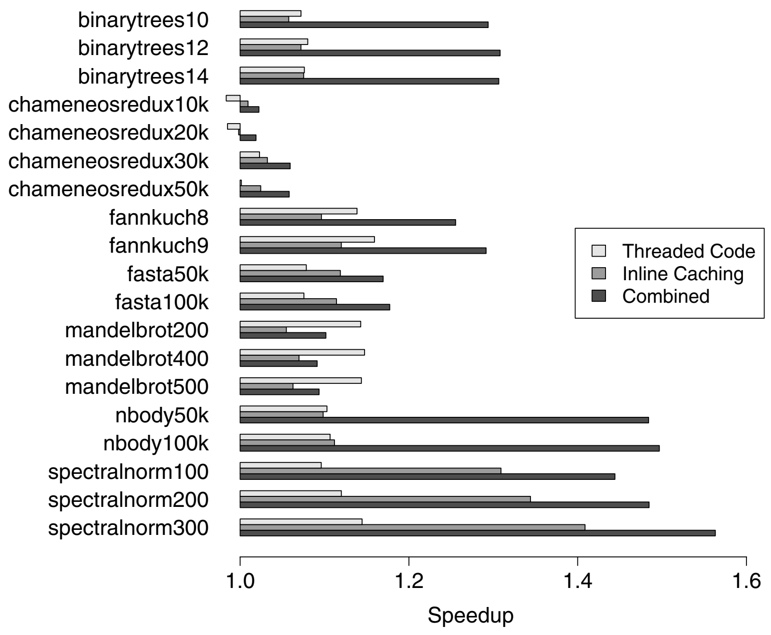


Fig. 10. Achievable speedups on various benchmarks on PowerPC CPU

the orthogonality of both optimizations implies better speedup potential when applied in combination rather than separately. This intuition is supported by our results on the Intel Atom CPU (cf. Figure 9): combining both optimization techniques almost always performs significantly better than one of the techniques alone. With the exception of the mandelbrot benchmark the same observation holds true for the binarytrees, nbody, and spectralnorm benchmarks on the PowerPC (cf. Figure 10). We are currently unsure why this effect occurs and will be investigating this issue in future work.

In the chameneosredux benchmark, we can see that threaded code execution can result in negative performance, too. This supports similar findings of Vitale and Abdelrahman on implementing threaded code for the Tcl interpreter [19]. Yet, this particular benchmark is inline caching friendly, and therefore a combination of both techniques results in a visible speedup.

4.1 Dynamic Frequency Analysis

In addition to the raw performance figures of the previous section, we want to quantify the effectiveness of our profiling system together with the available amount of “dynamic locality of type usage” [6] demonstrated by our benchmark programs. Therefore we instrumented the interpreter to count the executed instructions. Table 3 shows the amount of instructions executed during the corresponding benchmarks in the left column. The right column presents the percentage of instructions that were optimized among the overall instructions.

Unsurprisingly, we see that the optimization possibility depends very much on the actual benchmark. For those benchmarks where our technique achieves a very high speedup (cf. Figures 8, and 10), the utilization of optimized instructions is significantly higher. The converse observation holds, too: Figures 8, and 10 indicate that for the fannkuch benchmark, our technique is slower than threaded code and can even slow down a combined version—at least on the Intel i7 920. This is reflected by the exact same benchmark having the lowest utilization among all our results.

Table 3. Number of total interpreter instructions and percentage of optimized interpreter instructions

Benchmark	Argument	Total Instructions	Optimized Instructions
binarytrees	12	42.431.125	23.37%
chameneosredux	50.000	8.474.581	24.79%
fannkuch	8	7.865.843	18.80%
fasta	50.000	11.051.463	32.76%
mandelbrot	300	33.565.619	26.82%
nbody	50.000	68.974.547	38.57%
spectralnorm	100	12.963.104	37.07%
Median	—	—	26.82%
Mean	—	—	28.88%

5 Related Work

In his PhD thesis of 1994 [11], Hölzle mentions the basic idea of the data structure underlying our basic technique of Section 2. The major difference is that we are not only proposing to use this data layout for `send`—or `CALL_FUNCTION` instructions in Python's case—but for all instructions, since there is enough caching potential in Python to justify that decision. Hölzle addresses the additional memory consumption issue, too. We use a simple low-overhead invocation based counter heuristic to determine when to apply this representation, i.e., it is only created for code we know is *hot*. Therefore, we argue that the increased memory consumption is negligible—particularly when compared with the memory consumption of state-of-the-art just-in-time compilers.

In 2008, Haupt et al. [10] published a position paper describing details of adding inline caching to bytecode interpreters, specifically the Squeak interpreter. Their approach consists of adding dedicated inline caching slots to the activation record, similar to dealing with local variables in Python or the constant pool in Java. In addition to a one-element inline cache, they also describe an elegant object-oriented extension that enables a purely interpretative solution to polymorphic inline caches [12]. The major difference to our approach lies in the relative efficiencies of the techniques: whereas our techniques are tightly interwoven with the interpreter infrastructure promising efficient execution, their technique relies on less efficient target address look-up in the stack frame.

Regarding the use of look-up caches in purely interpretative systems, we refer to an article [5] detailing various concerns of look-up caches, including efficiency of hashing functions, etc., which can be found in “Smalltalk-80: Bits of History, Words of Advice” [16]. Kiczales and Rodriguez describe the use of per-function hash-tables in a portable version of common lisp (PCL), which may provide higher efficiency than single global hash tables [15]. The major difference to our work is that our inline cache does not require the additional look-up and maintenance costs of hash-tables. Our quickening based approach of Section 3 eliminates the use of indirect branches for calling inline cached methods, too.

Lindholm and Yellin [17] provide details regarding the use of quick instructions in the Java virtual machine. Casey et al. [4] describe details of quickening, superinstructions and replication. The latter technical report provides interesting figures on the performance of those techniques in a Java virtual machine implementation. The major difference to our use of instruction rewriting as described in Section 3 is that we are using quickening for inline caching. To the best of our knowledge there is no other work in that area. However, we share the use of a code generator to compensate for the increased maintenance effort, but do not use any form of superinstructions.

6 Conclusion

Inline caching is an important optimization technique for high abstraction-level interpreters. We report achievable speedups of up to a factor of 1.71 in the Python 3.1 interpreter. Our quickening based technique from Section 3 uses the instruction format described in Section 2. Therefore, the measured performance includes the compensation times for the profiling code and the creation of the new instruction format. However, it is possible to use the quickening based inline caching approach without the new instruction format—thereby eliminating the compensation overhead, which we expect to positively affect performance. Future work on such an interpreter will quantify these effects.

Using the quickening approach, a compiler might decide to inline function bodies of specialized derivative instructions. This will provide additional performance improvements at the cost of possible instruction cache penalties. Future work will investigate the potential of further inlining.

Efficient inline caching without dynamic translation is an optimization technique targeting operation implementation. Therefore, it is orthogonal to optimization techniques focusing on instruction dispatch and both techniques can be applied together. In the `spectralnorm` benchmark the application of both techniques results in a speedup of up to a factor of 1.92—only slightly lower than the maximum reported speedup of up to a factor of 2.02 achieved by efficient interpreters using threaded code alone [8].

Acknowledgments. I want to thank Urs Hölzle for providing details regarding the history of the basic technique of section 2. I am grateful to Anton Ertl for valuable discussions and feedback on earlier drafts of this paper. Furthermore, many thanks go to Jens Knoop, who consistently provided crucial guidance and advice on numerous topics, resulting in a much clearer presentation. Finally, I am deeply indebted to the many detailed comments and suggestions of the anonymous reviewers; addressing these gave the paper more than just finishing touches.

References

1. Bell, J.R.: Threaded code. Communications of the ACM 16(6), 370–372 (1973)
2. Brunthaler, S.: Virtual-machine abstraction and optimization techniques. In: Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2009). Electronic Notes in Theoretical Computer Science, vol. 253(5), pp. 3–14. Elsevier, Amsterdam (December 2009)
3. Brunthaler, S.: Efficient inline caching without dynamic translation. In: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010). ACM, New York (March 2010) (to appear)
4. Casey, K., Ertl, M.A., Gregg, D.: Optimizations for a java interpreter using instruction set enhancement. Tech. Rep. 61, Department of Computer Science, University of Dublin. Trinity College (September 2005), `https://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-61.pdf`
5. Conroy, T.J., Pelegri-Llopart, E.: An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In: Krasner [16], ch. 13, pp. 239–247 (1982)
6. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the Smalltalk-80 system. In: Proceedings of the SIGPLAN 1984 Symposium on Principles of Programming Languages (POPL 1984), pp. 297–302. ACM, New York (1984)
7. Ertl, M.A.: Threaded code variations and optimizations. In: EuroForth, TU Wien, Vienna, Austria, pp. 49–55 (2001)
8. Ertl, M.A., Gregg, D.: The structure and performance of efficient interpreters. Journal of Instruction-Level Parallelism 5, 1–25 (2003)
9. Fulgham, B.: The computer language benchmarks game, `http://shootout.alioth.debian.org/`
10. Haupt, M., Hirschfeld, R., Denker, M.: Type feedback for bytecode interpreters, Position Paper (ICOOOLPS 2007) (2008), `http://scg.unibe.ch/archive/papers/Haup07aPIC.pdf`
11. Hölzle, U.: Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. thesis, Stanford University, Stanford, CA, USA (1994)
12. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1992), pp. 21–38. Springer, London (1991)
13. Hölzle, U., Ungar, D.: Optimizing dynamically-dispatched calls with run-time type feedback. In: Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI 1994), pp. 326–336 (1994)
14. Intel: Intel turbo boost technology in Intel core microarchitecture (nehalem) based processors (November 2008), `http://download.intel.com/design/processor/applnots/320354.pdf?iid=tech_tb+paper`

15. Kiczales, G., Rodriguez, L.: Efficient method dispatch in PCL. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP 1990), pp. 99–105. ACM, New York (1990)
16. Krasner, G. (ed.): Smalltalk-80: bits of history, words of advice. Addison-Wesley Longman Publishing Co. Inc., Boston (1983)
17. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 1st edn. Addison-Wesley, Boston (1997)
18. Stallmann, R.M., Pesch, R.H., Shabs, S.: Debugging with GDB: The GNU source-level debugger. Free Software Foundation, 9th edn. (2009)
19. Vitale, B., Abdelrahman, T.S.: Catenation and specialization for Tcl virtual machine performance. In: Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME 2004), pp. 42–50. ACM, New York (2004)