Understanding the Potential of Interpreter-based Optimizations for Python

UCSB Technical Report #2010-14 August 11, 2010

Nagy Mostafa Chandra Krintz
Computer Science Department
University of California, Santa Barbara
{nagy,ckrintz}@cs.ucsb.edu

Calin Cascaval David Edelsohn Priya Nagpurkar Peng Wu
IBM T.J.Watson Research Center
Yorktown Heights, NY
{cascaval,edelsohn,pnagpurkar,pengwu}@us.ibm.com

ABSTRACT

The increasing popularity of scripting languages as general purpose programming environments calls for more efficient execution. Most of these languages, such as Python, Ruby, PHP, and JavaScript are interpreted. Interpretation is a natural implementation given the dynamic nature of these languages and interpreter portability has facilitated wide-spread use. In this work, we analyze the performance of CPython, a commonly used Python interpreter, to identify major sources of overhead. Based on our findings, we investigate the efficiency of a number of optimizations and explore the design options and trade-offs involved.

Keywords

dynamic scripting language, performance profiling, language interpretation

1. INTRODUCTION

Scripting languages such as Python, Ruby, PHP, and JavaScript have experienced rapid uptake in the commercial sector for software development and general widespread use in recent years. The popularity of these languages stems from two factors: 1) rapid prototyping through high level syntax, language flexibility and dynamism, and a large collection of frameworks and libraries, thus enabling programmer productivity; and 2) cross-platform portability – interpreters and runtime systems are available on a large number of systems. These features make it easy for non-experts and experts alike to employ these languages quickly for a wide variety of tasks and to deploy fairly complex applications on a variety of platforms.

Many of these languages were originally designed for "scripting", i.e., for writing short programs for text processing or as "glue code" between (or embedded within) modules or components written in other languages. Recently, they are increasingly employed for more complex, general-purpose, and self-contained applications. For example, Python is used in Linux for software packaging and distribution, for peer-to-peer sharing [3] and computer aided design [8, 19], for cloud computing [10, 5] as well as computationally intensive tasks [24]. PHP and JavaScript are commonly used for embedded server-side and client-side embedded scripting, respectively; Google uses JavaScript for their desktop applications, including GMail and Google Docs, while PHP is the most commonly used language for writing wiki software [30].

Development frameworks (e.g. Rails for Ruby and PHP (TRAX), Django for Python, etc.), high level syntax, and language dynamism (runtime resolution of variable types and extensive use of runtime reflection) all play a key role in the use of dynamic languages for increasingly complex applications.

Although researchers and open source efforts continue to pursue ways of dynamically compiling these programs [21, 22, 26, 29, 11, 9], the most popular versions of the distributed runtimes for dynamic scripting languages employ interpreters for execution. Interpretation plays a key role in the wide-spread use of these languages – the ease of development facilitates rapid prototyping and language evolution. Moreover, the runtime is architecture-independent (e.g., written in C) and thus can be compiled to any system without significant porting effort, as opposed to retreating a compiler or JIT system. Even when compilation is an option, dynamic languages are challenging to compile because types are not known statically, and must be inferred and speculatively translated.

Interpretation, however, is an inherently inefficient execution engine. The interpreter must decode and dispatch each bytecode (compact representation of source instructions). This incurs an overhead not present in the execution of native (compiled) code. In addition, interpretation of each bytecode is done independent of every other, resulting in poor code quality compared to code produced by even the simplest code generator. There has been significant research to identify ways of optimizing the interpretation process for non-scripting, statically typed, high-level languages (e.g. Java, OCaML), while maintaining the portability of the runtime [23, 17, 7, 31]. Such approaches target the interpreter dispatch loop since significant time is spent there for these languages.

Modern scripting languages however are very different from OCaML and Java given their dynamic nature. In this paper, we show that dispatch optimization provides only minimal benefits to Python programs. We, hence, investigate the primary sources of overhead and based on our findings, we identify and evaluate a set of optimizations that target these sources of overhead in an attempt to improve the performance of Python programs. Using caching of attributes to avoid name lookup, elimination of loads/stores to/from the operand stack, and inlining, we are able to extract performance improvement of up to 28%.

2. BACKGROUND

Python is a general-purpose high-level object-oriented programming language with dynamic typing. There are several implemen-

Benchmark	Description
2to3	A Python 2 to Python 3 translator translating itself
django	Django Python web framework building a 150x150-cell HTML table
html5lib	Parse the HTML 5 specification using html5lib
pickle	Use the pure-Python pickle module to pickle a variety of datasets
pybench	Run the standard Python PyBench benchmark suite
richards	The classic Richards benchmark
rietveld	Macrobenchmark for Django using the Rietveld [20] code review app
spambayes	Run a canned mailbox through a SpamBayes [25] ham/spam classifier
unpickle	Uses the cPickle module to un-serialize a variety of datasets

Table 1: The Unladen-Swallow Benchmarks

tations of the Python language [13, 14, 27]. In this work, we focus on the most widely used reference implementation: CPython [6]. The CPython is a simple implementation of the language written in C that is portable and easy to understand and extend. CPython employs switch-based interpretation and a combination of simple reference counting and cycle-detecting garbage collection. It compiles Python source to high-level type-generic bytecodes that run on a stack-based virtual machine. Being type-generic bytecodes, they must defer associating abstract operations with specific implementations until runtime. This is carried out via a sequence of type-checks and indirect-branches which makes bytecode handling slower than for statically-typed languages. CPython performs name-based late binding on variables and methods (attributes). Every Python object has a dictionary (a hash table), that maps attributes names to values. When loading an attribute, a chain of hash table lookups are performed on the receiver object and its type and super-types. Not only attributes are stored in dictionaries, so are global variables.

CPython also implements descriptors which are setter/getter objects that are used to associate an action with attribute access. Among these actions is binding methods to objects dynamically at runtime. When a method is loaded from an object dictionary, its descriptor is found instead which, when invoked, created the method object on-the-fly and binds it to the receiver.

3. METHODOLOGY

To understand the behavior of Python programs, their sources of overhead and the impact of interpreter optimizations on CPython, we evaluate the performance of benchmarks out of the Unladen Swallow project [28] which are listed and described in Table 1. These benchmarks exercise common activities found in Python programs including parsing and translation, (de-)serializing datasets, and HTML manipulation. Also included in this list is `pybench`, which implements a set of microbenchmarks that exercise low level Python activities including function calls, comparison operators, looping constructs, string manipulation, basic arithmetic, and others [18].

We modified the CPython-2.6 [6] source to collect a variety of profiles and to experiment with different optimizations. We ran our experiments on and Intel Core 2 64-bit machine clocked at 2.66 GHz with 2x32 KBytes of L1 instruction cache and 4 MBytes of shared L2 cache. running Linux 2.6.24 patched with Perfmon [16], the hardware performance monitoring interface for Linux.

4. PERFORMANCE ANALYSIS

Using this methodology, we first characterize the performance of CPython. We start by investigating how Direct-Threaded Interpretation (DTI), a traditional interpreter optimization to improve bytecode dispatch time, impact Python/CPython performance. The aim is to demonstrated that such optimization is less effective for a dynamic language.

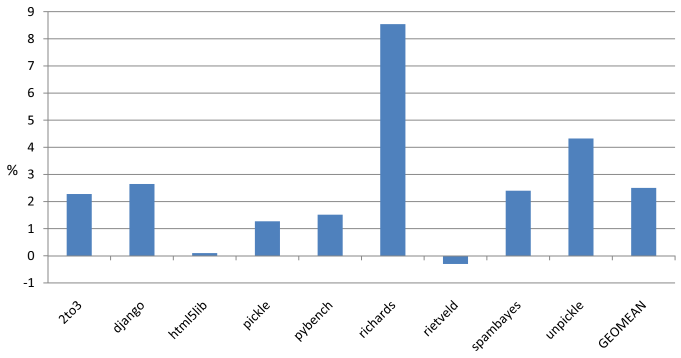


Figure 1: Effect of Direct Threaded Interpretation on CPython

DTI is a technique that replaces the opcode of every bytecode (upon first execution or ahead-of-time) with the address or label of its handler [1]. This replacement increases the size of each instruction (and still requires replicated code for decoding at the end of each handler) but reduces the overhead of indirect branches in two ways. (1) For every indirect branch, the number of possible targets decreases, which enhances target prediction. (2) Any biased relation between opcodes is exploited. Such optimization and others reduce the overhead of conditional and indirect branches, which are typically hard to predict and are thus costly on modern architectures, and proves efficient for statically-typed languages where the dispatch process is the bottleneck [7, 2].

Figure 1 shows the effect of Direct-Threaded interpretation on CPython performance. On average, there is a small speedup of 2.5%. For most benchmarks, speedup is around 2% and in one case we actually get a slowdown. This results validates our claim and shows that bytecode dispatching is not a bottleneck anymore for interpreters with high-level of abstraction.

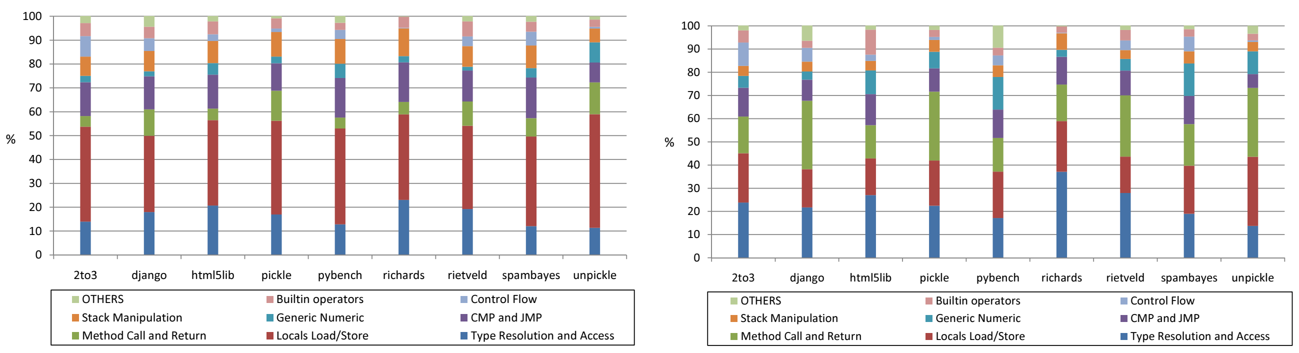
To understand, at a high level, where time is being spent in Python programs, we classify the bytecodes into several classes based on their actions. For each class, we measure the number of bytecodes executed and the percentage of time spent there. The classes we use are as follows:

Type Resolution and Field Access includes `LOAD/STORE_ATTR` and `LOAD/STORE_GLOBAL`. When given an attribute name and a receiver object, `LOAD/STORE_ATTR` perform the necessary work to resolve the attribute name. This operation is expensive and involves a number of indirections and dictionary lookups, specially if the attribute lies up in the inheritance chain.

`LOAD/STORE_GLOBAL` perform the same task for global variables. They look for the attribute name in the global dictionary, if not found they look for it in the builtins dictionary. Since at most two lookups are necessary, they are faster than `LOAD/STORE_ATTR`. In the standard CPython implementation, these lookups are performed every time one of these bytecodes are executed.

Locals Loads/Stores are bytecodes that transfer values between the locals/constants and the operand stack. There are three bytecodes in this class. Namely, `LOAD_FAST`, `STORE_FAST` and `LOAD_CONST`. These bytecodes are cheap, yet, as will be shown later, they are encountered quite often during execution.

Method Call and Return includes bytecodes that perform method calls and returns. The most common opcode in this class is `CALL_FUNCTION`, which pops a function object and its arguments from the operand stack and invokes it. This bytecode is used for calling Python functions as well as C (built-in) functions depend-



(a) Counts (b) Cycles

Figure 2: Bytecode Histograms based on bytecode class.

ing on the function object it operates on. When calling a Python function, setting up a method frame and copying arguments to it is part of the overhead. Obviously, calling a C function is cheaper than calling a Python one.

Compare and conditional jump includes `COMPARE_OP` and opcodes with the prefix of `JUMP_IF`. These bytecodes produce or consume generic boolean objects. These bytecodes are amenable for type specialization and unboxing.

Generic Numeric includes generic bytecodes for Python numeric and string objects, such as the ones with the prefix `BINARY`, `UNARY`, or `INPLACE`. These bytecodes are expensive as they are type-generic; thus involve type-checks and indirect-jumps. They are also good candidates for optimizations such as type specialization and unboxing.

Stack Manipulation are bytecodes that manipulate the top elements on the operand stack. For example, `DUP_TOP` duplicates the top element on the stack.

Control Flow includes bytecodes to manipulate loops such as `GET_ITER` and `FOR_ITER`. These bytecodes are also good candidates for type specialization as loop iterators are typically range of integers.

Built-in Operators includes bytecodes for built-in container types such as list, map, and tuple, e.g., `LIST_APPEND`. These bytecodes save the necessary calls to the runtime to achieve the same functionality.

Others includes all others.

Figure 2 shows the count and time distribution for each class of bytecodes, respectively. The percentages are from the total cycles/counts for all bytecodes. We make the following observations from the bytecode profile:

1. **Type Resolution and Access** *consumes a significant fraction of time.* For most benchmarks, this classes of bytecodes takes more than 20% of the time. In the CPython implementation, subsequent executions of the same `LOAD_ATTR`, say in a loop, causes the type resolution to be redone even in cases when invariance of the outcome is guaranteed. CPython does no effort to optimize this save for caching of bound methods by the type they are bounded to and the method name. This cache, however, is referenced late in the bytecode handler, making it inefficient.

2. *The most frequently executed bytecodes are* **Locals Loads/Stores**. Although they are cheap operations, being executed repeatedly results in considerable amount of time spent on moving values from/to the operand stack.

3. *Significant time spent in* **Method Call and Return**. Note that the bar for cycle distribution of this class in Figure 2.b includes only the 'overhead' of calling a method. It does not include the time spent in the target method, nor time for resolving the function via its name. The disproportionately of the large distribution in the 'Cycles' compared to its distribution in 'Counts' in Figure 2 suggest high overhead in method call. Both of these results indicate potential for optimizations that either eliminate or reduce method call overheads, such as method inlining.

4. **Type Resolution and Access, Locals Loads/Stores and Method Call and Return** *account for more than 50% of the time.* Each of these classes of bytecodes consume a reasonable amount of execution time, but when combined they dominate the execution frequency and time for all benchmarks. This behavior calls for a simple optimization to target each type of the three bytecode classes.

5. OPTIMIZATIONS

Our analysis of Python programs over CPython motivates us to investigate the overheads caused by the three primary sources of overhead: type resolution and access, local variable loads/stores to/from the operand stack, and method call/return. Our goal is to identify, design, and implement simple interpreter optimizations that target each source in an effort to improve Python performance and to better understand the challenges we face in doing so.

For type resolution and access, we propose a caching scheme to reduce access time and to avoid dictionary lookups. For local variable loads and stores, we investigate a mechanism that eliminates loads and stores from the bytecode stream. For method call/return, we investigate inlining opportunities. For each optimization, we examine its usefulness and how well-suited it is for the overhead it targets. Moreover, we examine the design parameters and tradeoffs using performance profiles.

5.1 Attributes Caching

Loading global variables and object attributes in Python is a common, yet expensive, operation. One reason behind this is that, due to the dynamic nature of the language, global variables and object attributes are stored in hash tables (dictionaries) and are referenced by their names. A single read can involve several dictionary lookups. Another source of overhead is (indirect) function calls that the runtime performs to handle a load. Two bytecodes are most frequently used for globals and attributes loading: `LOAD_GLOBAL` and `LOAD_ATTR`.

`LOAD_GLOBAL`, given a global variable name, resolves it as follows:

1. Look up variable name in the globals dictionary. If found, return it; else

2. look up in the built-ins dictionary. If found, return it; else
3. raise exception.

LOAD_ATTR is given a receiver object and an attribute name. The lookup process is fairly complex and proceeds in the following order:

1. Look up the attribute name in the dictionaries of the method resolution order (MRO) structure of the receiver object. If the entry is found and is a data descriptor, invoke its getter to read the attribute; else,
2. look up in the instance dictionary of the receiver object. If found, return attribute; else,
3. if the entry found in the MRO is a non-data descriptor, invoke its getter to read attribute; else,
4. return the entry found in the MRO as the attribute; else,
5. raise exception.

To avoid the above chain of lookups, we propose two caching schemes, which we next overview.

5.1.1 LOAD_GLOBAL *Cache*

Design The operand of a LOAD_GLOBAL is the index of a global variable name in the pool of names used by the code. On code loading, we replace the operand of every LOAD_GLOBAL to point to a structure that holds the original operand in addition to a caching structure (Figure 3). Thus, each LOAD_GLOBAL has its own single-entry cache. The cache holds the following values: a pointer a dictionary (dictObject), a pointer to a dictionary entry (dictEntry), a version number and an execution frequency counter.

Initially, each LOAD_GLOBAL is executed normally, following the slow path, and the execution frequency counter is incremented. Once a LOAD_GLOBAL gets hot (execution frequency goes above a threshold, which we chose to be 100 based on performance tuning), its cache is initialized. The cache holds a pointer to the dictionary where the variable was found and another pointer to the dictionary entry that holds it. Subsequent executions of a cached LOAD_GLOBAL will use the dictionary entry pointer to fetch the value directly. Since a pointer to the dictionary entry containing the variable, and not the variable itself, is cached, we guarantee that the value fetched is always correct.

Invalidation Although caching a dictionary entry pointer makes the cache valid even if the global variable value has changed, there are still reasons for which cache invalidation is needed:

1. **Dictionary shape change**: The shape of a dictionary changes if an element is deleted or the hash table representing the dictionary is resized. Thus, the cached dictionary entry pointer becomes invalid.
2. **Shadowing**: A variable name shadows another if it has the same name as an existing variable in another dictionary, and the two dictionaries are employed in the same lookup chain. For example, if the dictionary entry of the variable "foo" is already cached from the builtins dictionary, inserting a new variable "foo" in the globals dictionary makes the cache invalid.

We detect these cases by attributing a version number to every dictionary. Version numbers are incremented whenever a dictionary shape changes. When caching a variable from a dictionary, the version number is copied to the cache. If the cached version number and the dictionary version number(fetched using the cached dictionary pointer) do not match, a cache miss is declared and we bail out to the slow execution path and update the cache. Whenever a new variable is added to the globals dictionary (STORE_GLOBAL), the version number of the built-ins dictionary is incremented; thus, we conservatively invalidate all cached variables from the built-ins dictionary since there is a possibility they have been shadowed by the new insertion. Although, seemingly over-conservative, this simple solution is sufficient since adding new variables happens mostly at the beginning of execution and is quite rare afterwards.

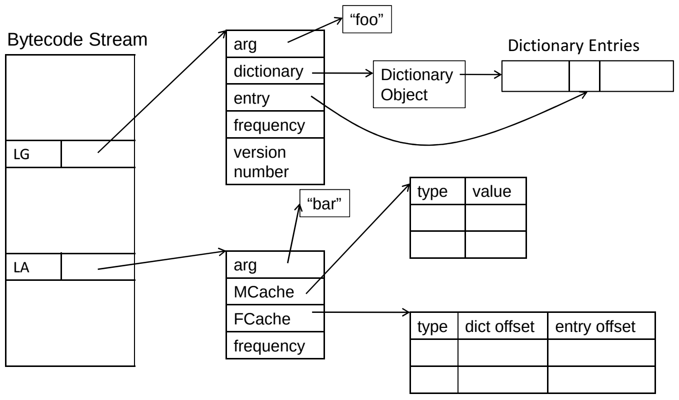


Figure 3: Cache layout for LOAD_GLOBAL and LOAD_ATTR

5.1.2 LOAD_ATTR *Cache*

Every LOAD_ATTR has potentially two cache structures, one for methods (MCache) and one for instance fields (FCache). Both caches are referenced by the type of the receiver object. The allocation of these caches, however, is deferred until it is known that the LOAD_ATTR is amenable to caching. Every LOAD_ATTR can be in any of the following five states. We express each using a different, specialized opcode:

- LOAD_ATTR This is the initial state for all LOAD_ATTRs. In this state, we check if the LOAD_ATTR is "cacheable" or not. A cacheable LOAD_ATTR is a one that follows the normal way of attribute access [1]. If the LOAD_ATTR is cacheable, its opcode is rewritten to be LOAD_ATTR_CACHEABLE, else LOAD_ATTR_NORMAL.
- LOAD_ATTR_NORMAL A non-cached LOAD_ATTR following the normal way of attribute access.
- LOAD_ATTR_CACHEABLE A cacheable LOAD_ATTR. At this state, if the LOAD_ATTR is executed more than two times, it is transformed into either LOAD_ATTR_CACHED_M or LOAD_ATTR_CACHED_F based on whether it loaded a method or an instance field.

  LOAD_ATTR_CACHED_M A cached LOAD_ATTR that loads a method from a Type object. Hence, loads from its MCache.

  LOAD_ATTR_CACHED_F A cached LOAD_ATTR that loads an instance field. Hence, loads from its FCache.

[1]This is the case with tp_getattro field in a Type object points to PyObject_GenericGetAttr(). Using CPython C-API, users can define their own functions to get attributes, in which case tp_getattro will point to a user-defined function making LOAD_ATTR uncacheable.

We define an instance as any object that is not a Type object (which cannot be instantiated). If an instance field is loaded, be it a method or not, the FCache is used. Although, in theory, it is possible for a `LOAD_ATTR` to mix loading of methods from Type objects and fields from instance objects, in all the cases we have studied, we have not encountered such a case.

When a cached `LOAD_ATTR` is executed, it reads the type of the receiver object on the top of the operand stack. The type is then used to reference either the MCache or the FCache. For the MCache, the value of the method object, or rather the non-data descriptor used to create it is cached [1]. For the FCache, since we are referencing instance fields, the offset of the field in the receiver object (dictionary offset and entry offset) is cached. If the type referenced is not found, we add a new cache entry and resize the cache, if necessary. All caches start with a single entry, upon the first resize, we make it five entries. For all future resizes, we double the number of entries.

MCache Invalidation Since in the MCache, we cache the method descriptor, not the dictionary entry containing it, dictionary shape changes do not invalidate the cache. In fact, the cache is always valid under the assumption that methods are not modified once created. This means that after the declarative steps where Python classes are created and methods are attributed to them, no meta-programming is done to manipulate those methods or to create new ones dynamically. For all of the code we experimented with, we did not face any cases where this assumption fails. For code with heavy meta-programming activity, the MCache can be simply turned off for correctness.

FCache Invalidation Caching of instance fields is based on the assertion that objects of the same type will most likely have dictionaries of identical shape. As we demonstrate later, this assertion is true. In such cases, for all objects of the same type, it is sufficient to cache the offset of the instance field instead of having a cache entry per object. It remains, however, to guarantee that the fetched field is the correct one. We achieve this by comparing the `LOAD_ATTR` operand with the key fetched from the dictionary. Since strings are interned in CPython, this is a quick equality check of pointers.

5.1.3 Analysis

We carried out a set of experiments to understand the effectiveness of the caching scheme. Figure 4 shows the ratio of dynamic `LOAD_ATTR`s that cannot be cached. Most of the benchmarks have a negligible percentage of executed non-cacheable `LOAD_ATTR`s (below 5%). On average, only 0.7% of the `LOAD_ATTR`s are non-cacheable. This result, although expected, shows the applicability of the caching scheme on the majority of `LOAD_ATTR`s in the code.

For the `LOAD_ATTR` caching, we experimented with three possible varieties:

1. **LA multi swap** The cache is resizable, starting from a single entry. The last referenced cache entry is swapped with the top of the cache as a simple heuristic to exploit locality of reference.

2. **LA multi no swap** Same as above, but without swapping. We test this configuration because if the majority of the caches are small then swapping might only incur overhead without much benefit.

3. **LA single** Each cache has only a single entry which is overwritten on a cache miss.

[1] In CPython, the Type object dictionary contains a non-data descriptor of the method, instead of the method object itself. Method objects are allocated on the fly, when referenced, where they are bound to their receiving objects.

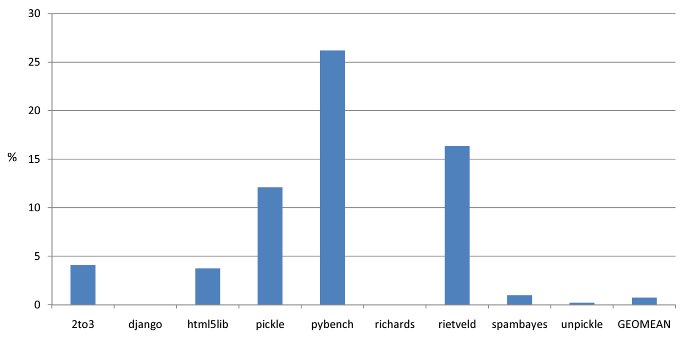


Figure 4: Percentage of non-cacheable `LOAD_ATTR`s

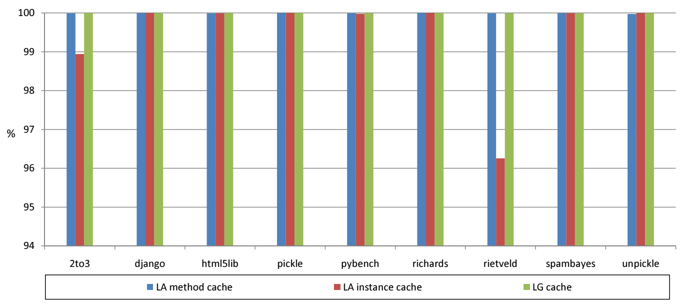


Figure 6: Cache hit rate for all three types of cache: `LOAD_ATTR` method and instance field caches and `LOAD_GLOBAL` cache

We test the above configurations with and without `LOAD_GLOBAL` caching. Figure 5 reports on the speedup achieved for all combinations. We can see that `LOAD_GLOBAL` caching by itself is not sufficient to gain a reasonable speedup and can even lead to a slowdown. Notice, however, that for nearly all benchmarks, adding `LOAD_GLOBAL` caching to `LOAD_ATTR` caching improves overall performance (up to 16%), despite that in some cases this happens by mere constructive interference of the two optimizations (pickle and spambayes). Overall, the best configuration to use is `LOAD_GLOBAL` caching with multi-entry `LOAD_ATTR` caching with swapping which achieves 8% speedup.

Figure 6 shows the hit ratio for all caches using the multi-entry with no swapping configuration. The Figure shows a nearly perfect cache performance. Most of the cache misses are actually due to a cold cache. The instance field cache exhibits the highest miss rate. This is the effect of few `LOAD_ATTR`s that operate on a variety of objects having one common attribute.

It may appear at first that we achieve perfect caching due to the unlimited resizing that we allow. This is not the case, however. Figure 7 presents a histogram of the caches sizes for the `LOAD_ATTR`s caches. The vast majority of the caches have a single entry. This is most interesting for instance field caches. The maximum cache size reached is 80 entries; only two programs reach this maximum size for multiple caches (5 caches for 2to3 and 18 for pybench).

Finally, we look at the impact on memory usage. Figure 8 shows the extra memory needed in KiloBytes for multi-entry `LOAD_ATTR` cache and `LOAD_GLOBAL` cache. The memory includes all necessary data structures and cache entries used for maintain the cache. Most benchmarks consume less than 150KB of extra memory. Naturally, the `LOAD_ATTR` cache consumes more memory due to the dynamic nature of the bytecode where more cache entries are needed. The size of one `LOAD_ATTR` cache entry is 16

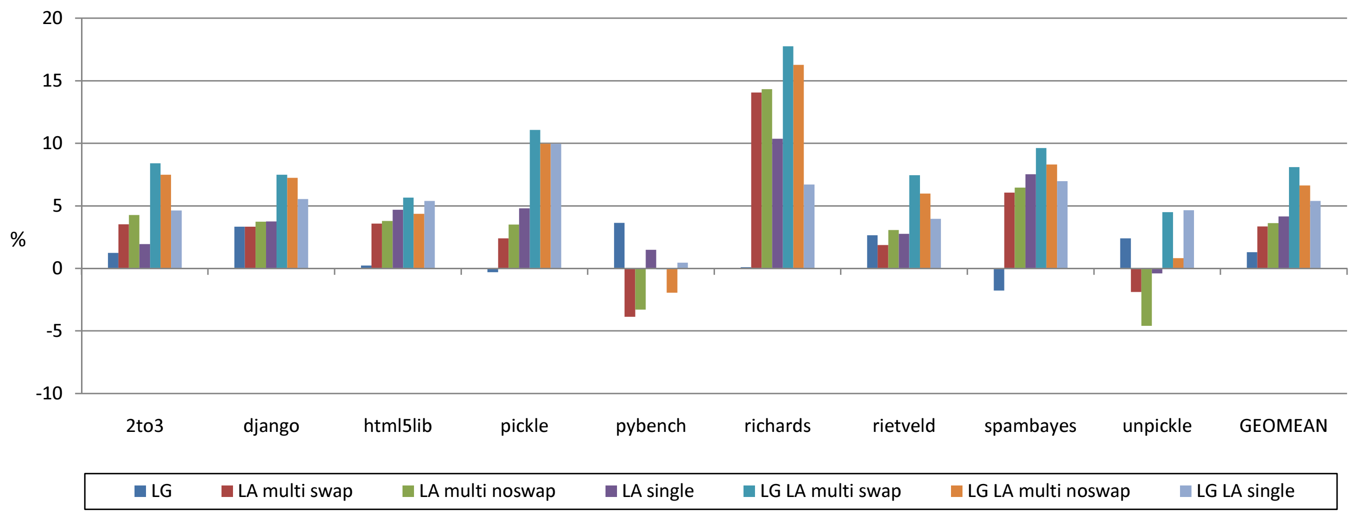


Figure 5: Speedup of different combinations of caching

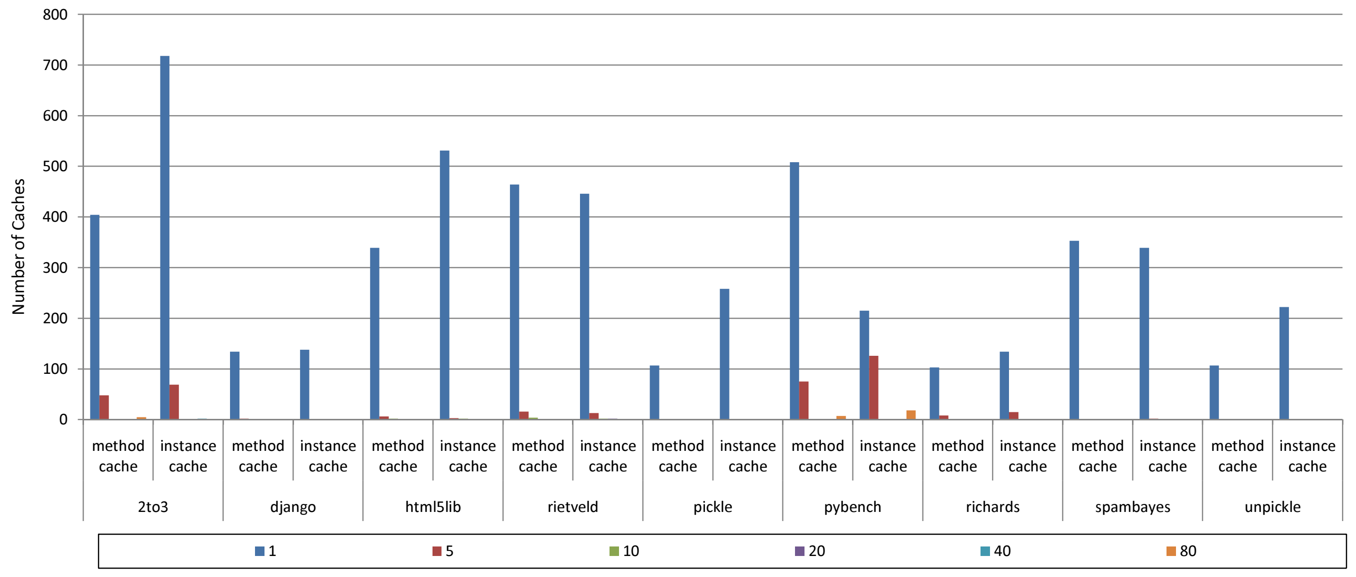


Figure 7: A histogram of the `LOAD_ATTR` cache sizes. It shows the state of the cache sizes at program termination

byte, while for the `LOAD_GLOBAL` cache it is 28 bytes.

Notice that in Figure 7, 2to3 has a higher number of `LOAD_ATTR` cache entries than rietveld, yet rietveld consumes more memory. The reason is that some data structures are allocated statically for some `LOAD_ATTR`s without necessarily allocating cache entries for them (e.g. if they are never executed). Hence, the allocated structures add to the total memory size and not to the total number of cache entries.

5.2 Load/Store Elimination

We next consider how to eliminate loads and stores – the bytecode instructions that move values between local storage and the operand stack. In this Section, we investigate a static bytecode transformation technique that converts certain bytecodes from stack-based to register-based versions. The latter access the locals from the virtual local registers directly.

We perform the operation selectively on bytecodes for which the transformation eliminates the need to copy values to/from the stack. Figure 9 shows a simple example of the process. In the given a basic block of bytecodes, three values are loaded on the stack via a `LOAD_GLOBAL`, a `LOAD_CONST` and a `LOAD_FAST` which are then consumed by `STORE_SUBSCR`. By converting the `STORE_SUBSCR` to a register-based version (`R_STORE_SUBSCR`), we can eliminate the `LOAD_CONST` and the `LOAD_FAST`.

We copy the source local register address from the loads as arguments to the new register-based bytecode which can now read its operands directly from the locals, saving four memory references. Notice, that it is not possible to eliminate the `LOAD_GLOBAL` since it fetches its variable from a dictionary. In such cases, we leave the `LOAD_GLOBAL` and place zero (underlined) in the `R_STORE_SUBSCR` argument list. This indicates that the read should be from the stack. `R_STORE_SUBSCR` thus operates by reading its argument list left-to-right; for non-zero arguments, the values are read from registers, else they are popped from the operand stack.

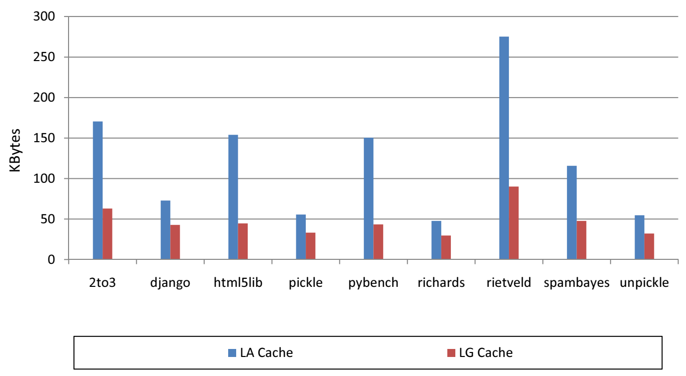


Figure 8: Amount of memory, in KiloBytes, needed for all caches

```
Before
0:     LOAD_GLOBAL      1
3:     LOAD_CONST       1
6:     LOAD_FAST        18
9:     STORE_SUBSCR
10:    LOAD_FAST        1
13:    LOAD_ATTR        5
16:    STORE_GLOBAL     1

After
 0:    LOAD_GLOBAL      1
 3:    R_STORE_SUBSCR   18, 1, 0
10:    R_LOAD_ATTR      5, 1; 0
15:    STORE_GLOBAL     1
```

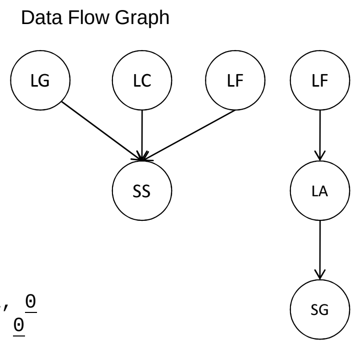


Figure 9: Example of Load/Store elimination with the corresponding Data FLow Graph

Similarly, in Figure 9, we transform `LOAD_ATTR` to employ register-based local variable access. In this case, we eliminate only `LOAD_FAST`. Since `STORE_GLOBAL` inserts a value into a dictionary, the output from the `LOAD_ATTR` is pushed on the stack. Notice that the only loads/stores eliminated are those that operate on registers. Namely, `LOAD_FAST` , `LOAD_CONST` and `STORE_FAST`, which we refer to, in this section, as loads/stores.

We employ a simple static analysis to perform this optimization when we load the bytecode. In particular, we build a control flow graph (CFG) from the bytecode stream and use abstract interpretation to build a data flow graph (DFG) for every basic block. Using the DFG, we select which nodes to transform to register-based using four selection criteria:

- **STRICT** A DFG node is transformed to register-based iff all of its immediate predecessors and successors are loads/stores. This criterion maximizes the number of loads/stores eliminated per transformation.

- **INPUT** A DFG node is transformed to register-based iff all of its immediate predecessors are loads.

- **MAJORITY** A DFG node is transformed to register-based iff the majority of its immediate predecessors and successors are loads/stores.

- **ANY** A DFG node is transformed to register-based iff at least one of its immediate predecessors and successors is a load/store.

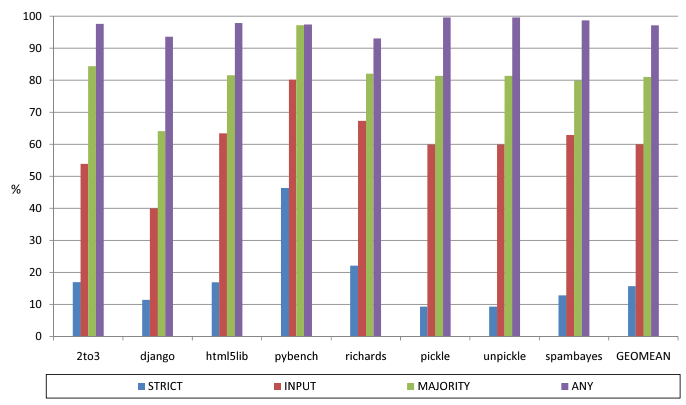


Figure 10: Profile of the percentage of eliminated Loads/Stores for all variations of the Elimination technique

There is a key trade-offs that we make with this optimization. The stack-based version of a bytecode is more compact than the register-based version. In the former, the operands are implicit and are read from the operand stack, while in the latter, each operand's location must be explicitly included in the bytecode. Therefore, we are trading off code size for the number of eliminated loads/stores. We, thus, must transform only when this extra code size is amortized by the elimination of loads/stores.

Each of the above criteria has it advantages and disadvantages in that sense. Doing a STRICT transformation guarantees gain out of every transformed bytecode, but, since the criteria is strict, few bytecodes can be transformed. The INPUT criterion is more relaxed, it requires only the inputs to come from loads, it still transforms when advantageous, yet, being more relaxed, it transforms more bytecodes. MAJORITY is even less relaxed but still applies a simple heuristic that ensures gain. ANY is the most relaxed of all, eliminating the majority of loads/stores in the code while increasing the code size significantly.

Another trade-off that this optimization makes, is the complexity of the register-based bytecode handlers. If a register-based bytecode can mix reading from register and from the stack, then checks are needed in its handler to determine where to read from. For STRICT and INPUT, register-based bytecodes always read from registers, thus the handlers are simple. MAJORITY and ANY require checks.

5.2.1 Analysis

Figure 10 compares the dynamic count of eliminated loads/stores for all four criteria. Rietveld is missing here as we were unable to get it to run with this optimization (we will include it in the final version should this paper be accepted). One can see that STRICT performs poorly and eliminates, on average, less than 20% of the loads/store executed. INPUT is much better with an average of 60%. The numbers go up for MAJORITY and ANY, which eliminate almost all loads/stores. Based on the trade-offs mentioned, we adopt INPUT as our selection criteria. It eliminates more than half of the loads and allows simple implementations of the register-based bytecodes handlers.

The next question we investigate is how many register-based bytecodes to support. Figure 11 addresses this question using a cumulative function of the estimated speedup plotted against the number of bytecodes supported. Based on our experience and our evaluation data, we currently support the register-based version of the 15 bytecodes listed in Table 2

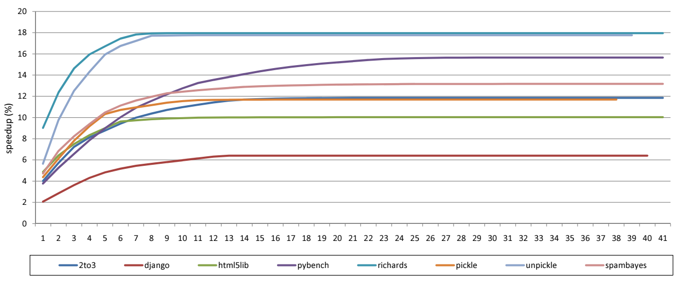


Figure 11: A cumulative function of the speedup estimate plotted agains the number of bytecodes transformed to Register-based

LOAD_ATTR
COMPARE_OP
BINARY_SUBSCR
RETURN_VALUE
SLICE
BUILD_TUPLE
STORE_ATTR
YIELD_VALUE
STORE_FAST
BINARY_ADD
BINARY_SUBTRACT
STORE_SUBSCR
BUILD_SLICE
INPLACE_ADD
BINARY_MULTIPLY

Table 2: Bytecodes for which a register-based version is supported

To support simple register-based bytecode handlers, all locals and constants must be referenced in a uniform manner. This is not the case in CPython, as constants are stored in code objects while locals are part of the virtual call-stack frames. To overcome this, we maintain a copy of the constants of a code object in all call-stack frames that correspond to it.

Figure 12 shows the overhead incurred by the static analysis on the bytecode and the constants copying. We measure this by carrying out all the code transformations without actually using the transformed code. On average, the overhead is less than 2% and for 5 out of the 8 benchmarks shown have less than 1% overhead.

Finally, Figure 13 reports the efficiency of the optimization and some hardware performance metrics. In terms of speedup, we achieve 5% speedup on average and as much as 9%. There is a consistent, and sometimes large, increase in L1 instruction cache miss rate. This is due to the addition of new bytecode handlers in the dispatch loop. Some increase is also seen in the L2 cache miss rate, we attribute this to the code size increase as well as to the data structures that we employ to implement the static optimizations. For all benchmarks, the amount of work (instruction count) performed is reduced.

5.3 Inlining

The last optimization, that we investigate is inlining of method calls. We measure the dynamism of method calls in Figure 14. The Figure divides the calls made into C calls, which are calls to CPython runtime, and into Python calls, which are calls to user-level code. Each category is divided further into monomorphic and polymorphic calls. Almost all of the C calls are monomorphic calls and for some majority of benchmarks, there is no polymorphic C calls made. This is a good indication that C calls are a potential inlining target.

Other evidence is shown in Figure 15, which shows the per-

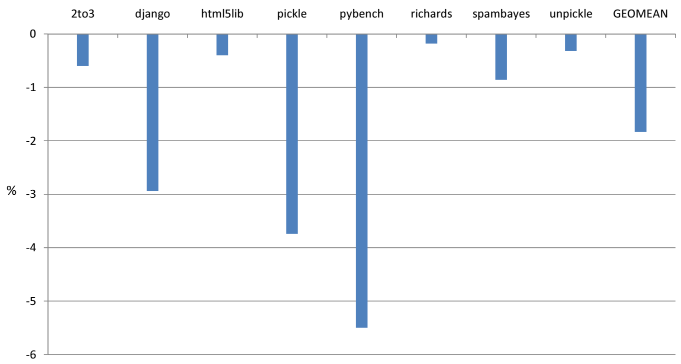


Figure 12: Static analysis and constant copying overhead for Load/Store elimination

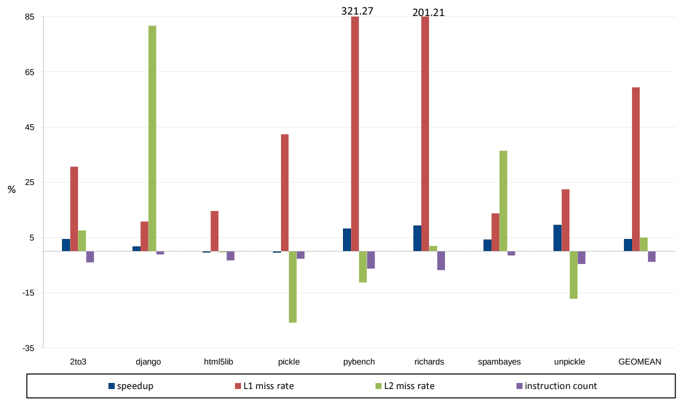


Figure 13: Speedup of Load/Store Elimination optimiziation. The figure also shows effect on L1 ICache and L2 Cache as well as the decrease in the instructions executed

centage of call-sites responsible for 90% of the calls made for C and Python functions. The figure shows that the 90/10 rule holds strongly for calls to C functions, where less than 10% of the C functions call-sites are sources of 90% of the calls invoked. This is not the case for Python functions. These results motivate us to look more closely into the call targets of the most frequent C calls. We find that isinstance() is a commonly used builtin function, especially for django. We tried a simple optimization where we employ a special bytecode to implement this function – to simulate inlining it into Python bytecode. Figure 16 shows the speedup. We attempted to inline additional functions in this way, but adding more opcode handlers degrades performance quickly.

Finally, in Figure 16, we report the speedup of all three optimizations in combination, using all cache configurations. We achieve a maximum speedup of 28% and 15% on average. Multi-entry cache with swapping remains the best performing caching configuration for most cases. Multi-entry with no swapping and single-entry are quite similar performance-wise.

6. RELATED WORK

In this section, we identify research contributions that characterize interpreter performance and that propose techniques for its improvement.

A work by Holkner et al. [12] aims to understand the extent and scope of use of dynamic language features like runtime object- and code modification. In particular, the authors examine whether

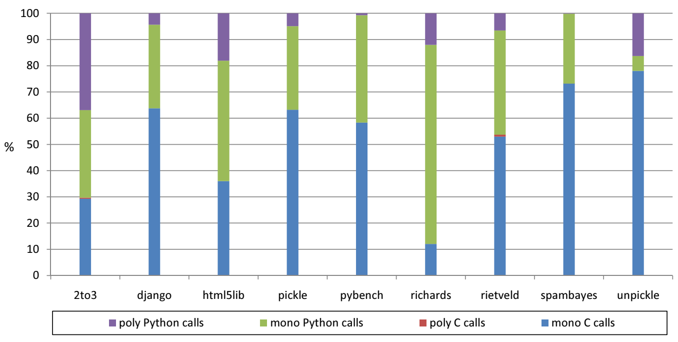


Figure 14: A breakup of the method calls by their morphism and type (builtin or Python)

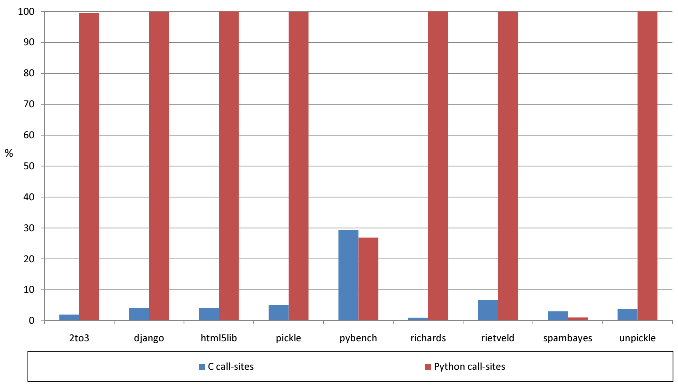


Figure 15: Percentage of C and Python call-sites responsible for 90% of the calls made

Python programs only rarely use dynamic features, and whether this use of dynamic features is restricted to an initial startup phase in the application. For the programs and the set of dynamic features that they analyzed, the authors concluded that while programs do make use of dynamic features over the entire execution, this use is relatively higher during startup, thus lending themselves well to runtime analyses and feedback-directed optimization.

Hidden Classes is a caching optimization used in Google V8 Javascript engine [29]. The idea is to have a table for objects with the same layout mapping attribute names to their offset in the instance object. This technique is well-suited for Javascript since it is prototype-based and there is no notion of classes. In Python, that is not the case and every object is an instance of some class. In our results, we have shown that objects instantiated from the same class show to great extent identical layouts. This was demonstrated with the extremely low miss rate for the instance field cache in Section 5. Additionally, we think that Hidden Classes are more suited for dynamic code generation than interpretation since the generated code has the indices of the referenced Hidden Classes entries inlined within. Adopting Hidden Classes in an interpreter would require a cache, similar to the one we proposed, to cache the indices. This double caching will unlikely yield additional performance gain.

Similarly, in unpublished work [15], Lua language implementers employ caching within code generation. Code accessing hashes with constant keys are specialized for that key/hash. This is similar to polymorphic inline caching [4] where the code generated is specialized based on the outcome of method resolution.

7. CONCLUSIONS

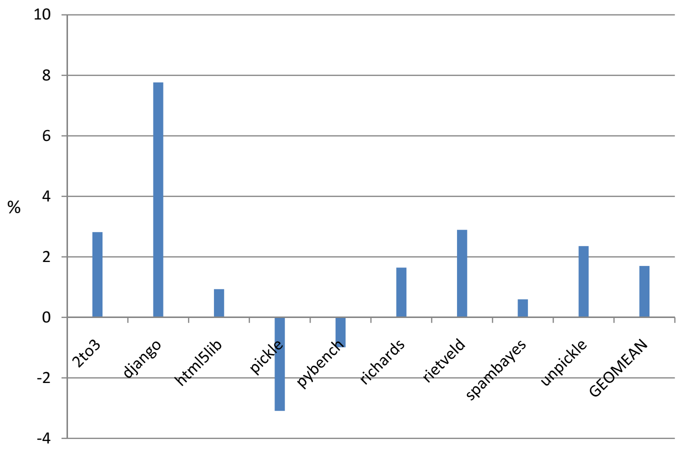


Figure 16: Speedup with a special ISINSTANCE bytecode

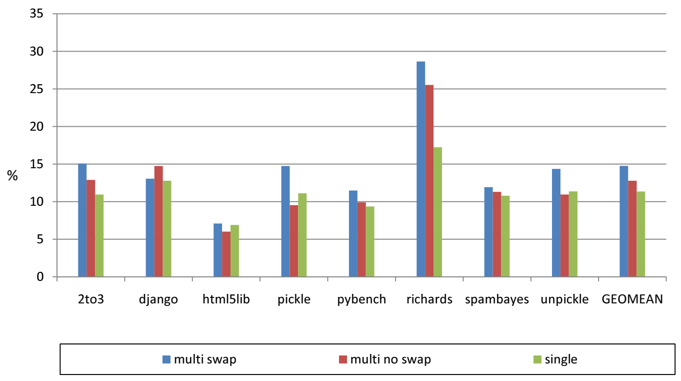


Figure 17: Summary of speedup when all optimization are enabled

In this paper we evaluate the performance of the Python language and CPython interpreter. We perform an analysis of the behavior of this system for a representative set of programs. We find that traditional interpreter optimizations for more static languages do not improve Python performance significantly due to the dynamic nature of the language and its bytecode design. We investigate the primary forms of overhead in CPython and identify three simple optimizations to target this overhead. We find that there are many design trade-offs associated with optimizing Python in a portable way (within the interpreter). However, some performance improvement is possible – we show improvements of up to 28% and 15%on average.

8. REFERENCES

[1] BELL, J. R. Threaded code. *Communications of the ACM 16*, 6 (1973), 370–372.

[2] BERNDL, M., VITALE, B., ZALESKI, M., AND BROWN, A. D. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the international symposium on Code generation and optimization (CGO'05)* (2005), IEEE Computer Society, pp. 15–26.

[3] BitTorrent. `http://www.bittorrent.com/`.

[4] CHAMBERS, C., UNGAR, D., AND LEE, E. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1989), ACM, pp. 49–70.

[5] CHOHAN, N. J., BUNCH, C., PANG, S., KRINTZ, C., SOMAN, N. M. S., AND WOLSKI, R. AppScale design and implementation. Tech. Rep. 2009-02, UCSB, Jan 2009.

[6] Cpython. `http://www.python.org/`.

[7] ERTL, M. A., AND GREGG, D. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism 5* (Nov. 2003). http://www.jilp.org/vol5/.

[8] FreeCAD. `http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Main_Page`.

[9] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), ACM, pp. 465–478.

[10] Google app engine. http://code.google.com/appengine/.

[11] HA, J., HAGHIGHAT, M. R., CONG, S., AND MCKINLEY, K. S. A concurrent trace-based just-in-time compiler for single-threaded JavaScript. In *Proceedings of the Second Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures* (Jun 2009).

[12] HOLKNER, A., AND HARLAND, J. Evaluating the dynamic behaviour of python applications. In *In Australasian Computer Science Conference. (ACSC'09)* (2009).

[13] Ironpython. `http://ironpython.codeplex.com/`.

[14] Jython. `http://www.jython.org/`.

[15] Lua Programming Language. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`.

[16] perfmon2: the hardware-based performance monitoring interface for linux. `http://perfmon2.sourceforge.net/`.

[17] PIUMARTA, I., AND RICCARDI, F. Optimizing direct threaded code by selective inlining. *SIGPLAN Not. 33*, 5 (1998), 291–300.

[18] Pybench — a python benchmark suite. `http://svn.python.org/projects/python/trunk/Tools/pybench/README`.

[19] PythonCAD. `http://sourceforge.net/projects/pythoncad/`.

[20] rietveld, code review for subversion, hosted on google app engine. `http://code.google.com/p/rietveld`.

[21] RIGO, A. Representation-based just-in-time specialization and the Psyco prototype for Python. In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation - PEPM'04* (Aug 2004).

[22] RIGO, A., AND PEDRONI, S. PyPy's approach to virtual machine construction. In *Proceedings of the Dynamic Languages Symposium* (Oct 2006).

[23] ROMER, T. H., LEE, D., VOELKER, G. M., WOLMAN, A., WONG, W. A., LOUP BAER, J., BERSHAD, B. N., AND LEVY, H. M. The structure and performance of interpreters. In *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII* (1996), ACM Press, pp. 150–159.

[24] Scientific python. http://wiki.python.org/moin/NumericAndScientific.

[25] Spambayes. `http://spambayes.sourceforge.net`.

[26] SquirrelFish Extreme JavaScript engine. http://webkit.org/blog/189/announcing-squirrelfish/, 2008.

[27] Stackless python. `http://www.stackless.com/`.

[28] Unladen-swallow project. `http://code.google.com/p/unladen-swallow/wiki/Benchmarks`.

[29] Google V8 JavaScript engine. http://code.google.com/p/v8/.

[30] Wiki matrix – programming language comparison. http://www.wikimatrix.org/statistic/Programming+Languages, Sept 2009.

[31] ZALESKI, M., STOODLEY, K., AND BROWN, A. D. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd ACM/USENIX international conference on Virtual execution environments* (New York, NY, USA, 2007), ACM Press, pp. 83–93.