Efficient Interpretation using Quickening

Stefan Brunthaler

Institute of Computer Languages
Vienna University of Technology
Argentinierstrasse 8
1040, Vienna, Austria
brunthaler@complang.tuwien.ac.at

Abstract

Just-in-time compilers offer the biggest achievable payoff performance-wise, but their implementation is a non-trivial, time-consuming task—affecting the interpreter's maintenance for years to come, too. Recent research addresses this issue by providing ways of leveraging existing just-in-time compilation infrastructures.

Though there has been considerable research on improving the efficiency of just-in-time compilers, the area of optimizing interpreters has gotten less attention—as if the implementation of a dynamic translation system was the "ultima ratio" for efficiently interpreting programming languages. We present optimization techniques for improving the efficiency of interpreters *without* requiring just-in-time compilation—thereby maintaining the ease-of-implementation characteristic that brought many people to implementing an interpreter in the first place.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Interpreters, Optimization, Memory Management

General Terms Design, Languages, Performance

Keywords Python, interpreter, quickening, reference counting, instruction format

1. Motivation

The implementation and maintenance of just-in-time compilers requires a lot of resources—probably too much for many projects in their early beginnings, i.e., without financial resources, or popularity/visibility to get enough attention from the open source world. Recent research addresses this huge impact on resources by trying to leverage existing virtual machine infrastructures [YWF09, BCFR09], thus supporting the re-use of existing just-in-time compilers similarly to the front-end/back-end abstraction in traditional compilers. Increasing the efficiency of interpreters without violating their main characteristics, ease of implementation and portability, is an interesting and important problem. The optimization of interpreters is interesting because often simple techniques have a huge impact—for example changing the instruction dispatch from the common switch-based dispatch technique to the more advanced threaded-code[1] [Bel73] dispatch techniques results in reported speedups of up to 2.02 [EG03b]. Furthermore, we are convinced that exploring the design space for efficient interpretation techniques is important, because it provides language implementers with attractive options to optimize their interpreters without having to spend their scarce resources on a dynamic translation sub-system. Without having their resources committed to implementing a just-in-time compiler, they are free to focus on continuing innovation on their programming languages.

In 2003, Ertl and Gregg identified a set of optimization techniques that achieve significant speedup for virtual machines [EG03b]. In addition to the aforementioned threaded-code dispatch optimization, the paper suggests several other optimization techniques, for example using superinstructions [EG03a, EG04]. Most of these virtual machine optimization techniques focus on eliminating the overhead in instruction dispatch, i.e., getting from one bytecode instruction to its successor. These instruction dispatch costs are very high for interpreters where the native machine provides for most of the operation implementation, e.g., by re-using the native machine integer addition instruction to implement the virtual machine integer addition. However, these dispatch costs are disproportionally lower for interpreters with a much higher abstraction level than for these low abstraction-level virtual machines. Therefore, optimiz-

DLS 2010, October 18, 2010, Reno/Tahoe, Nevada, USA.

[1] Threaded-code should not to be confused with multi-threaded code: We use threaded-code to identify the optimized interpreter instruction dispatch technique.

ing away the dispatch costs for interpreters where they do not constitute the primary bottleneck yields proportionally lower speedups. As demonstrated in 2009 [Bru09], the virtual machine abstraction-level has considerable effects on the performance potential of several optimization techniques. Subsequent research on more suitable optimization techniques for high abstraction-level virtual machines indicates that promising optimizations need to cut down the costs implied by operation implementation to achieve significant speedups. Our previous work focuses on enabling efficient inline caching in interpreters without just-in-time compilers [Bru10a, Bru10b], resulting in speedups of up to 1.71.

This paper presents our results of extending this line of research, specifically our contributions are:

- We describe several advanced quickening-based optimization techniques to improve the performance of an interpreter (Section 3). Among others, we describe a technique to cache local variables of the host language in the stack frame of the executing language (Section 3.4).
- We introduce a novel technique to eliminate reference count operations in interpreters (Section 4).
- We present results of our careful and detailed evaluation, demonstrating the effectiveness of our techniques (Section 5). We are able to identify and eliminate up to two thirds of increment, and up to half of the decrement reference count operations (Section 5.4). We report speedups of up to 2.18 when combined with the threaded code dispatch technique (Section 5.5).

2. Background

Many of our target high abstraction-level virtual machines feature dynamic typing. Therefore, we investigated the implementation of efficient inline caching techniques for interpreters without a dynamic translation sub-system [Bru10a, Bru10b]. The primary research vehicle we use to demonstrate our optimizations is the Python 3.1 interpreter. We report our results in the context of the Python 3.1 source code, however, the techniques themselves are general and can be applied to many interpreters with similar characteristics.

By using a code generator to generate instruction derivatives at pre-compile time in combination with quickening at run-time to incorporate the type feedback, we were able to achieve a speedup of up to 1.71 [Bru10b]. Since this technique focuses on optimizing the operation implementation of interpreter instructions, they are orthogonal to optimization techniques focusing on instruction dispatch. Thus, when combined with the threaded-code optimization technique, the maximum speedup we reported is 1.92 [Bru10b].

In order for the interpreter to be able to use the new optimized instruction derivatives, the instruction format needs to be changed. This change removes the limit of 255 possible instructions available to the interpreter. Besides enabling the use of more than 255 instructions, our new regular instruction format allows for more efficient instruction decoding. Instead of using an opcode byte plus optionally two adjacent bytes carrying its argument, we encode the instruction opcode and its argument in one native machine word. To enable further optimization, our previous instruction format interleaves these instruction words with additional native machine words.

The change of the instruction format, requires a separate dispatch loop with different instruction decoding. We use a simple profiling technique to decide when to use this optimized interpreter routine [Bru10b]. This profiling technique involves counting the invocations of each function. Once the invocation count reaches a pre-definable threshold, we start using the optimized interpreter dispatch routine. At the beginning of the first optimized execution, we allocate memory for the new instruction format and re-encode the sequence of bytecode instructions. We use quickening to rewrite instructions from most-generic implementations to their optimized derivatives. This describes our starting point for the additional optimizations we describe in this paper. Though most of these techniques can be applied without this basis, our evaluation builds on the improvements upon this earlier work.

3. Advanced Quickening-based Optimizations

Section 2 briefly explains the foundation of our previous work [Bru10b], whereupon we will subsequently add several new optimization techniques in the remaining part of the paper. First, we provide details for adding inline caching to the comparison instruction of the Python interpreter (`COMPARE_OP`); which our old interpreter did not do and therefore provides an important link to our previous work [Bru10b]. Second, we present a simple quickening-based technique to unfold code (Section 3.2). Next, we introduce a new instruction format together with two new optimizations using it (Section 3.3). Finally, we describe a new technique to cache local variables of the host language in the stack frame of the executing language (Section 3.4).

3.1 Inline Caching the Comparison Instruction

The regular comparison instruction relies on its argument to decide which comparison operation to execute. Because of its rich set of types, and possible ad-hoc polymorphism using the comparison instruction, most of the invocations end up invoking a type-dependent comparison function. Figure 1 shows the changes necessary for constructing an inline cached version of the comparison instruction: the implementation on the right side contains the changes necessary to create an optimized derivative of the standard `COMPARE_OP` instruction. The optimized derivative provides an inline cache for Python's integer type, `PyLong_Type`. The lines marked with > represent unchanged copies from the stan-

```
TARGET(COMPARE_OP)
  w = POP();
  v = TOP();

  x = cmp_outcome(
      oparg, v, w);

  Py_DECREF(v);
  Py_DECREF(w);

  SET_TOP(x);
  if (x == NULL) break;
  DISPATCH();
```

```
TARGET(INCA_CMP_LONG)
>
>
/* check for misses */
if (v->ob_type
    != w->ob_type)
    goto COMPARE_OP_MISS;
if (v->ob_type
    != &PyLong_Type)
    goto COMPARE_OP_MISS;

/* inline cached call */
x= PyLong_Type
   .tp_richcompare(
   v, w, oparg);
>
>
>
>
>
```

Figure 1. Implementation of COMPARE_OP on the left, an optimized derivative on the right.

dard operation implementation on the left side. As we have briefly described in Section 2, the function resolving the dynamic types (do_richcompare) quickens the instruction from the generic COMPARE_OP instruction to the optimized INCA_CMP_LONG instruction.

At this point we introduce some internals of the Python interpreter. The source code example for the standard COMPARE_OP contains several macros which we explain here for the convenience of the reader:

- TARGET: This macro is an auxiliary macro for helping to weave in the necessary instruction decoding when using threaded-code dispatch. It defines a label, that can be used as a jump target on compilers that support the label as value feature (TARGET_COMPARE_OP).
- POP/TOP, PUSH/SET_TOP: These macros manipulate the operand stack. The difference between the POP/PUSH and TOP/SET_TOP macros is that the latter do not change the stack pointer.
- Py_DECREF: Decreases the reference count of its argument, if the reference count drops to zero, this macro invokes the reclamation procedure. Incrementing the reference count is done via Py_INCREF.
- DISPATCH, FAST_DISPATCH: These macros take care of jumping to the next instruction when threaded-code dispatch is enabled.

3.2 Unfolding with Quickening

Several instructions within the Python 3.1 interpreter have different behavior depending on their argument. While it is convenient to encode multiple behaviors into just one instruction, it is sub-optimal with respect to the performance of this instruction. If we take a look at the following BUILD_TUPLE instruction, we see that the loop-body depends on the instruction argument oparg:

```
TARGET(BUILD_TUPLE)
    x = PyTuple_New(oparg);
    if (x != NULL) {
        for (; --oparg >= 0;) {
            w = POP();
            PyTuple_SET_ITEM(x, oparg, w);
        }
        PUSH(x);
        DISPATCH();
    }
    break;
```

Results of a dynamic bytecode frequency analysis on our benchmark programs as well as some other Python programs find that the argument is most often either 2 or 3. Therefore we construct optimized BUILD_TUPLE instructions with their argument fixed, for example with opcode := 3, the implementation can be optimized like this:

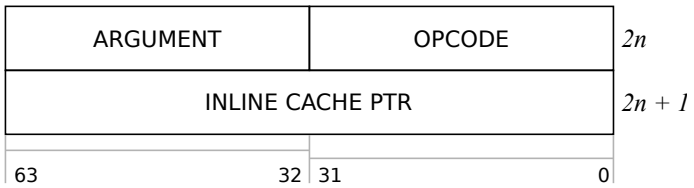
```
TARGET(BUILD_TUPLE_THREE)
    x= PyTuple_New( 3 );
    if (x != NULL) {
        PyTuple_SET_ITEM(x, 2, TOP());
        PyTuple_SET_ITEM(x, 1, SECOND());
        PyTuple_SET_ITEM(x, 0, THIRD());
        STACKADJ(-2);
        SET_TOP(x);
        DISPATCH();
    }
    break;
```

This manual unrolling of interpreter instructions enables the compiler to perform more optimizations on this block. Further examples for unfolding with quickening are not limited to loops, but rather to straightening cascaded conditional blocks that solely depend on the instruction argument. This corresponds directly to an optimization for Smalltalk 80 interpreters, suggested by Allen Wirfs-Brock in 1982 [WB82]. We apply this optimization to several instructions: BUILD_TUPLE, BUILD_LIST, UNFOLD_SEQUENCE.

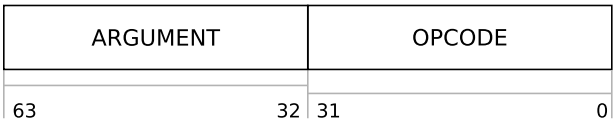
3.3 Reduced Instruction Format

Our instruction format of [Bru10b] contains interleaved words for storing pointers (cf. Figure 2(a)). We use these pointers to either cache addresses of functions or data objects. Though we found sufficient uses for justifying these interleaved words, some were always being empty, i.e., NULL,

because there was just no use for them in the instruction implementation. Some of the optimizations we use this pointer cache for, was to significantly speed up the implementation of the LOAD_GLOBAL instruction [Bru10b].



(a) Old instruction format requiring two words [Bru10b].



(b) New instruction format using just one word.

Figure 2. Changing the instruction format.

We propose a new instruction format that removes the interleaved inline cache pointer words, such that the instruction words are adjacent (cf. Figure 2(b)). Thus we require only half as much native machine words as before (cf. Figure 2). However, since we do not want to interpret without our optimizations, we describe alternative implementation approaches.

Inlining Data Object References

We evenly divide the native machine word into two sections, one for the instruction opcode, and another for the corresponding instruction argument. So, for 64 bit systems, there are 32 bits for each segment. Many LOAD_CONST instruction occurrences load objects that have been allocated in the lower memory area below 32 bits. Consequently, for all constant objects where this observation holds, we can actually replace the array indirection by a direct push of the instruction argument. Figure 3 shows how we implement the optimized derivative (right side) and which lines remain unchanged from the standard implementation (left side). Figure 4 shows the effect of inlining a small object pointer into an occurrence of the INCA_LOAD_CONST instruction.

```
TARGET(LOAD_CONST)          TARGET(INCA_LOAD_CONST)
    x = GETITEM(              x= (PyObject*) oparg;
        consts, oparg);

    Py_INCREF(x);           >
    PUSH(x);                >
    FAST_DISPATCH();        >
```

Figure 3. Implementation of LOAD_CONST and its optimized derivative.

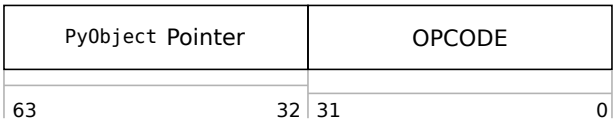


Figure 4. Inlining a PyObject pointer.

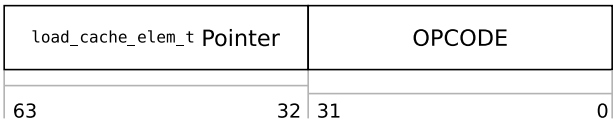


Figure 5. Inlining a load_cache_elem_t pointer.

A similar optimization applies to the LOAD_GLOBAL instruction. However, things are not quite as easy as in the LOAD_CONST case: Since there exists a corresponding destructive update instruction (STORE_GLOBAL), any execution of that instruction can invalidate the inline cached instruction argument. Therefore, we have to use another level of indirection involving a dedicated cache. First, we allocate an array of n elements of the following data structure in the lower memory area, below 32 bits.

```
typedef struct {
    bytecode_t *ip;
    PyObject *elem;
} load_cache_elem_t;
```

The ip field in the load_cache_elem_t record holds the corresponding instruction pointer of the LOAD_GLOBAL instruction for which the elem field holds the cached data object pointer. By construction we ensure that this element fits into the 32 bits length restriction of the instruction argument section of the instruction word. Therefore, we can replace the integer argument by the address of one of the n load_cache_elem_t elements (cf. Figure 5). The following optimized instruction derivative can be used instead of the much more complicated LOAD_GLOBAL implementation:

```
TARGET(INCA_LOAD_GLOBAL)
  if (PyLoadCache_IsValid(oparg, INSTR_PTR()) {
      x= PyLoadCache_GetElem(oparg);
      Py_INCREF(x);
      PUSH(x);
      FAST_DISPATCH();
  }
  oparg= get_oparg(codeobject, INSTR_PTR());
  goto TARGET_LOAD_GLOBAL_SKIP_DECODE;
```

PyLoadCache_IsValid and PyLoadCache_GetElem are auxiliary macros hiding the details of the load_cache_elem_t definition. Additionally, we see that we need to recover the original instruction argument from the old list of bytecode instructions in case we have a cache miss. Whenever we quicken an instruction from LOAD_GLOBAL to INCA_LOAD_GLOBAL, we need to use an element from the array of n elements. A simple and effective strategy is to use this list as a ring

buffer, where we maintain a pointer to the next free element and reset it to zero whenever it points to $n + 1$.

As with any other cache, we have to take care of properly invalidating it to maintain correctness. Whenever we execute a `STORE_GLOBAL` instruction, our implementation clears the `ip` field of all n `load_cache_elem_t` records. This causes cache misses in all sub-sequent `INCA_LOAD_GLOBAL` instructions. These cache misses lead to invocation of the `LOAD_GLOBAL` instruction, which updates the cache and requickens the instruction back to `INCA_LOAD_GLOBAL`.

3.4 Partial Stack Frame Caching of Local Variables

In any stack-based interpreter, load instructions are usually among the most frequently executed instructions. In Python, two kinds of load instructions are usually among the most frequent ones: a) the `LOAD_CONST` instruction for loading constants, and b) the `LOAD_FAST` instruction for loading local variables. We already took care of optimizing the first case in Section 3.3. Now, we turn our attention to the `LOAD_FAST` instruction:

```
TARGET(LOAD_FAST)
    x = fastlocals[oparg];
    if (x != NULL) {
        Py_INCREF(x);
        PUSH(x);
        FAST_DISPATCH();
    }
    /* omitted rest of implementation */
```

`Fastlocals` is a pointer to a field inside the Python stack frame, where enough memory has been allocated to hold references for all local variables. Every `LOAD_FAST` instruction uses an array indirection, and, while not expensive by itself, this accrues considerable time during the execution of any program, because the interpreter executes these load instructions very frequently. On the other hand, since these instructions execute that often (early measurements indicate that about a third of all instructions are loads), even small optimizations will pay off quickly. We propose to partially cache variables from the Python stack frame in the interpreter's C stack frame. The optimization technique requires the following steps:

1. Declare additional variables in the C stack frame of the interpreter.
2. Generate instruction derivatives that use the additional local variables.
3. Promote contents of the local variables of the Python stack frame to the local variables of the C stack frame *before* entering the interpreter main loop.
4. Write back contents of the local variables *after* leaving the main loop.

Our implementation of such an optimized derivative is:

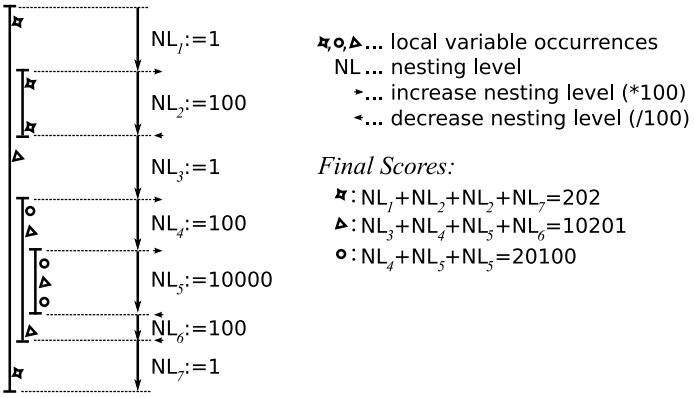


Figure 6. Calculating scores in the presence of multiple loops.

```
TARGET(LOAD_FAST_A)
    Py_INCREF(fast_slot_a);
    PUSH(fast_slot_a);
    FAST_DISPATCH();
```

As we can see, we create a new variable `fast_slot_a`, which holds the contents of some variable of the stack frame. This directly points us to the next technicality: Which local Python variables do we cache with `fast_slot_a`? A very naive strategy would be to just allocate the first n of total m local variables of the Python stack frame to our additional n caching variables. This only makes sense when the number of total variables m is actually lower than or equal to the number of available additional caching variables n. In cases where the number of local variables of the Python stack frame exceeds the number of available caching variables, i.e., $n < m$, we can surely do better.

An optimal solution to this problem is when we allocate the most frequently executed `LOAD_FAST` instructions to the n available additional local caching variables. Therefore, we want to estimate which `LOAD_FAST` and `STORE_FAST` instructions will be executed most often. One heuristic is to just rank `LOAD_FAST` and `STORE_FAST` instructions according to their number of occurrences within a given sequence of instructions. However, we can improve this heuristic significantly: In the presence of loops, it is reasonable to assume that the load and store instructions within the loop-body will be executed more often. This assumption is recursive, i.e., the deeper the loops are nested, the more often we assume the load and store instructions in their bodies to be executed. We use this heuristic to select the candidates among the local variables of the Python stack frame. In addition to being reasonable, this heuristic can be efficiently implemented: In only one linear pass we can calculate a score for each local variable. Whenever we find an occurrence of a local variable, we increase its score by the current nesting level (cf. Figure 6). To give occurrences inside loops higher scores, we multiply the nesting level by 100 whenever we enter a loop,

and divide it by 100 whenever we leave a loop. Thus, an occurrence within a loop equals 100 occurrences outside of that loop. We arrived at this value by experimentation, for example using a nesting level weight of just 10 does not lead to optimal local variable selection in some of our benchmarks. Among all scores we select the n highest and rewrite their occurrences to the optimized instruction derivatives.

4. Reference Counting meets Quickening

Section 3 deals with optimizing various load instructions, primarily using a combination of caching and quickening. Unfortunately, the operation implementation of Python 3.1 instructions is still sub-optimal with respect to maximum efficiency. We find additional overheads in two respects: a) all operands are objects that require (un-)boxing, and b) reference counting. Among those two, we choose to optimize reference counting.

Measurements on the overheads accrued by reference counting during execution of Smalltalk programs by Ungar and Patterson [UP82] indicate that a reduction of reference count operations is highly beneficial for a stack-based interpreter. Further measurements of the Berkeley Smalltalk-80 system using deferred reference counting as suggested by Deutsch and Bobrow in 1976 [DB76] indicate that using this optimization removes up to 90% of reference count operations [Bad82]. Based on these promising prior results, we expect that focusing on optimizations of reference counting will be beneficial.

The original problem with reference counting, viz., the massive amount of reference count operations accrued by the local operand stack modifications presents a bottleneck. First, we investigate what the nature of these operand stack modifications is: whenever a stack-based interpreter executes an n-ary operation, it expects its operands to be the top-most n entries on the operand stack. These operands are either the results of previous computations or pushed onto the operand stack by a corresponding load operation.

Instr. Pos.	Instruction	Operand Stack
1	LOAD_FAST	x_1
2	LOAD_FAST	x_1 x_2
3	BINARY_MULT	x_3
4	LOAD_FAST	x_3 x_4
5	BINARY_MULT	x_5

Figure 7. Sequence of instructions and effect on the operand stack.

Figure 7 shows a sequence of bytecode instructions and their effect on the operand stack, i.e., after executing the LOAD_FAST instruction at position 1, the operand stack contains its result—x_1—as the top-of-stack element.

```
TARGET(LOAD_FAST)
 x = GETLOCAL(oparg);
 Py_INCREF(x); /* A */
 PUSH(x);
 FAST_DISPATCH();
```

```
TARGET(BINARY_MULTIPLY)
 w= POP();
 v= TOP();
 x= PyNumber_Multiply(
    v, w);
 Py_DECREF(v); /* B */
 Py_DECREF(w); /* B */
 SET_TOP(x);
 if (x != NULL)
     DISPATCH();
 break;
```

Figure 8. Implementation of LOAD_FAST instruction on the left and of BINARY_MULTIPLY the right.

Figure 8 shows the corresponding implementations of our sequence of instructions. The mark A in the left implementation of LOAD_FAST in Figure 8 shows the Py_INCREF operation for the object x. The mark B in the right implementation of BINARY_MULTIPLY in Figure 8 shows the corresponding Py_DECREF operations for the operands v and w. In our sequence of bytecode instructions, however, the first BINARY_MULTIPLY—at position 3—is directly executed after two LOAD_FAST instructions—at positions 1, and 2 respectively. Therefore, the two increment reference count operations of the first and second LOAD_FAST result object—x_1, and x_2 respectively—are immediately decremented in the first BINARY_MULTIPLY occurrence. Exactly this arrangement of redundant increment-decrement reference count operations constitutes a big part of the overhead in local operand stack modifications mentioned by Deutsch and Bobrow in 1976 [DB76].

Next, we present a way to optimize away these conservative reference count operations, which consequently allows us to significantly improve the efficiency of our interpreter. Our solution consists of two steps:

1. We eliminate the reference count operations from the operation implementation,
2. We identify sequences of instructions where reference count operations are too conservative, and can, therefore, be safely optimized away.

Using systematic pre-generation of optimized derivatives allows us to remove the redundant reference count operations from operation implementation. Until now, we have only described the case where all reference count operations in a sequence are *explicit*. But, in fact, only one of the two operands for the second occurrence of BINARY_MULTIPLY is loaded explicitly by the third LOAD_FAST instruction—x_4. The other operand is the result of executing the first BINARY_MULTIPLY instruction—x_3. Examining the implementation of BINARY_MULTIPLY again (cf. Figure 8, right side), we notice that there is no explicit increment reference count operation for the result object x_3. Since there is an

explicit decrement reference count operation in the second BINARY_MULTIPLY instruction—at position 5, this means that there has to be an *implicit* increment reference count operation hidden within PyNumber_Multiply. If we want to eliminate such implicit reference count operations, we have to provide our own version of PyNumber_Multiply, which eliminates these hidden increment reference count operations. While this is certainly possible, and would allow us to eliminate such implicit reference count operations, too, we avoid doing so for illustrative purposes—our basic approach can be adapted to this scenario as well. Instead, we focus on the removal of explicit reference count operations only, which necessitates that we account for reference count operations being either explicit or implicit. Next, we eliminate redundant reference count operations from a sequence by quickening its instructions from their most generic implementations to their optimized derivatives.

```
/* data type definitions */
typedef struct {
  signed char imp; /* implicit rc ops */
  signed char exp; /* explicit rc ops */
  /* value > 0: no of increment rc ops */
  /* value < 0: no of decrement rc ops */
} tuple_t;

typedef struct {
  bytecode_t *instrPtr;
  tuple_t effect;
} effect_stack_elem_t;

/* local variables */
bytecode_t *instrPtr= codeobject->co_opt_code;
bytecode_t *cur= instrPtr;
effect_stack_elem_t *stackPtr= stack;
tuple_t effect;
opcode_t opcode;
int i= 0, n= 0, size= Py_SIZE(codeobject->co_code);

/* simple abstract interpreter */
while (i < size) {
  cur= instrPtr;
  opcode= decodeInstr();
  i++;

  if (basicBlockBorder(opcode))
    clearStack();

  if (rotateInstr(opcode)) {
    rotateStackElems(stackPtr, opcode);
    continue;
  }

  if (refcountEffect(opcode, &effect)) {
    if (effect.exp < 0) {
      n= -effect.exp;
      stackPtr-= n;

      /* additional quickening, e.g., function calls */
      if (n <= 2 && isMarkable(opcode)) {
        if (n == 2)
          quickenBinaryOp(&cur, &stackPtr);
        else if (n == 1)
          quickenUnaryOp(&cur, &stackPtr);
      }
    }
    pushOpnds(&cur, &stackPtr, &effect);
  }
}
```

Figure 9. Implementation of our simple abstract interpreter.

To find these redundant reference count operations, it is necessary that we simulate the operand stack of the actual interpreter by using a simplified abstract interpreter. Similar to Xavier Leroy's description of the first step in Java bytecode verification [Ler03], we use an abstract interpreter over the reference count operations occurring in operation implementations. We do this in a linear pass over a sequence of instructions. Since there is no data-flow analysis involved, our optimization is restricted to basic block boundaries. Whenever our simple abstract interpreter finds matching pairs of redundant reference count operations, it uses quickening to rewrite the occurrences of these instructions to their more aggressive derivative. Going back to our previous example, our simple abstract interpreter would be able to detect that all reference count operations are conservative, *except* the implicit reference count operation from the first BINARY_MULTIPLY instruction to the second BINARY_MULTIPLY. Figure 9 contains an illustrative implementation of this simple abstract interpreter. Therefore, our systematic pre-generation of optimized derivatives needs to account for selective generation of reference count operation optimized derivatives, e.g., for binary operations the following four optimized derivatives are possible:

- both reference count operations can be removed,
- the top-of-stack operand's reference count operations can be removed,
- the reference count operations of the second element on the operand stack can be removed,
- no reference count operations can be removed (this corresponds to the standard implementation).

The creation of reference count operation optimized derivatives leads to increased requirements for the instruction cache, because the dispatch loop size increases. While this holds also true for our approach of [Bru10b], it is much more expensive when creating the reference count operation optimized derivatives: for every case in our previous enumeration, we require a new derivative. This invariably leads to instruction cache misses and we need to ensure that the implied cache penalties do not over-compensate for our gains in efficiency. So, we perform an analysis to make informed decisions. This analysis uses an instrumented interpreter that prints the current instruction in addition to the results we obtain by running our simple abstract interpreter. We use a separate program to accumulate the results we obtain when running our benchmarks on this instrumented interpreter. Thus we can quantify the dynamic instruction frequency as well as the number of optimization scenarios for reference count quickening. According to our results, the first two of our four cases occur most often, therefore, we create derivatives for almost all operations. Exceptions are infrequently occurring instructions, such as Python's floor divide instruction, and the logical operator instructions. Only for the most frequent instructions, we provide the optimized derivatives

for our third case, i.e., the elimination of reference count operations for the second operand on the stack. Our analysis identifies the BINARY_ADD and BINARY_MULTIPLY instructions as candidates here—we omit the generation of optimized derivatives for all other instructions in that scenario.

So far, we only described the situation for interpreter instructions for host-level language operations. In addition to those operations, however, a typical Python program relies on numerous function calls. Therefore, it is only natural to apply this optimization to the call instruction of the Python interpreter, too. Using our dynamic bytecode frequency analysis together with our simple abstract interpreter over the amount of reference count operations in an instruction, we find that the following cases occur most often:

- all reference count operations necessary to load the arguments of a function/method call can be safely eliminated,
- all but the top-most reference count operation can be safely eliminated.

We can easily add the quickening code for both of the function/method call optimizations to our simple abstract interpreter: Since we properly simulate the stack behavior, we just iterate over all arguments for any given occurrence of a CALL_FUNCTION instruction and check that all operands have explicit Py_INCREF operations. If this holds, we quicken the instruction and all of its operands to their optimized derivatives. If it does not hold, we check that for a function call of n arguments, $n-1$ have explicit Py_INCREF operations, and only the top-most operand has an implicit Py_INCREF—this indicates quickening potential for our second case.

5. Evaluation

5.1 Interpreter Configuration

Since all of our new techniques can be applied together, we briefly explain how we do this. First, we allocate memory and re-encode the word-code as previously described in Section 2—with the notable exception of using our changed instruction format as described in Section 3.3. Because our new instruction format changes the instruction position, we have to relocate the jump offsets subsequently. Next, we use our abstract interpreter over the number of reference count operations per instruction over the list of bytecodes to find sequences with redundant reference count operations. Once we find a such a sequence, we quicken its instructions to their more aggressive derivatives. We do this before actually executing any of the instructions, hence, a compiler could already emit optimized instructions. Finally, we start interpreting and quicken instructions to their optimized derivatives based on acquired type feedback. The reason we do reference count operation quickening before inline caching is that the reference count operation optimized instruction set is smaller than the type-dependent instruction set for inline caching via quickening, i.e., it is merely a matter of practicality.

The following two sections of this paper describe techniques in general using variables without giving concrete values. We use the following configuration:

- Section 3.3: Our load cache in the lower memory area uses 128 elements.
- Section 3.4: We promote 4 local variables to their dedicated slots.

All in all, our optimized interpreter has 395 instructions.

5.2 Systems

We use several benchmarks from the computer language benchmark game [Ful]. We would like to give results of popular real-world Python applications, such as Zope, Django, and twisted. Unfortunately, however, the adoption of Python 3.x in the community is rather slow, and there are currently[2] no ports of these applications available. We ran the 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 [Int08].) Instruction cache size is 32 KB.
- IBM PowerPC 970 with 2.0 GHz, running Linux 2.6.18-4 and gcc version 4.1.2. Instruction cache size is 64 KB.

We used a modified version of the nanobench program of the computer language benchmark game [Ful] to measure the running times of each benchmark program. The nanobench program uses the UNIX getrusage system call to collect usage data, for example 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, cache effects, and unstable timing results for benchmarks with only little running time, we ran each program 100 successive times and use arithmetic averages over these repetitions for our evaluation.

5.3 Code Generator Statistics

We measured the lines of code using the sloccount program of David Wheeler [Whe10]. Our code generator produces 6178 lines of C code that is to be included in the main interpreter. The Python code of our generator amounts to 2739 lines of code, however, out of those 2739 lines of code, 1700 lines of code are consumed by our type-data file generated by raw-data from gdb. Therefore, the actual amount of Python code without the master data needed to generate the C code is 1039 lines of code. In addition to the generator and its product, we have manually coded 1759

[2] As of May 2010.

lines of code. These are 400 lines of code for quickening the `CALL_FUNCTION` instruction and supplying our own version of `unicode_concatenate`, 347 lines of code for our simple abstract interpreter to rewrite the reference count operations, 272 lines of code for the creating and manipulating the new instruction format (including the scoring heuristic), and 87 lines of code that implements the load cache that we described in Section 3.3. The remaining lines of code are mostly externalized interpreter macros from the original dispatch loop and smaller auxiliary files.

Using `‘cat * | wc -l‘` to calculate the number of C-code markup inside the Mako templates adds another 1385 lines of code. Here, we cannot use the `sloccount` program, since it does not understand the Mako template language.

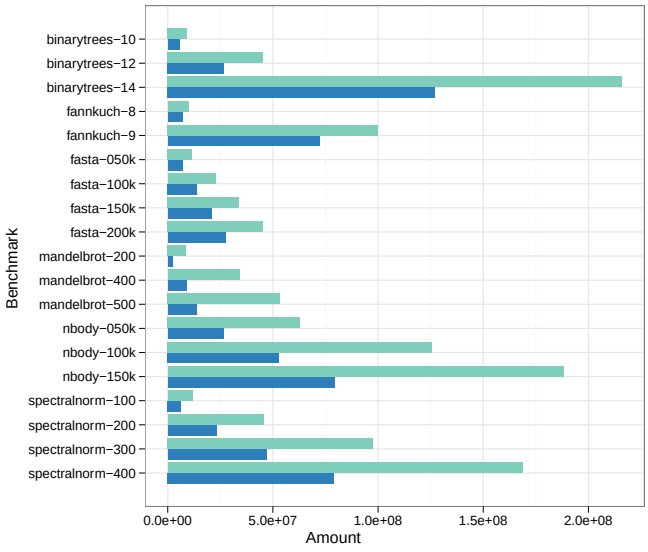
5.4 Reference Count Operations Eliminated

Figure 10 illustrates the effect of eliminating *explicit* reference count operations, broken down by increment (cf. Figure 10(a)) and decrement (cf. Figure 10(b)) operations. We notice that for all benchmarks, the number of decrement operations exceed the number of increment operations. This is due to the fact that we only counted occurrences of *explicit* reference count operations, i.e., our figures do not include the implicit increment reference count operations, such as those occurring when we allocate new objects, or within operation implementations that do not use these macros to update the reference count for an object.

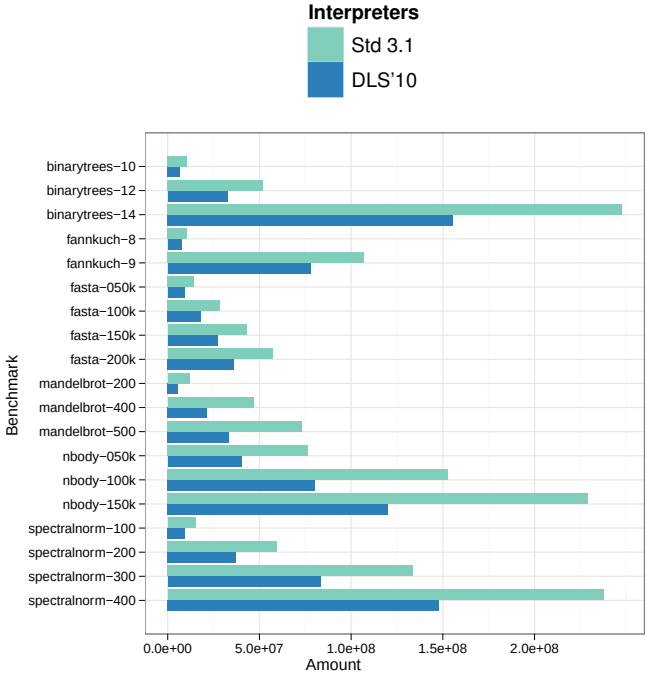
Regarding the actual results, we notice that for some benchmarks, such as `mandelbrot` and `spectralnorm`, our technique finds and eliminates substantial amounts of redundant reference count operations. The `mandelbrot` benchmark benefits particularly from this optimization: we eliminate more than two thirds of all increment reference count operations, and about half of all decrement reference count operations. Our technique achieves good elimination rates on all other benchmarks—with the `fasta` benchmark being the only exception: while we are able to reduce both, increment and decrement reference count operations, the reduction rates are not nearly as impressive as with the other benchmarks. This is due to the benchmark containing several occurrences of the `BUILD_SLICE` instruction, for which we do not provide reference count operation optimized derivatives. As we will see in the next section, this has implications on the performance, too.

5.5 Benchmarks

We present the results running our benchmarks in Figure 11. The actual speedups are arithmetic averages over all repetitions for one benchmark, and we normalize by the standard Python 3.1 interpreter *without* the threaded-code dispatch optimization. Because of our previous experience, we are not surprised to find the `spectralnorm` benchmark to be particularly amenable to our optimizations (cf. Figure 11(a)). Surprisingly, however, we find that the `mandelbrot` benchmark benefits most from our new optimizations—this holds



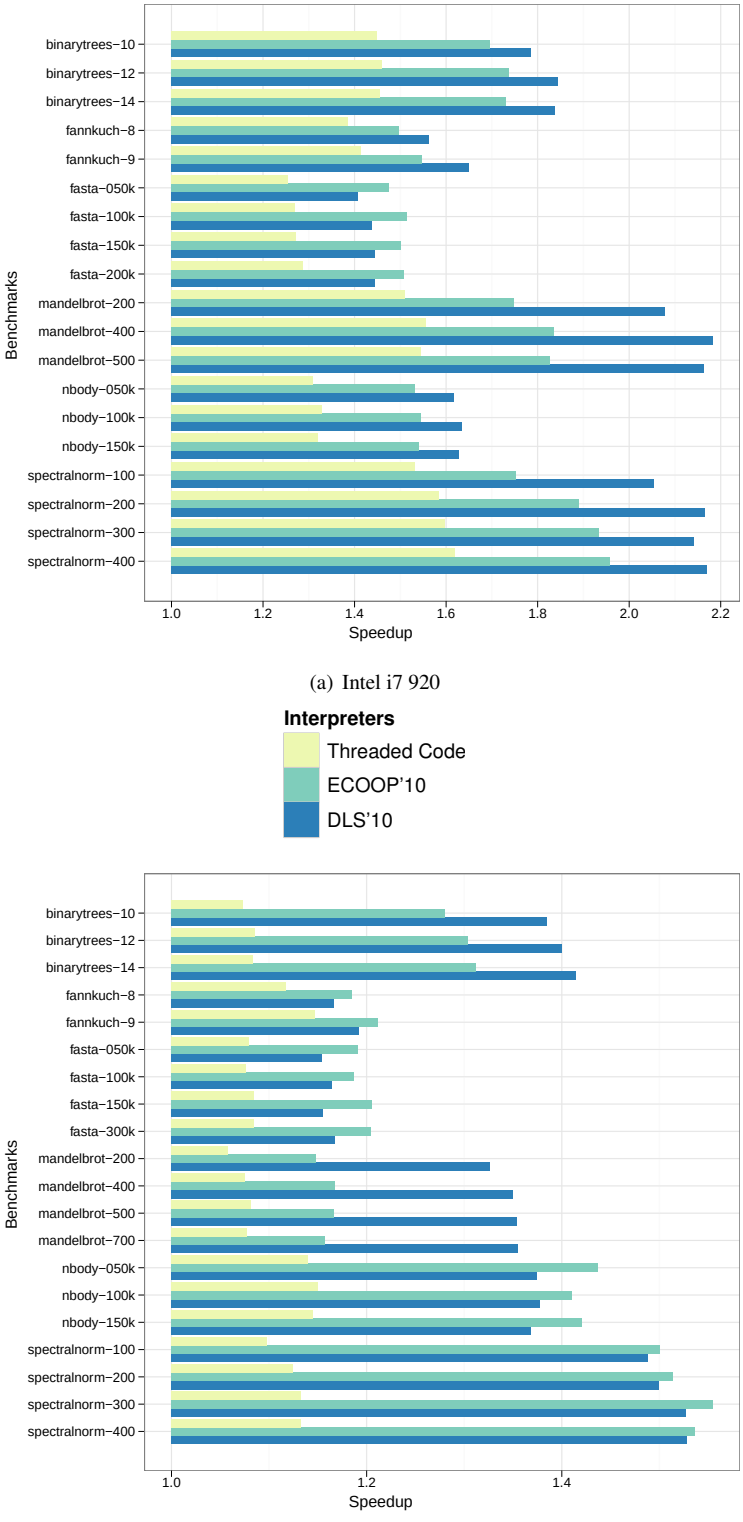
(a) Number of increment reference count operations.



(b) Number of decrement reference count operations.

Figure 10. Reference count operations occurring per benchmark.

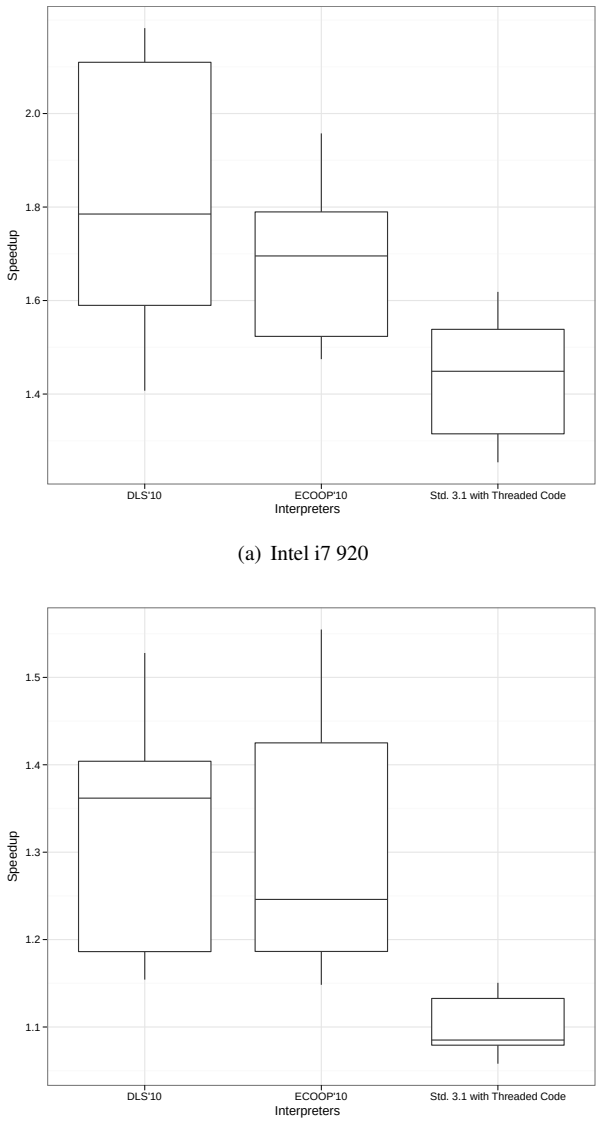
true on both architectures (cf. Figure 11(a), and Figure 11(b) respectively). In addition to the `mandelbrot` benchmark, the `binarytrees` benchmark performs noticeably better using our new optimizations, too. While our optimizations fare particularly well on the Intel i7-920 Nehalem architecture, we note that on the PowerPC 970, for the `nbody` and `spectralnorm` benchmarks, our new optimizations actually reduce the maximum possible speedup by a small amount.



(a) Intel i7 920

(b) PowerPC 970

Figure 11. Benchmark speedups relative to standard Python 3.1 interpreter.



(a) Intel i7 920

(b) PowerPC 970

Figure 12. Overall speedup relative to standard Python 3.1 interpreter.

First, we assumed that these results are due to differences in instruction cache size. It turns out, however, that the PowerPC 970 has double the amount of resources here, i.e., 64 KB instruction cache size for the PowerPC 970 as compared to the 32 KB of the Intel i7. Thus, we need further research to investigate what is actually causing the slowdowns. As noted in Section 5.4, our technique does not cover some important parts of the `fasta` benchmark. In consequence, our technique is not able to deliver its full potential here.

To further assess the benefits of our optimizations, we compare all possible speedups using standard box plots in Figure 12. The upper, middle, and bottom line of the box correspond to the 75^{th}, 50^{th}, and 25^{th} percentile respectively. The ends of the whiskers represent the maximum and minimum measured results.

Interestingly, we see that the optimized dispatch using the threaded-code dispatch technique performs considerably

worse on the PowerPC architecture: the median speedup in our benchmarks is below the factor of 1.1, compared to a median speedup factor of more than 1.4 on the Intel i7-920. Using the techniques presented in this paper, we are able to significantly improve the performance of the Python interpreter on Intel architectures (cf. Figure 12(a)): While the median speedup improves from about 1.7 to about 1.8, the maximum possible speedup improvement is more impressive: from a previously possible speedup of about 1.9, we now reach a maximum speedup of over 2.1. Our performance results for the PowerPC are not directly comparable. In comparison to our inline-caching-only work [Bru10b], we already noted that using the new optimization techniques actually reduces the maximum possible speedup on some benchmarks. That notwithstanding, however, applying the new techniques visibly improves the median speedup. Consequently, we argue in favor of applying our techniques despite possibly lower maximum speedups.

Unfortunately, all of our benchmarks measure the impact of combining all optimizations. Yet, we are interested in how each optimization performs on its own, in order to allow implementers to choose among them. Therefore, we did preliminary benchmarks on our first system, i.e., the Intel i7 920, with our reference counting optimization of Section 4 turned off, resulting in an interpreter with just 214 instructions. While the interpreter with all optimizations of Section 3 achieves a higher speedup of almost 2.31 on the `spectralnorm` benchmark, its overall performance is significantly below the interpreter that removes redundant reference count operations. On three benchmarks (`fannkuch`, `fasta`, `nbody`) the interpreter without reference count quickening delivers only negligible performance improvements over our previous interpreter [Bru10b], whereas enabling reference count quickening adds up to 7% on top of those results. For the remaining benchmarks, the reduced interpreter adds 5% (`binarytrees`) and 16% (`mandelbrot`) performance over our previous interpreter, enabling reference count quickening adds another 5% on top of those results.

6. Related Work

The inline caching of the comparison instruction extends our work on inline caching for high abstraction-level interpreters. In 2010, Williams, McCandless, and Gregg from the Trinity College Dublin presented similar work for optimizing dynamic typing of the Lua interpreter [WMG10]. The biggest difference between both our approaches is that we pre-generate optimized instruction derivatives, while their work uses a background thread that generates optimized instructions on a by-need basis using a separate compiler. This compiled representation of a typed Lua instruction is then linked into the running interpreter for future use. Their approach has an advantage over ours: while our pre-generated derivatives target only optimizations of types and functions of the standard library, their by-need compilation approach is able to optimize code regardless of provenience.

Using quickening to unfold loop bodies is a modern interpretation of a technique mentioned in Allen Wirfs-Brock's excellent article on the design decisions of Smalltalk interpreters [WB82] in "Smalltalk 80: Bits of History, Words of Advice" [Kra84]. When we generalize the notion that his technique eliminates the necessity of instruction argument decoding, we can basically subsume all of our optimizations by noting that we increase the amount of information that is attached to an interpreter instruction jump. For example, after our technique has finished incorporating type feedback, every instruction jump from one instruction to another one not only indicates which operation we want to call, but implicitly contains information for the type we expect and the amount of reference count operations we need.

There exist several optimizations that rely on changing the instruction format. In 1982, Pemberton and Daniels [PD82] describe how the Pascal P4 system uses the 60 bit native machine words of a CDC 6000 series computer to accommodate two 30 bit interpreter instructions with operands. As a recent example, we refer to the instruction formats available in Google's Android Dalvik VM [dal07]. The Dalvik virtual machines defines multiple instruction formats for optimized interpretation. Particularly close to our work is the instruction set of the Inferno virtual machine, called Dis [WP97]. Dis' instruction set provides three-operand memory-to-memory operations, where every instruction can directly operate on the memory addresses referred to by its operands. This is reminiscent of register based virtual machine architecture [SCEG08], with the notable exception of not using registers. In comparison with our work presented in Section 3.3, however, we note that our technique does not change the machine architecture of the Python interpreter, i.e., it remains a stack-based virtual machine. The use of quickening allows us to offset any additional complex instruction decoding logic machinery that is inherently necessary in the Dis interpreter—since most of the encoded information is static, we expect that using our combination with quickening would be able to eliminate much of the overhead of the instruction decoding in Dis. We are not aware of any prior publication of an optimization technique that combines data object reference inlining with quickening.

Concerning the caching of host language local variables in the stack frame of the executing language (Section 3.4), we find that Ertl briefly describes how this could work for register based virtual machines [Ert95], and notes that this invariably leads to explosion of specialized instructions for all possible combinations of instructions and registers. Theoretically, this corresponds to our optimization when a compiler decides to promote our caching variables to dedicated registers. However, we use this technique to optimize load instructions of a stack based interpreter architecture and provide details of choosing which variables to cache for maxi-

mizing payoff. Finally, there are presentational issues: While Ertl presents his idea in assembly, we show our implementation in C.

Regarding our second contribution, the elimination of reference count operations, we cite the following related work. Introduced by Collins in 1960 [Col60], Deutsch and Bobrow found in 1976 [DB76] that while reference counting has its advantages, the amount of reference count operations caused by local stack modifications, i.e., load and store operations, have a considerable negative impact on the performance of such systems. Hence, Deutsch and Bobrow suggest to remove the immediate processing of reference count operations from the mutator and defer them to a dedicated processing phase—similar to the explicit garbage collection phase of other automatic memory management techniques. Because of their introduction of deferred reference counting, the original reference counting approach is often described as immediate or non-deferred reference counting. As early as 1977, just a year after the deferred reference counting approach described by Deutsch and Bobrow [DB76], Barth described a technique to eliminate reference count operations using a global data-flow analysis in a compiler [Bar77]. In addition to what we describe, Barth's description is able to eliminate more reference count operations than our approach. While our approach works for stack-based interpreters, Barth's description optimizes a derivative of Pascal that uses reference counting for automatic memory management. Unfortunately, he does not give any evaluation we could use for comparison purposes. Much of the following research on optimizing reference counting focuses on deferred reference counting as suggested by Deutsch and Bobrow [DB76]. Ungar and Patterson [UP82] describe a set of optimization techniques to eliminate redundant reference count operations from the implementation of standard Smalltalk instructions, such as eliminating an increment and decrement reference count operation by directly copying a value from the callee stack frame to the caller stack frame and nilling out the source. These optimization techniques are static and do not take dynamic instruction sequences into account, which is precisely what allows us to eliminate large amounts of reference count operations. As recently as 2006, however, Joisha took up the basic idea of Barth—with much more comprehensive goals [Joi06]. The basic idea is to use data-flow analysis to optimize a research version of a C♯ compiler that generates code with reference counting for automatic memory management. Joisha uses liveness properties of objects to remove way more reference count operations than our simple approach is able to recognize. His work addresses the "coalescing" of reference count operations that basically corresponds to our approach—but it is only a minor part in his work. His subsequent work of 2008 describes ways to eliminate reference count operations in the presence of modern object-oriented constructs, such as exceptions [Joi08]. While his work achieves a much higher elimination rate of reference count operations, it is certainly not easily realizable in our setting. Our approach does not require any kind of data flow analysis or fix-point computation, but on the other hand can not possibly eliminate as many reference count operations. To the best of our knowledge, there is no similar work for elimination of reference count operations in interpreters.

7. Conclusions

We presented a set of optimization techniques to further improve our maximum speedup up to 2.18, which is an additional speedup of up to 1.14 from our previous maximum speedup of 1.92 [Bru10b]. Since our presented optimization techniques aim at improving the efficiency of instruction implementation, they can be freely combined with orthogonal optimization techniques targeting instruction dispatch—which our combination with threaded-code dispatch clearly demonstrates. We expect to further improve the efficiency of our techniques by combining them with the dynamic superinstructions optimization, as described by Ertl and Gregg [EG03a, EG04].

Another starting point for future work is to use parts of our optimizations to attack the last remaining part of inefficiency of the Python interpreter: optimizing the (un-) boxing of Python objects. A simple strategy would be to pregenerate instructions working on native machine integers, floats, etc. and use a simple abstract interpreter to find sequences of instructions that could safely use native machine primitives instead of the Python objects. We estimate that such optimizations would have a big performance impact, particularly on numerical benchmarks.

Finally, our techniques enable a compiler to do further inlining, which has consistently shown to be crucial for interpreters. Unfortunately, `gcc` is currently not able to do cross-module inlining, which prevents this optimization from being applied. Future work will address this issue to enhance performance even further.

Acknowledgments

I am very grateful to Anton Ertl and Andreas Krall for providing valuable hints concerning the related work of interpreter instruction formats and the caching of host language variables. Furthermore, the author is deeply indebted to Jens Knoop for his many suggestions and ideas on clarifying and improving the presentation and organization of the material. Finally, many thanks go to the anonymous reviewers, who provided constructive comments and detailed feedback for additional polishing of rough edges and correcting unfortunate mishaps.

References

[Bad82] Scott B. Baden. High performance storage reclamation in an object-based memory system. Technical report, Berkeley, CA, USA, 1982.

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, Lecture Notes in Computer Science, pages 18–25. Springer, 2009.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[Bru09] Stefan Brunthaler. Virtual-machine abstraction and optimization techniques. In *Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09)*, volume 253(5) of *Electronic Notes in Theoretical Computer Science*, pages 3–14, Amsterdam, The Netherlands, December 2009. Elsevier.

[Bru10a] Stefan Brunthaler. Efficient inline caching without dynamic translation. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, pages 2155–2156, New York, NY, USA, March 2010. ACM.

[Bru10b] Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21-25, 2010 (ECOOP '10)*, volume 6183/2010 of *Lecture Notes in Computer Science*, pages 429–451. Springer, 2010.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[dal07] Dalvik VM Instruction Formats. Online, (checked: May 2010) 2007.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)*, pages 278–288, New York, NY, USA, 2003. ACM.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, November 2003.

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04)*, pages 7–14, New York, NY, USA, 2004. ACM.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI '95)*, pages 315–327, 1995.

[Ful] Brent Fulgham. The computer language benchmarks game. `http://shootout.alioth.debian.org/`.

[Int08] Intel. Intel turbo boost technology in Intel core microarchitecture (nehalem) based processors. online, November 2008.

[Joi06] Pramod G. Joisha. Compiler optimizations for nondeferred reference counting garbage collection. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*, pages 150–161, New York, NY, USA, 2006. ACM.

[Joi08] Pramod G. Joisha. A principled approach to nondeferred reference-counting garbage collection. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*, pages 131–140, New York, NY, USA, March 2008. ACM.

[Kra84] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983, reprinted with corrections 1984.

[Ler03] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[PD82] Steven Pemberton and Martin Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[UP82] David M. Ungar and David A. Patterson. chapter 11, Berkeley Smalltalk: Who Knows Where the Time Goes?, pages 189–206. September 1982.

[WB82] Allen Wirfs-Brock. *Smalltalk-80: Bits of History, Words of Advice*, chapter 4, Design Decisions for Smalltalk-80 Implementors, pages 41–56. In Krasner [Kra84], 1982.

[Whe10] David Wheeler. sloccount. `http://www.dwheeler.com/sloccount/`, May 2010.

[WMG10] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM SIGMICRO/SIGPLAN International Symposium on Code Generation and Optimization (CGO '10)*, pages 278–287, April 2010.

[WP97] Phil Winterbottom and Rob Pike. The design of the Inferno virtual machine. In *Proceedings of the 42nd IEEE Computer Society International Conference, San Jose, California, US (COMPCON '97)*, pages 241–244, 1997.

[YWF09] Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th Symposium on Dynamic Languages (DLS '09)*, pages 79–88, New York, NY, USA, 2009. ACM.