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. Unfortunately, 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 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. Therefore, it was usually not an issue for them to substantially increase the a) the complexity and b) the maintenance efforts of their implementations by adding just-in-time compilers. 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 advances in the internet's evolution.

In 2001, Ertl and Gregg found that there are certain optimization techniques for interpreters, e.g., threaded code [1,5], that cause them to perform at least an order of magnitude better than others [6]. Actually, their examination of Perl finds that its interpreter is not particularly efficient—which led us to subsequently analyze a similar interpreter [2]. Our analysis of the Python 3.0 interpreter shows that because of the nature of its interpreter, the application of exactly those techniques that cause other interpreters to perform significantly better, results in a comparatively lower speedup. Vitale and Abdelrahman report cases where the application of threaded code in the Tcl interpreter actually results in a slowdown [16].

This is due to the differing abstraction levels of the respective interpreters: While the Java virtual machine [14] 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 virtual machine. 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 a factor of 2.02 [6]. On the other hand, however, the same techniques achieve a much lower speedup in higher 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[1] template benchmarks—a real world application.

In consequence, our examination of the operation implementation in the Python 3.x interpreter finds that there is substantial overhead caused by its dynamic typing—a finding that was true for Smalltalk systems 25 years ago. In 1984, however, Deutsch and Schiffman [4] 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. Via polymorphic inline caches and type feedback [11], inline caching became an important optimization technique by itself. 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).
- We subsequently introduce a more efficient inline caching technique using instruction set extension (Section 3) with quickening (Section 3.1).
- We provide detailed performance figures on how our schemes compare with respect to the standard Python 3.x distribution on modern processors (Section 4). Our advanced technique achieves a speedup of up to 1.71. Using a combination of inline caching and threaded code results in a speedup of up to 1.92.

2 Basic Inline Caching without Dynamic Translation

In 1984, Deutsch and Schiffman describe their original version of inline caching [4]. During their optimization efforts on the Smalltalk-80 system, they observe a "*dynamic locality of type usage*", i.e., for any bytecode instruction within a given function or method, a call to the system default lookup routine is very likely to evaluate to the same address as it did during its previous invocation. Using

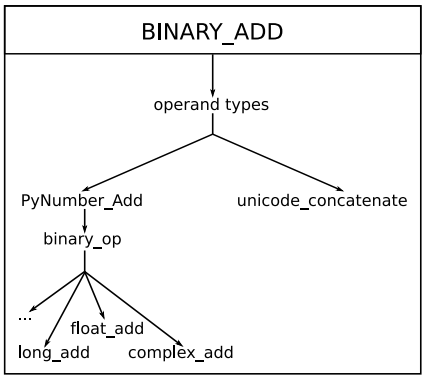
[1] Django is a popular Python Web application development framework.

this observation, they use their dynamic translation scheme to rewrite native machine call instructions from calling the system default lookup routine to directly calling its resulting target address. Like any caching technique, there is a strategy for detecting that the cache is invalid and what is to be done about that. For detecting an invalid inline cache entry, the interpreter ensures that the call circumventing the system default lookup routine depends upon the 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 lookup routine instead—which in turn takes care of properly updating the corresponding inline cache element.

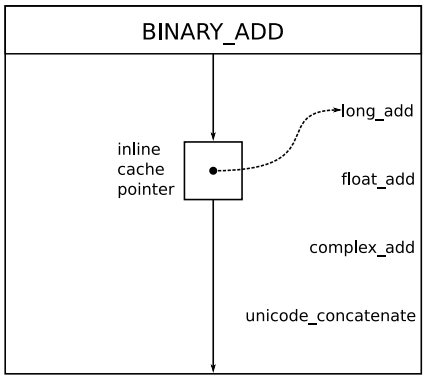
With the notable exception of the native instruction rewriting, the previous paragraph does not indicate any prerequisites towards a dynamic translation schema. In fact, several method lookup caches—most often hash-tables—have been used in purely interpretative systems in order to cache a target address for a set of given instruction operands. The interpreter requires an indirect branch to call the address found in the cache. The premise is that the indirect branch is less expensive than the call to the system default lookup routine. In case of a cache-miss, the interpreter calls the system default lookup routine and places its returned address in the cache. In consequence, using a function pointer via an indirect call instruction 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 collision when placing an element in the hash table, etc. However, we show that we can completely eliminate the need for method caches, too: During dynamic translation, a sequence of interpreter instructions is compiled to its corresponding native machine representation. Within this representation, the inline cache effectively specializes an interpreter instruction to a more efficient derivative—based on the "*dynamic locality of type usage*". Fortunately, we can project this information back at 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 [4], too.

Figure 1(a) shows how the ad-hoc polymorphism is resolved in the `BINARY_ADD` instruction of the Python 3.x interpreter. 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 lookup 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),



(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.x ad-hoc polymorphism.

right branch to `unicode_concatenate`), we have to introduce an auxiliary function that mirrors the operation implementation and acts as a dedicated system default lookup routine for that instruction.

Even though Deutsch and Schiffman [4] report that the "*inline cache is effective about 95% of the time*", we need to account for the remaining 5% that invalidate the cache. Our implementation changes the implementation of the leaf functions to check whether their operands have their expected types. In case we have a cache miss, e.g., we called `long_add` with `float` operands, a call to `PyNumber_Add` will correct that mistake and properly update the inline cache with the new information, i.e., the address of the `float_add` function, along the way.

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

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

Finally, we show how we implement the inline cache pointer in Python 3.1. The interpreter has a conditional instruction format: if an instruction has an argument, i.e., its ordinal number is above some pre-defined threshold, the two consecutive bytes are arguments to that instruction. If the opcode is below that threshold, the next byte contains the next instruction. Hence, two instructions in the array of bytecodes are not necessarily adjacent, which complicates not only instruction decoding, but updating the inline cache pointers, too. Our implementation solves this by encoding the instruction opcode and its argument in one

machine word, and the inline cache pointer in the adjacent machine word. Thus, all instructions have even offsets, while the corresponding inline cache pointers have the subsequent odd offsets (cf. Figure 2).

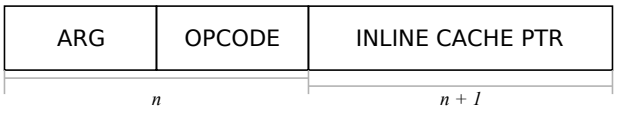


Fig. 2. Changed instruction format.

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—they have to be relocated to the new offsets in order to work properly.

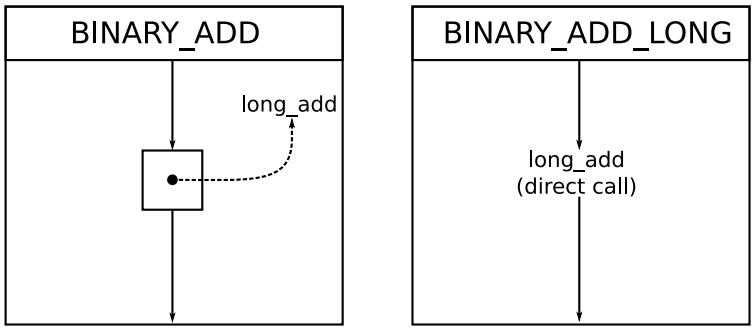
Summing up, this describes a basic and simple, yet more efficient version of an inline caching technique for interpreters without dynamic translation. This basic technique requires significantly more memory space: Instead of one byte for the instruction opcode, this technique requires two machine words per instruction. Therefore, this classic example for trading space for time is not recommended to be applied at all times. Our approach to compensating for this additional memory requirement is to use a simple low-overhead profiling technique to determine which code benefits from inline caching and needs to use this more efficient instruction format.

3 Instruction-Set Extension

Our basic inline caching technique from Section 2 introduces an additional indirect branch per instruction that uses an inline cache. Though this indirect branch is almost always cheaper than calling the system default 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 it is unrealistic to even approach the limit of 2^{16}.

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 3 shows how we can eliminate the inline cache pointer all together by using a specialized instruction that directly calls the `long_add` function.



(a) Basic Inline Caching `long_add`.

(b) New instruction with direct call.

Fig. 3. Instruction-set extension illustrated for operands having `long` type.

Rewriting virtual machine instructions is a well known technique. In the Java virtual machine, this technique is called "quick instructions" [14]. 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 lookup routine. Another significant difference between the well known application in the Java virtual machine and our technique is that whereas the actual quickening in the JVM is used for intialization, i.e., is only used once per affected bytecode, our technique requires instruction rewriting per cache miss, i.e., in the 5% of cases where our "guess" was wrong, we have to rewrite the instruction to match our new information.

Rewriting the bytecodes 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 rewriting approach increases the code size of the interpreter dispatch loop, this may have negative performance impacts on architectures with small instruction caches. For these architectures, the basic technique of Section 2 might perform better because of fewer instruction cache misses. On modern desktop and server hardware, however, the rewriting approach is clearly preferable. 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.

3.1 Inline Caching via Quickening

In Python, each type structure contains a list of function pointers that can be used on arguments of that type. Our specialization technique focuses on three sub-structures within that type structure that capture the context of the 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 a type in list context, e.g. list concatenation, containment, length, repetition (i.e. operation of a list and a scalar), etc.
3. Map context: this context captures the use of 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.

	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	

Table 1. Specialized types by context.

For each of the types in table 1, we can determine whether it implements a 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, e.g., the `nb_add` member points to the implementation of the binary addition for that type. A concrete example is for the integrated long type: `PyLong_Type.tp_as_number->nb_add` points to the `long_add` function, which implements the unbounded range integer addition of Python 3.x. We have a short Python program in a pre-compile step that generates the necessary opcode definitions and operation implementations for several types. Currently, the program generates 77 specialized instructions for several bytecode instructions.

Apart from the generation of dedicated instructions, we need to take care of rewriting 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 does that. This function already updates the inline cache pointer of the current instruction, therefore adding code that changes the instruction opcode of the current instruction is easy. Reusing this function as a means to rewrite instruction opcodes also ensures that we can reuse the cache-miss strategy of the basic technique.

Unfolding the Comparison Instruction: Depending on its operand, Python's `COMPARE_OP` instruction chooses which comparator 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 topmost four 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 [13], 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.

Unfolding the Iteration Instruction: Python has a dedicated instruction for iteration, `FOR_ITER`. It uses a function from the type structure (`tp_iternext`) of the top-of-stack element and calls this function with the top-of-stack element as its argument. This function returns the next value for the iterator variable, which is pushed onto the stack again.

```
TARGET(FOR_ITER)
    /* before: [iter]; after: [iter, iter()] *or* [] */
    v = TOP();
    x = (*v->ob_type->tp_iternext)(v);
    if (x != NULL) {
        PUSH(x);
        DISPATCH();
    }
    if (PyErr_Occurred()) {
        if (!PyErr_ExceptionMatches(PyExc_StopIteration))
            break;
        PyErr_Clear();
    }
    /* iterator ended normally */
    x = v = POP();
    Py_DECREF(v);
    JUMPBY(oparg);
    DISPATCH();
```

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

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

Combination of Variable Caches and Quickening There are several instructions in Python that deal with lookups in environments. For instance, when we examine the `LOAD_GLOBAL` implementation, we find that there is a precedence lookup encoded, i.e., first there is a lookup in the `f->f_globals` member, and if the argument key was not found, a second lookup attempt using the `f->f_builtins` member is tried.

```
TARGET(LOAD_GLOBAL)
    w = GETITEM(names, oparg);
    if (PyUnicode_CheckExact(w)) {
        /* Inline the PyDict_GetItem() calls. */
    }
    /* This is the un-inlined version of the code above */
    x = PyDict_GetItem(f->f_globals, w);
    if (x == NULL) {
        x = PyDict_GetItem(f->f_builtins, w);
        if (x == NULL) {
```

```
            load_global_error:
              format_exc_check_arg(
                      PyExc_NameError,
                      GLOBAL_NAME_ERROR_MSG, w);
              break;
          }
      }
      Py_INCREF(x);
      PUSH(x);
      DISPATCH();
```

Now, dictionary lookup using complex objects is an expensive operation. 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 caching slot of Section 2 and rewrite the instruction to a faster version:

```
TARGET(FAST_LOAD_GLOBAL)
    Py_INCREF(GET_INLINE_CACHE());
    PUSH(GET_INLINE_CACHE());
    DISPATCH();
```

Our current implementation checks whether there occur any `STORE_GLOBAL` instructions in the bytecode, and only then rewrites the instruction. This is a simple way of dealing with this problem and we found no problems in building Python with its standard library and our benchmarks. However, an industrial strength implementation of this technique might require more sophisticated invalidation mechanisms. The same optimization applies to the `LOAD_NAME` instruction.

Unfolding the Call Instruction: 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 be too abstract for efficient interpretation [9]. The same holds true for the Python interpreter: There are only a few bytecodes for calling a function, and the compiler generates `CALL_FUNCTION` instructions most often. Aside from their use for calling host-level functions, the same bytecode is used for calling C-functions. Because Python is a multi-paradigm programming language, the first issue is complicated by the fact that the targets can be either Python functions or methods—the latter being more complicated because of dynamic binding. The second issue—calling C functions—is important because C function targets can have a multitude of arguments, including variable arguments, as well as named arguments. 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).

	Number of Arguments			
Target	0	1	2	3
C std. args	x	x		
C variable args	x	x	x	x
Python direct	x	x	x	
Python method	x	x	x	

Table 2. Specialized `CALL_FUNCTION` instructions.

4 Evaluation

We used several benchmarks from the computer language shootout game [7]. 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. All benchmarks were run on an Intel i7 920, with 2.6 GHz running Linux 2.6.28-15 and gcc version 4.3.3. We used modified version of the `nanobench` program of the computer language shootout game [7] to measure the running times of each benchmark program. The `nanobench` program uses the `getrusage` function to get timings for elapsed user and system time. We add both values and use them as the basis for our benchmarks. In order to account for proper measurement and cache effects, we ran each program 1000 successive times with the Intel Turbo Boost technology turned off. This benchmark was repeated 10 times and our data provides averages over those repetitions.

Figure 4 contains 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.

With the exception of the fannkuch benchmark, the combined approach of using threaded code with inline caching is always faster. From negligible improvements ($< 10\%$) in the case of the binarytrees, fasta, and mandelbrot benchmarks, to a significant speedup ($\geq 10\%$) in the case of the nbody, and spectralnorm benchmarks. In the chameneosredux benchmark, we can see that threaded code execution can result in negative performance, too. Yet, this particular benchmark is inline caching friendly, and therefore a combination of both techniques results in a visible speedup.

We find that the particularly beneficial benchmarks contain only a few function calls in the interpreted programming language. To that end there are several possible reasons for this: a) function calls requires the creation of stack frame objects, as already observed in the BrouHaHa implementation of Smalltalk [15], the creation of equivalent Smalltalk Context objects is expensive and therefore an inline cached function call is less expensive; b) our profiling infrastructure is too expensive, and c) our `CALL_FUNCTION` inline caching schema is restricted to the number of arguments and call types in Table 2, but in those benchmarks the

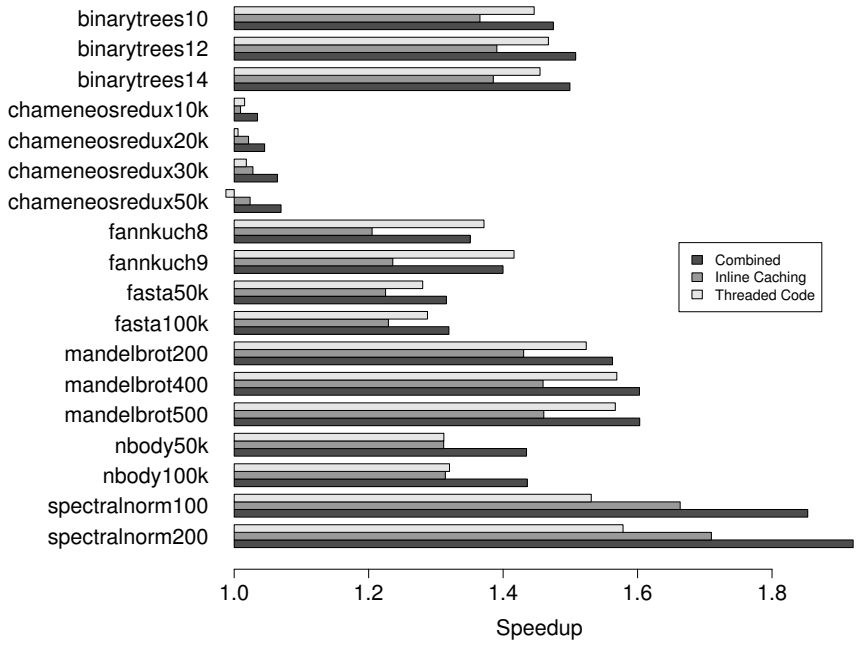


Fig. 4. Achievable speedups on various benchmarks.

expensive calls cannot be optimized and have to resort to the default method lookup. Further investigation is necessary to identify the root cause.

5 Related Work

In his PhD thesis of 1994 [9], 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, it is only created in code we know is *hot*. Therefore, we argue that the increased memory consumption is negligible—particularly since the memory consumption caused by state of the art just-in-time compilers is much more intensive than what our approach requires.

In 2008, Haupt et al. [8] 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 [10]. 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 lookup in the stack frame.

Regarding the use of lookup caches in purely interpretative systems, we refer to an article detailing various concerns of lookup caches, including efficiency of hashing functions, etc., which can be found in "Smalltalk-80: Bits of History, Words of Advice" [13]. 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 [12]. The major difference to our work is that our inline cache does not require the additional lookup and maintenance costs of hash-tables.

Lindholm and Yellin [14] provide details regarding the use of quick instructions in the Java virtual machine. Casey et al. [3] describe details of quickening, superinstructions and replication. The latter technical report provides interesting details 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. We are not aware of any other work in that area. However, our approach to replication is similar to theirs, as is the use of a code generator to compensate for the increased maintenance effort implied by adding new instructions. Our techniques 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 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 architecture will quantify these effects.

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 1.92—only slightly lower than the maximum reported speedup of 2.02 achieved by efficient interpreters using threaded code alone [6].

Acknowledgments

I want to thank Urs Hölzle for details regarding the history of the basic technique of section 2. Furthermore, I am particularly grateful to Jens Knoop and Anton Ertl for valuable discussions and feedback on earlier drafts of this paper.

References

1. Bell, J.R.: Threaded code. Communications of the ACM 16(6), 370–372 (1973), the original reference for threaded code
2. Brunthaler, S.: Virtual-machine abstraction and optimization techniques. In: Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09). pp. 19–30. Elsevier, York, UK (March 2009)
3. 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`
4. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the Smalltalk-80 system. In: Proceedings of the SIGPLAN '84 Symposium on Principles of Programming Languages (POPL '84). pp. 297–302. ACM, New York, NY, USA (1984)
5. Ertl, M.A.: Threaded code variations and optimizations. In: EuroForth. pp. 49–55. TU Wien, Vienna, Austria (2001)
6. Ertl, M.A., Gregg, D.: The structure and performance of efficient interpreters. Journal of Instruction-Level Parallelism 5 (2003)
7. Fulgham, B.: The computer language benchmarks game. `http://shootout.alioth.debian.org/`
8. Haupt, M., Hirschfeld, R., Denker, M.: Type feedback for bytecode interpreters. Position Paper. (ICOOOLPS '07). `http://scg.unibe.ch/archive/papers/Haup07aPIC.pdf` (2008), `http://scg.unibe.ch/archive/papers/Haup07aPIC.pdf`
9. Hölzle, U.: Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. thesis, Stanford University, Stanford, CA, USA (1995)
10. 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 '92). pp. 21–38. Springer-Verlag, London, UK (1991)
11. Hölzle, U., Ungar, D.: Optimizing dynamically-dispatched calls with run-time type feedback. In: Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94). pp. 326–336 (1994)
12. Kiczales, G., Rodriguez, L.: Efficient method dispatch in PCL. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90). pp. 99–105. ACM, New York, NY, USA (1990)
13. Krasner, G. (ed.): Smalltalk-80: bits of history, words of advice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
14. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Boston, MA, USA, first edn. (1997)

15. Miranda, E.: Brouhaha–a portable smalltalk interpreter. In: Proceedings of the SIGPLAN '87 International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87). pp. 354–365. ACM, New York, NY, USA (1987)
16. Vitale, B., Abdelrahman, T.S.: Catenation and specialization for Tcl virtual machine performance. In: IVME '04: Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04). pp. 42–50. ACM, New York, NY, USA (2004), General Chair-Michael Franz and Program Chair-Etienne M. Gagnon