IST FP6-004779

PYPY

Researching a Highly Flexible and Modular Language Platform and Implementing it by Leveraging the Open Source Python Language and Community

STREP

IST Priority 2

D06.1 Core Object Optimization Results

Due date of deliverable: March 31st, 2007

Actual Submission date: April 4th, 2007

Start date of Project: 1st December 2004 **Duration: 28 months**

Lead Contractor of this WP: HHU

Authors: Michael Hudson, Armin Rigo, Carl Friedrich Bolz

Revision: Interim

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level: PU (Public)

Revision History

Date	Name	Reason of Change
2006-12-21	Michael Hudson	Created initial structure, draft some sections
2007-01-15	Michael Hudson	More sections drafted
2007-02-27	Carl Friedrich Bolz	Restructured, more content
2007-03-01	Carl Friedrich Bolz	Describe multilists
2007-03-08	Michael Hudson	Describe multidicts
2007-03-12	Armin Rigo	More content
2007-03-13	Jacob Hallen	Internal Review
2007-04-02	Armin Rigo & Michael Hudson	Final content
2007-04-04	Carl Friedrich Bolz	Incorporate suggestions by Georg Brandl
2007-04-04	Carl Friedrich Bolz	Publish Interim version on web page

Abstract

The document describes the alternative designs and implementations of core data structures and algorithms that the flexibility of our Python interpreter allows us to consider. Most of these alternatives aim at improving the performance of our interpreter, and we present performance results, along with a description of how we obtained these results.

Purpose, Scope and Related Documents

The main purpose of this document is to describe the "core" optimizations made to PyPy's Standard Interpreter, i.e. optimizations made by editing the source code implementing the interpreter. In general it does not cover optimizations that are part of the translation process, although there is often a relationship between a rewrite of the interpreter and an optimization in the tool chain, and some of those are described herein.

However, because optimization is at its core such an empirical activity, we begin by describing the process by which we assessed the benefit of each attempt at optimization (including optimizations that affected the translation process).

Related documents:

- D04.2 Complete Python Implementation [D04.2]
- D07.1 Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects [D07.1]
- D13.1 Integration and Configuration [D13.1]

Contents

1 Executive Summary

PyPy's Python interpreter [D04.2] was designed to be as flexible as possible, and one of the motivations for this was to enable experimentation with alternative implementations of the core object types and algorithms.

An important goal of these alternatives is to improve performance. We have reimplemented or enhanced various core types and algorithms, including:

- Specialized versions of dictionaries.
- Several improvements to reduce the cost of a method call.
- Improvements to our multimethod implementation.

In addition, profiling identified a few performance deficiencies in the interpreter, which were often straightforward to resolve.

The impact of each alternative implementation was rigorously measured on a set of five benchmarks, which include both synthetic and real world code.

The effect of this work has been obvious: the speed of a compiled PyPy with all optimizations enabled is more than twice that of a binary with some optimizations disabled. As some optimizations, especially those implemented before the integration of our configuration framework [D13.1], were sufficiently clear wins that they were not made optional, the overall impact of this work was even higher than these measures show.

2 Assessing Optimizations

It is relatively easy to come up with an idea that *might* speed up the Standard Interpreter and rather less so to establish whether the implementation of this idea actually has any beneficial effect.

Our basic tool for assessing optimizations has been to run benchmarks. Finding suitable benchmarks is no trivial task. We settled on five benchmarks:

- **pystone**: the archetypal Python benchmark, but in no way typical Python code.
- **richards**: a Python translation of an object oriented benchmark that has been part of the language community folklore for years.
- **templess**: a simple templating toolkit, written by Guido Wesdorp.
- **gadfly**: a pure-Python SQL database.
- **mako**: another templating engine.

We chose these benchmarks because they exercise a range of parts of the Python interpreter, and can conveniently be arranged to run in a reasonable length of time for benchmarking, i.e. a time of a few seconds per run.

Pystone is the standard Python benchmark. It is a translation into Python of a C version of the Dhrystone benchmark originally written in Ada, and as such is very far from being typical Python code. However it runs quickly and provides a basic measure of the speed of the core interpreter loop.

Richards is another program written as a benchmark, this time from an original in BCPL written by Dr Martin Richards in 1981 at Cambridge University Computer Laboratory, England. It is an object oriented benchmark, and of all the benchmarks most tests the performance of method calls.

By contrast, the remaining benchmarks test the performance of more real-world code. "templess" is a templating system for HTML generation. It mostly stresses regular expressions and string handling. Gadfly is an SQL database written by Aaron Watters. When used with CPython, it is normal to use a C implementation of some

of the core algorithms, but we used the pure Python versions. It is an example of the implementation of some decidedly non-trivial algorithms, and it makes heavy use of long integer objects. mako is another templating library, and stresses Unicode handling most of all.

In addition, the last three benchmarks allocate much more memory than the first two, and so stress the garbage collector a lot more.

We also wrote a number of so-called "microbenchmarks", small snippets of code that exercise one particular part of the interpreter, allowing us to focus on one performance problem at a time.

We set up automated nightly runs of the pystone and richards benchmarks [BENCHHTML] on a variety of configurations of PyPy which allowed us to track our performance progress over time and quickly identify performance regressions.

2.1 Benchmarking Environment

All performance measurements in this document were performed on *wyvern*, a machine with four Intel Xeon 3.20 GHz cores[1], each with 2 MB of L2 cache, and a total of 5 GB of RAM, running a SuSE/Gentoo Linux combination.

It should be noted, though, that the benchmark page [BENCHHTML] referred to above reports results on a two-core 2.5 GHz G5 Power Macintosh with 3 GB of RAM running OS X 10.4.

3 Optimizing the Standard Object Space

The design of the Standard Object Space allows us to provide multiple implementations for each Python type, and then enable one or even several specific implementations in each translated PyPy. The particularity of the PyPy context is that in addition to allowing new implementations to be researched and developed, it gives a framework in which to *compare* previously existing object implementations and study their relative trade-offs – not only with micro- or small benchmarks, but with large real Python applications.

3.1 Multimethod Dispatch

The object implementation flexibility of the Standard Object Space, essential for the work described in the following sections, is based on multimethod dispatch [MM]. However, multimethods are not natively part of any of our current target platforms: they require custom dispatch code, which itself can be implemented in several ways, with various trade-offs. PyPy thus allows us to study the relative merits of various multimethod implementations with real-life applications even though the Python language itself has no built-in notion of multimethods.

At the beginning of the project multimethod dispatch was implemented by automatically generating chained double dispatching code. This turned out to be a size problem as the number of types increased: indeed, for N types, each doubly-dispatched multimethod consumes N entries in each of the N virtual method tables. This quadratic behavior eventually resulted in virtual method tables occupying more than 50% of the total size of the PyPy executable.

To overcome this we implemented the compact dispatch table scheme described by Pang et al. in [MRD]. It only requires a pair of integers per virtual method table, plus a few small compressed tables. No new research work was done in PyPy on multimethod dispatch, but we can contribute a comparison of the experimental results of the two approaches: compared to double-dispatch, Multiple Row Displacement [MRD] shrinks the executable of the PyPy interpreter by 2.5 MB (almost 50%). The run-time performance is very similar, only a few percents

[1] No benchmark uses more than one core.

slower. This is expected, as double-dispatch requires a double indirect jump, while Multiple Row Displacement requires about 4 arithmetic instructions followed by a single indirect jump. This is more instructions in total, but indirect jumps are particularly costly on modern processors, so the effects even out. In summary, double-dispatch is marginally faster than Multiple Row Displacement but requires a prohibitively large amount of static data, making the latter a better practical choice for most applications.

The situation is less clear-cut for a PyPy compiled via an Object-Oriented (OO) backend. Indirect method calls are particularly well optimized in most OO target platforms, at the expense of indirect function calls (which often require the use of platform-specific features that are not really natural in an OO context, like delegates in the Common Language Infrastructure). Double-dispatch only costs two method calls, while Multiple Row Displacement requires arithmetic and a couple of table lookups followed by such an indirect function call. We measured a 15% performance impact of Multiple Row Displacement over double-dispatch in `pypy-cli` running on Mono. Because of this result, we chose to continue using double-dispatch for OO backends by default. In future work we plan to search for and compare with other multimethod implementations offering a better trade-off in OO environments.

3.2 Integer boxing

The standard approach to implement integers in Python is to consider them as a regular boxed type.

A simple optimization is to remark that the majority of integer objects created during the lifetime of a program are very small in magnitude. To relieve the GC pressure, we can prebuild all integer boxes between, say, -5 and 100, and systematically reuse them.

The following table shows the mixed results of this approach, for two ranges of prebuilt integer objects. The baseline pypy-c is normalized to "1", and lower numbers are better:

prebuilt integers:	**between -5 and 99**	**between -5 and 256**
richards	0.98	1.06
pystone	0.98	1.01
templess	1.00	1.05

We speculate that the table of prebuilt integers has bad cache effects: a freshly built integer object is entirely in the CPU cache, while simply taking a reference to a prebuilt object might require extra memory loads. We only obtained the marginal improvements seen in column 1 after we added a hack that has the effect of prefetching the `intval` field of the prebuilt integers into the CPU caches.

In cooperation with WP07, we also implemented *pointer tagging*. This is a common technique in virtual machine implementations. Uncharacteristically, we did not implement it by pervasive source code changes, but by minimal support in the translation tool-chain. We refer the reader to [D07.1] page 35 for more information.

3.3 Special string implementations

The basic implementation for string objects is with an RPython string, which is a simple array of characters, stored contiguously in memory. This is not the best representation when the strings manipulated by the Python program become large and are operated upon in non-trivial ways. We have experimented with the following three alternate representations:

- **String join objects:** can be used as the result of string concatenation (via + or `join()`). Internally stores a list of the strings it is made of. This avoids copying the characters when concatenating large strings.
- **String slice objects:** for large sub-slices of large strings. Stores a reference to the original string and the start and end position. As above, this avoids a copy.

- **Ropes:** the general flexible string representation format proposed by Boehm et al. in [ROPES].

We compared versions of pypy-c that all include multidicts (described below): they tend to be less sensitive to the overhead of the advanced string implementations because they represent string-keyed dictionaries in a different way. On all five benchmarks (richards, pystone, templess, gadfly, mako) the results are between 3% faster and 9% slower, with no clear tendency.

We interpret these results as follows: Each of these implementations has clear theoretical benefits, to the point of improving the algorithmic complexity of some Python code examples, sometimes drastically. Synthetic benchmarks can be devised that show arbitrary improvements. However, existing Python applications are all written in a style that expects strings to be a flat array of characters. This is why so benchmarking these applications shows mostly no benefit (instead, small slow-downs are common, caused by the additional overhead). An application targeting PyPy only could be written to make use of the improved string representations; for example, managing the buffer of a text editor is much simpler if ropes-like performance can be assumed.

Another potential advantage is that ropes often perform better for naive algorithms, which means that code can stay readable and new programmers have a greater chance of writing efficient code.

3.4 Dictionary Implementations – Multidicts

It is a well known fact in the Python community that the performance of the dictionary implementation affects the performance of the interpreter more than that of any other object, as dictionaries are used internally by many parts of the interpreter. This inspires two approaches to optimizing the interpreter:

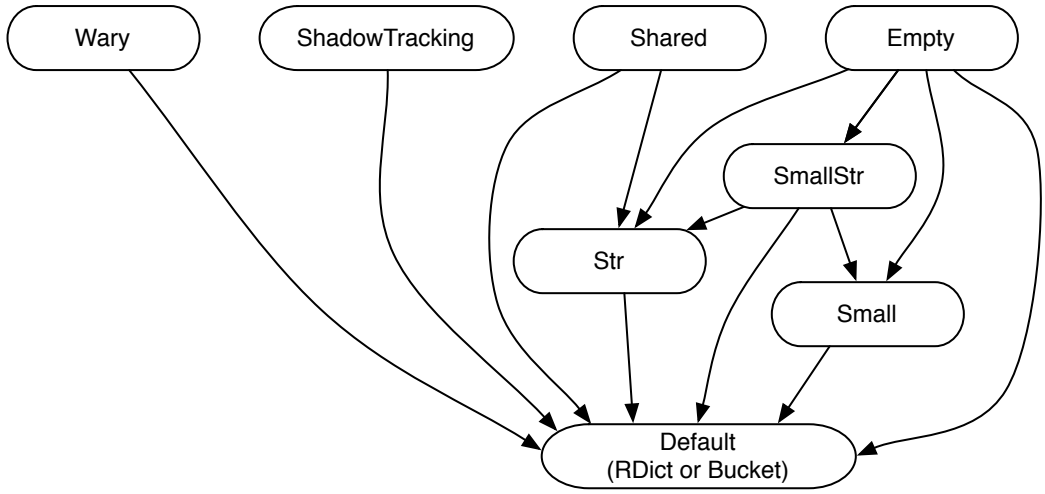
- Reducing the core interpreter's dependence on dictionary lookups, the subject of several later sections.
- Improving the performance of dictionaries themselves, by optimizing for common cases and experimenting with alternative algorithms, which is the subject of this section.

The fact that, unlike strings and integers, dictionaries are mutable makes optimizing them for specific usage patterns harder. Even if a specific way of using a dictionary is detected and an optimized dictionary implementation chosen, it is still possible that the dictionary is used in radically different ways later in its lifetime. Therefore the way a dictionary is implemented should be changeable over its lifetime. To achieve this, we implemented *multidicts*: a dictionary is split into two objects, a thin wrapper that constitutes the identity of the dictionary and a dictionary *implementation* that is responsible for the dictionary's behaviour. All dictionary operations and methods delegate to methods of the implementation object. The methods of the implementation object that actually *mutate* the dictionary (like setting an item) can choose to change the implementation of the dictionary by creating and returning a new implementation. This new implementation represents the same dictionary but potentially in a different way.

This splitting into thin wrapper and actual implementation also has the added benefit that adding new dictionary implementations is eased, because the interface an implementation needs to implement is more minimal than the rich set of methods and operations the Python dictionary type provides.

It should be noted that the implementation object is responsible for managing the storage associated with the dictionary. While it would be possible for implementations to share storage, this would be hard to arrange in the strongly typed RPython language, and empirical observation suggests that changing dictionary implementations is an exceedingly rare occurrence (apart from the initial change away from the `EmptyDictImplementation`).

At the time of writing, there are ten dictionary implementations present in PyPy, although only a subset will be used in any given translated binary. Nine of these implementations are shown in the following diagram:



The arrows depict the possible transitions between the implementations. Additionally, invoking the `clear()` method on any dictionary will reset the implementation to the `EmptyDictImplementation` singleton. The implementation not shown above is `MeasuringDictImplementation`.

3.4.1 `EmptyDictImplementation`

Being the simplest, and indeed almost trivial, dictionary implementation, the empty dictionary always fails a lookup. The only non-trivial part is deciding which implementation to switch to when a key is inserted into the dictionary. By default, this too is very simple: if the key is a string, the `StrDictImplementation` is chosen, if not then an `RDictImplementation` is used. As alluded to above, observations indicate that this is the only implementation that will be used in the lifetime of the dictionary in an overwhelming majority of cases.

Because the `EmptyDictImplementation` holds no state, only one instance ever needs to be constructed, which is stored on the object space object. This means that allocating a fresh multidict is a relatively cheap operation.

3.4.2 `StrDictImplementation`

The simplest and most effective specialization is for the case of all the dictionary keys being strings, as module, instance and class dictionaries all meet this criterion in the common case.

`StrDictImplementation` is a straightforward implementation of this idea, although about half of the dictionary implementations listed above are also specialized for this case.

Using `StrDictImplementation` gives a performance improvement of between 15% and 40%, with the bulk of the gain likely coming from not having to perform any kind of method dispatch to check equality of keys.

3.4.3 `SmallStrDictImplementation` and `SmallDictImplementation`

`SmallStrDictImplementation` and `SmallDictImplementation` were inspired by a simple observation in Chapter 6 of [KNUTH] that in a table of just a few items it is hard to beat sequential search, even in terms of just operation count. As sequential access has better locality of reference, this effect is accentuated by the expense of a cache miss on modern processors.

However, implementing small dicts (and the string-keyed variant) had almost no effect on performance. Further analysis revealed the likely cause: most dictionary lookups are to a fairly small number of fairly large dictionaries

– particularly module dictionaries and the builtins. A build using small dicts may be of some use if there are a lot of instances with just a few member variables, but there are other optimizations giving better results for this purpose, like the `SharedDictImplementation`.

3.4.4 `SharedDictImplementation`

The `SharedDictImplementation` class aims to save memory for the dictionaries of class instances. Python instances usually contain a dictionary, which for many classes with a small and fixed set of instance attributes is a waste of memory: All the instance dictionaries share the same set of keys but each of them stores these keys in addition to just the values (which are the only actual difference between the various instances). The Python language provides the `__slots__` mechanism (which fixes the allowed attributes a class can have and which makes instances not use a dictionary at all) to fix this problem. The disadvantage of this solution is that it has to be used explicitly and that it decreases flexibility – typical Python programs occasionally attach extra unforeseen attributes to existing instances. Also, it does not play well with subclassing.

The `SharedDictImplementation` is another strategy to solve this problem, transparently to the programmer. A dictionary is split into two parts, a part that stores just the keys of the dict and a part that stores the actual values. The former part is shared by all instances that actually use the same set of attributes; only the latter part is separate. If the dictionary of an instance has very different keys than other instances, a `StrDictImplementation` is used instead. The resulting memory behaviour is quite similar to that of using `__slots__` but any code gets the memory benefit, not just code where `__slots__` were explicitly added.

To measure the impact of this strategy, we ran a simple program that simply created 1000000 instances, each with just two identically named attributes. The final memory usage was read in the output of `top`. For comparison purposes, we also changed the program to use `__slots__` explicitly:

	no slots	**slots**
normal pypy-c	266860 KB	96640 KB
multidict	199592 KB	89036 KB
sharing dict	140136 KB	89036 KB
CPython	202548 KB	60328 KB

3.4.5 `RDictImplementation` and `BucketDictImplementation`

The `RDictImplementation` and `BucketDictImplementation` classes are the most general classes – they can handle any kind of key type and be of any size.

The `RDictImplementation` class uses `rdicts`, an RPython feature providing a dictionary that uses application-supplied equality and hashing functions. The low-level backends use an implementation patterned after CPython's open addressing scheme – itself based on Algorithm D from [KNUTH], Sec. 6.4 – while the high-level backends generally use an implementation provided by the target platform.

`BucketDictImplementation` provides a backend-independent implementation that implements hash collision resolution by chaining (as in Algorithm C from the same section of [KNUTH]). It is somewhat slower than `rdicts` for low-level backends, but provides a degree of independence from the target platform for high-level backends.

3.4.6 `ShadowDetectingDictImplementation` and `WaryDictImplementation`

The `ShadowDetectingDictImplementation` and `WaryDictImplementation` classes are implementation details of two other optimizations, CALL_LIKELY_BUILTIN and Method caching, and are described in the sections devoted to those optimizations.

3.4.7 `MeasuringDictImplementation`

The `MeasuringDictImplementation` class exists only to gather statistics about dictionary usage patterns. When this option is compiled in, a file containing such information is created at process exit. We used the statistical package `R` to analyze this data and drive our optimization work.

3.4.8 Summary

The basic `StrDictImplementation` shows the largest benefits in PyPy. This is related to CPython's optimization of the lookup logic for dictionaries that are known to contain only strings, and it proves again that dictionaries play a central role in Python, performance-wise.

There are many more dictionary-related optimizations that could be experimented with. Even more interesting, however, is the fact that many other optimizations – described in the sequel – would have been difficult or impossible without custom multidict implementations. The multidict approach thus brought an extra level of flexibility to the Standard Object Space for experimentation and optimizations purposes.

3.5 Lists Implementations – Multilists

Lists are another mutable core Python type, and to experiment with different implementations of lists we implemented *multilists*, similar in design and implementation to multidicts.

We experimented with various implementations including:

- **range lists:** for the result of a call to `range(start, stop, step)`, which is commonly used in `for` loop. Only stores the bounds and computes the elements lazily as needed.
- **list slices:** same idea as for string slices (see above), with the additional complication that if the base list is mutated, the sliced sublist must in fact be copied.
- **chunk lists,** allocating an array of pointers to fixed-size chunks (the default implementation uses a single over-allocated array; both implementations provide amortized constant-time appends but the chunk lists try to reduce the amount of copying needed).
- **optimal resizable arrays:** a data structure introduced by Brodnik et al. in [RESIZABLE]. It still provides amortized constant-time appends, but with a theoretically optimal space usage: it over-allocates lists of n elements by $O(\sqrt{n})$ bytes instead of the traditional $O(n)$.

The benchmarks showed little noticeable effects: between 3% speed-ups and 11% slow-down (with an exception for the optimal resizable arrays, which can be as bad as 4 times slower – the algorithm is optimal in a theoretical sense but its constants are huge). These results can be explained with the same arguments as for our string implementations (see above): although some algorithms could be written in Python in a simpler way by assuming that the optimized list implementations are available, in practice existing applications assume the baseline CPython behavior. The benchmarks thus mostly report the extra overhead of multilists. For example, `for` loops generally use the `xrange()` built-in function instead of `range()` whenever it makes a difference. The latter returns a special Python object of a different type that is only suitable in this context; it is an example of how the design of CPython seeks a balance between language simplicity and ease of implementation. In a `pypy-c` with range lists, `xrange` would not be necessary.

3.6 CALL_LIKELY_BUILTIN

An often heard "tip" for speeding up Python programs is to give an often used builtin a local name, since local lookups are faster than lookups of builtins, which involve doing two dictionary lookups: one in the globals dictionary and one in the builtins dictionary. PyPy approaches this problem at the implementation level, with the introduction of the new `CALL_LIKELY_BUILTIN` bytecode. This bytecode is produced by the compiler for a call whose target is the name of a builtin. Since such a syntactic construct is very often actually invoking the expected builtin at run-time, this information can be used to make the call to the builtin directly, without going through any dictionary lookup.

However, it can occur that the name is shadowed by a global name from the current module. To catch this case, a special dictionary implementation for multidicts is introduced, which is used for the dictionaries of modules. This implementation keeps track which builtin names are shadowed by it. The `CALL_LIKELY_BUILTIN` bytecode asks the dictionary whether it is shadowing the builtin that is about to be called and asks the dictionary of `__builtin__` whether the original builtin was changed. These two checks are cheaper than full lookups. In the common case, neither of these cases is true, so the builtin can be invoked directly.

This optimization gives good speed-ups, as shown below ("1" is a baseline pypy-c, lower is better):

	with CALL_LIKELY_BUILTIN
richards	0.81
pystone	0.85
templess	0.85
gadfly	0.89
mako	0.89

3.7 Method optimizations

Method calls are another very common (and costly) operation in Python programs, which makes them a good optimization target. The most expensive part of method calls (especially for deep class hierarchies) is the method lookup, which is the part that PyPy attempts to speed up.

3.7.1 Shadow Tracking

The first step in speeding up method lookups is to check whether an instance attribute shadows an attribute with the same name in its class. This is achieved by introducing a new dictionary implementation for multidicts, `ShadowDetectingDictImplementation`. Whenever an attribute is set on an instance, there is a check on its class whether this attribute is on the class already (this check is performed anyway, to find data descriptors). If it is already there, a flag is set on the dictionary implementation.

This information alone enables a small optimization for method lookups: Whenever an attribute lookup on an instance is performed in Python, the following steps are done: First there is a lookup for that attribute on the class, to check for data descriptors. Then there is a lookup on the instance's dictionary to find actual attributes. This second lookup can be saved if the first one succeeds and the instance is known to not shadow a class attribute, since it can never succeed. This saves one dictionary lookup per method call for the most common case where an instance does indeed not shadow its class.

In our measures, shadow tracking gave speed-ups over a baseline pypy-c ranging from minor to very interesting: templess is 4% faster, pystone is 14% faster, and richards is 21% faster. These results can probably be explained by the fact that richards, in a typical OO style, never puts any attribute on its instances that shadows a class attribute. In a typical Python application, though, this is likely to occur at least occasionally. So far, the

optimization is entirely disabled for all attribute lookups on such instances; further research could improve on this.

3.7.2 Method caching

The most expensive part of a method lookup is looking at the *method resolution order* (MRO) of a class to find the method, which involves one dictionary lookup per class. PyPy solves this with a global *method cache* that caches these lookups. The cache is a fixed-size hash table mapping pairs `(version_tag, name)` to the result of the lookup, where `name` is the name being looked up and `version_tag` represents the type where the lookup happens.

More precisely, the `version_tag` is a tiny object attached to the class. If the dictionary of the class itself is changed, it gets a new version tag, and so do all its subclasses (since the change can be seen from these classes as well). This ensures that the method cache never returns outdated entries.

Note that despite the name, the method cache actually caches the results of any lookup, independently of whether it returned a method or another kind of class attribute.

This scheme can be compared to polymorphic inline caches [PIC], with the difference that we have a single global cache instead of one per call site. A fixed-size global cache consumes less memory, so it might have a memory locality advantage while retaining the property that the cache is unlikely to miss often during the execution of tight loops. Yet another scheme would be "flattening": for each class `C`, we could collect all names from all superclasses of `C` into a single dictionary attached to `C`, with the appropriate updating logic when superclasses are modified. This could give good performance but have a potentially large, non-linear memory impact in the presence of deep hierarchies with many names.

To find a good default cache size, we ran the following benchmarks (lower numbers are better, "1" is a baseline pypy-c with multidicts):

cache size	richards	pystone	templess
32	0.97	0.98	0.99
64	0.89	0.97	1.00
128	0.83	0.93	0.97
256	0.85	0.97	0.98
512	0.80	0.95	0.94
1024	0.77	0.95	0.95
2048	0.79	0.95	0.95
4096	0.79	0.94	0.93
8192	0.82	0.95	0.94

The richards benchmark benefits the most (as expected). A cache of 128 entries appears to already give good results; 1024 entries give the best result. The templess case shows that 128 entries are not enough in real-life applications, but 512 appears to be.

In future work we plan to improve the resilience against hash collisions, for example by implementing a 2-way or 4-may mapping (as is common in hardware caches), or with probabilistic schemes like arbitrarily changing the `version_tag` of a type after too many cache misses.

3.7.3 `CALL_METHOD`

Although the Python syntax for method calls looks familiar, its precise semantics are decomposed in a slightly unusual way: `obj.meth(x, y)` is equivalent to `(obj.meth)(x, y)`. The corresponding bytecode is:

```
LOAD_GLOBAL      obj       # push 'obj' on the stack
LOAD_ATTR        meth      # read the 'meth' attribute out of 'obj'
LOAD_GLOBAL      x         # push 'x' on the stack
LOAD_GLOBAL      y         # push 'y' on the stack
CALL_FUNCTION    2         # call the 'obj.meth' object with arguments x, y
```

Each method call actually instantiates a so-called *bound method object* (at the `LOAD_ATTR`) which is often just used as the target of a call (performed by `CALL_FUNCTION`) and then becomes garbage. In CPython, trying to avoid the temporary object creation is a recurring discussion topic; so far, several patches [py709744] have been proposed but not integrated, generally because they were considered to not give an interesting balance between performance gained and complexity of implementation.[2]

We made an attempt based on the idea of keeping method lookup separated from method call, but using the value stack as a cache instead of building a temporary object. We extended the bytecode compiler to (optionally) generate the following code for `obj.meth(x, y)`:

```
LOAD_GLOBAL      obj
LOOKUP_METHOD    meth
LOAD_GLOBAL      x
LOAD_GLOBAL      y
CALL_METHOD      2
```

`LOOKUP_METHOD` contains exactly the same attribute lookup logic as `LOAD_ATTR` – thus fully preserving semantics – but pushes two values on the stack instead of one. These two values are an "inlined" version of the bound method object: the *im_func* and *im_self*, i.e. respectively the underlying Python function object and a reference to `obj`. This is only possible when the attribute actually refers to a function object from the class; when this is not the case, `LOOKUP_METHOD` still pushes two values, but one *(im_func)* is simply the regular result that `LOAD_ATTR` would have returned, and the other *(im_self)* is a None placeholder.

After pushing the arguments, the layout of the stack in the above example is as follows (the stack grows upwards):

`y` *(2nd arg)*
`x` *(1st arg)*
`obj` *(im_self)*
`function object` *(im_func)*

The `CALL_METHOD N` bytecode emulates a bound method call by inspecting the *im_self* entry in the stack below the `N` arguments: if it is not None, then it is considered to be an additional first argument in the call to the *im_func* object from the stack.

A few variants of this scheme have been tried, but the above one is the first that fully preserves the semantics and gives major speed-ups in PyPy. The results are particularly good when combined with other optimizations, most notably method caching (to cache the method lookup itself, of course, but also to cache the absence of a custom `__getattribute__` method which would make the optimization invalid) and shadow tracking (to cache the absence of an instance-level attribute of the same name).

In all measures, the `LOOKUP_METHOD` and `CALL_METHOD` bytecodes are either a net win or have no effect. They give a speed-up of about 6% over a baseline pypy-c on all benchmarks except pystone, on which they have no effect. When added to a pypy-c with method cache enabled, they improve templess by 5%, but have no effect on pystone, and (somewhat surprisingly) neither on richards.

[2] A common attempt is based on the idea of pushing on the value stack the target object, then the arguments, and then using a special bytecode encoding both the method name and the arguments count. However, this is messy to implement and introduces a minor change in semantics – the method would be looked up on the object *after* the arguments are evaluated, instead of before.

3.7.4 Combined results

The three method optimizations described above are orthogonal, being targeted at different aspects of the attribute lookup semantics. By enabling all three of them in a single pypy-c, we get the best results by far, surpassing even what could have been expected from the individual speed-ups that each optimization provides on its own – up to 47% of speed-up. As measured on wyvern ("1" is the pypy-c baseline including multidicts[3], lower is better):

	shadow tracking	**method cache**	**CALL_METHOD**	**combined**
richards	0.79	0.77	0.94	0.53
pystone	0.86	0.95	1.01	0.78
templess	0.96	0.95	0.94	0.75

3.7.5 Precomputed lookups on built-in types

Python specifically forbids the user from mutating built-in types (though this is only a design decision, not a fundamental limitation). In CPython, built-in types are based on a table of C function pointers, which makes looking up internal methods particularly fast. Internal methods are methods like `__add__` and `__getitem__`; almost every Python operation needs to look up one or more of them.

In PyPy, the source code uses the same interface for both internal and regular lookups, i.e. `space.lookup(w_obj, name)`, which looks up the provided `name` in the Python type of `w_obj`. Instead of breaking this uniformity in the source code, we implemented the following idea: at translation time, our type inference process notices which calls provide a constant `name` and collects these names. It also collects all the built-in Python types that it sees. Eventually, for each name `__xyz__` and each built-in Python type, it pre-computes the result of the lookup and caches it on the W_TypeObject itself, under a special attribute called `cache__xyz__`. The calls `space.lookup(w_obj, "__xyz__")` can then all be replaced by optimized logic that directly returns the value of `cache__xyz__` when `w_obj` has a built-in Python type.

We have included this optimization in all the above measures (it is included in what we called the baseline pypy-c). The reason is that it is an obvious big win with no trade-off: specifically disabling it incurs a slow-down of 22% to 32% on all benchmarks. We also measured a pypy-c with precomputed lookups disabled but method cache enabled: the latter is essentially a more general version of the former, but its overhead is larger, so it gives less clear-cut results: (lower numbers are better, "1" was the baseline in the measures of previous sections)

	none	**precomputed lookups only**	**method cache only**	**both**
richards	1.32	1	0.88	0.79
pystone	1.22	1	1.04	0.95
templess	1.32	1	1.23	0.95

[3] Multidicts give a speed-up of their own and are required for shadow tracking, so they were included in all versions of pypy-c presented in this table.

4 Standard Interpreter Optimizations

4.1 The Dispatch Loop

The bytecode dispatch of PyPy's bytecode interpreter [D04.2] was originally implemented as a list of function pointers, into which the dispatch would index using the next bytecode's value. Each iteration through the bytecode dispatch loop would thus contain a call to one of the functions from this list. This has some drawbacks, the biggest being that none of the functions that implement a bytecode can be inlined into the dispatch loop – not even the simplest ones – because the call to the function is an indirect one using a function pointer. Some examples for simple bytecodes are NOP (no operation), POP_TOP (removes the top element of the stack), DUP_TOP (pushes the topmost element of the stack again onto the stack) and many of the other stack-manipulating bytecodes.

In CPython, the reference implementation of Python, the dispatch loop is implemented using a switch statement with one case for every opcode. Since Python does not support a switch statement we could not do exactly that. Instead we rewrote the bytecode dispatch in a form that is essentially equivalent to a long chain of blocks of the following form:

```
if opcode == ...:
    appropriate_opcode_implementation()
```

Without further changes this would have been much slower – a linear search. This approach is only viable in combination with another translation-time optimization, which finds chains of comparisons of one fixed value with many different constants and turns that into a switch statement in the flow graph. The backends can then render it as a switch.

This optimization gives mixed results, for reasons that we can only guess – we speculate that the "switch" version of the dispatch function becomes very large after inlining, which negatively impacts locality. The results are still positive on average, as shown below ("1" is a pypy-c containing all worthwhile optimizations described in this document except this one, and lower is better):

	dispatch loop using switch
richards	0.95
pystone	1.00
templess	1.05
gadfly	0.91
mako	1.01

4.2 Argument passing in function calls

During the course of the project, profiling has repeatedly shown that application-level function calls are a bottleneck. This insight is based on microbenchmarks that compare the relative speeds of various operations: the call-related ones are systematically near the end of the list.

We have thus tweaked the related parts of the interpreter source code in various ways. A major overhead of function calls is that we are using an `Arguments` class that encapsulates the shape and value of the actual parameters at the call site, and contains the complicated logic required by Python to match them to the formal arguments of the function. This means that an instance of `Arguments` had to be built for each call; this created significant memory pressure. Over time, we tried to avoid having to build an `Arguments` object in the common cases:

- Call sites with only positional arguments – no keywords, no $\star$ and no $\star\star$ – invoke a different interface on the object space, `call_from_valuestack()`. This interface breaks encapsulation, trading it for

performance: the interpreter frame object is passed to the object space, which reads the arguments directly off the value stack of the frame, without any intermediate data structure to store them in.

- Fast paths for calls with a small number of arguments – to completely avoid the argument decoding logic, calls that only have a small number of positional arguments (up to 4) go through shortcuts at a few levels of the call logic.
- Builtin function calls have been streamlined, by removing an intermediate abstraction that introduced too much overhead.

It is difficult to provide more precise measures of each particular optimization, as they have become part of the core of the interpreter and cannot be easily reverted or temporarily disabled. Various call-related benchmarks show pypy-c to be currently (as of March 2007) between 2.65 and 3.44 times slower than on CPython; one year ago (March 2006) it was between 5.03 and 6.56.

4.3 Bytecodes special-casing common types

An idea taken from CPython: some bytecodes are expected to be often used with a given type. For example, addition is often between two integers, and indexing is often performed on a list with an integer index. We can thus write the corresponding bytecode implementations in such a way that they first check for the common type, and only use the general dispatch mechanism in other cases.

However, these optimizations implemented in PyPy did not give conclusive speed-ups, as shown below ("1" is a pypy-c containing all worthwhile optimizations described in the rest of this document, and lower is better). This might show that the normal dispatching mechanisms (which use multimethod dispatch) are well-optimized already.

	`integer + integer`	`list[integer]`	**both**
richards	1.01	0.98	0.99
pystone	0.96	0.99	0.96
templess	0.98	1.01	1.00
gadfly	0.99	0.99	1.00
mako	0.99	0.99	1.02

4.4 Miscellaneous optimizations

In the course of the project a number of smaller optimizations have been implemented. We will now quickly describe some of them without reporting benchmark results – such numbers are delicate to obtain now, because over time the changes had pervasive effects over the whole source code base. They are difficult to disable or revert for comparison purposes. These changes are:

- *String interning*: a classical idea present in most virtual machines is to distinguish between two classes of string-like objects: regular strings and "atoms", also called "symbols". The latter are typically used for program identifiers. The environment maintains a table of all atoms in use in the program and uses it to enforce the uniqueness of atoms – for each string, at most one atom object may exist.

  Python has no separate notions of strings and atoms, but CPython can selectively "intern" strings in such a table. The result is that comparison between two strings is reduced to a pointer comparison if both strings are marked as interned.

  In PyPy, we implemented a similar scheme for the application-level string objects, as follows: a dictionary maps selected interpreter-level strings to a unique `W_StringObject` instance containing it. It gives the

same benefit as it does to CPython: uniqueness of the application-level string objects used in places like the namespace of modules. It also helps to reduce memory pressure by avoiding the allocation of a new `W_StringObject` instance if one already exists.

- A related optimization was the change of some interfaces that used to accept an interpreter-level string and now accept an application-level string. Performing a `setattr` on an object is a typical example. The name of the attribute being set is originally an application-level string, and in many cases the result of the `setattr` is to set a key/value pair in the namespace dictionary of the object. The key itself is again an application-level string. However, the Python language specifies that only a string can be used as a name in this situation, so that some parts of the `setattr` logic used to accept and manipulate interpreter-level strings, which appeared to be more natural. The result, though, was that a new `W_StringObject` had to be created to be used as the key in the namespace dictionary. We now pass the original `W_StringObject` all the way through.

- Avoiding exceptions: we discovered that exceptions are costly – likely because they have to be instantiated, so contribute to memory pressure. Subsequently, we introduced interfaces for a couple of performance-critical operations that return `None` instead of raising an exception. Examples are the `getdictvalue()` method of application-level objects and the `finditem()` method of object spaces, which return `None` instead of raising, respectively, application-level `AttributeError` and `KeyError` exceptions.

5 Conclusion

The expected goal of this work package was stated as "Measurably better performance for non-trivial programs." As the tables in the above sections should have made clear, this has certainly been achieved.

To summarize our results, here is a table comparing the performance of PyPy on our five standard benchmarks, with and without those optimizations that are generally useful and can easily be selectively enabled and disabled:

	optimizations off	**optimizations on**
richards	1.00	0.40
pystone	1.00	0.56
templess	1.00	0.65
gadfly	1.00	0.71
mako	1.00	0.62

Our performance relative to CPython appears to depend very strongly on the amount of memory being allocated, as the following table of performance indicates (the pypy-llvm column represents the fastest PyPy available: compiled with the fastest backend, and with all optimizations enabled):

	CPython 2.4.4	**pypy-llvm**
richards	1.00	1.17
pystone	1.00	1.55
templess	1.00	5.41
gadfly	1.00	6.38
mako	1.00	7.65

While PyPy's string and Unicode implementations could probably be improved to help these numbers somewhat, the biggest single improvement would likely be from the addition of a more sophisticated garbage collector, something that is out of scope for this work package.

The most significant new optimizations described in this report are the method optimizations, and this can be seen in the fact that the optimizations have the greatest effect on the richards benchmark. The CALL_METHOD and method cache optimizations could be ported to CPython relatively easily – and indeed the method cache already has been [py1685986], where it gives speedups similar to those seen with pypy-c. More generally, while not all of the new optimizations that we experimented with are easy to port to all Python implementations, they certainly give strong hints about the direction in which further efforts could be valuable for them as well.

Within the scope of this work package, the potential for future work is only restricted by imagination. There have been many, many optimization ideas mentioned in the Python community over the years, most of which were never implemented. While a good number of these have now been tried out in PyPy, as described in this document, many more remain. In addition, the flexibility of PyPy's implementation, for example the ability to have a custom dictionary implementation for a given situation, allows for ideas that would simply not be practical for CPython.

The flexibility of PyPy's Python implementation has been essential to the task of experimenting with and especially assessing these new implementations, vindicating the effort we put in to allow this flexibility.

6 Glossary of Abbreviations

The following abbreviations may be used within this document:

6.1 Technical Abbreviations:

AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree
CPython	The standard Python interpreter written in C. Generally known as "Python". Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
CCLP	Concurrent Constraint Logic Programming.
CPS	Continuation-Passing Style.
CSP	Constraint Satisfaction Problem.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime.
docutils	The Python documentation utilities.
F/OSS	Free and Open Source Software
GC	Garbage collector.
GenC backend	The backend for the PyPy translation toolsuite that generates C code.
GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
GenCLI backend	The backend for the PyPy translation toolsuite that generates CLI code.
Graphviz	Graph visualisation software from AT&T.
IL	Intermediate Language: the native assembler-level language of the CLI virtual machine.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.

Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.
Pygame	A Python extension library that wraps the Simple DirectMedia Layer - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
pypy-c	The PyPy Standard Interpreter, translated to C and then compiled to a binary executable program
ReST	reStructuredText, the plaintext markup system used by docutils.
RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.
VM	Virtual Machine.

6.2 Partner Acronyms:

DFKI	Deutsches Forschungszentrum für künstliche Intelligenz
HHU	Heinrich Heine Universität Düsseldorf
Strakt	AB Strakt
Logilab	Logilab
CM	Change Maker
mer	merlinux GmbH
tis	Tismerysoft GmbH
Impara	Impara GmbH

References

[BENCHHTML] http://tuatara.cs.uni-duesseldorf.de/benchmark.html

[D04.2] *Complete Python Implementation*, PyPy EU-Report, 2005

[D07.1] *Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects*, PyPy EU-Report, 2007

[D13.1] *Integration and Configuration*, PyPy EU-Report, 2007

[KNUTH] Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3 / Sorting and Searching*, Addison Wesley

[MM] "Multimethods" in "PyPy - Object Spaces": http://codespeak.net/pypy/dist/doc/objspace.html#multimethods

[MRD] Pang C., Holst W., Leontiev Y., Szafron D. 1999. *Multi-Method Dispatch Using Multiple Row Displacement*, Lecture Notes in Computer Science Vol. 1628

[PIC] Hölzle U., Chambers C., Ungar D. 1991. *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*, ECOOP'91 Conference Proceedings, Geneva, 1991.

[py1685986] *CPython Patch 1685986: Method cache*, http://www.python.org/sf/1685986

[py709744] *CPython Patch 709744: CALL_ATTR opcode*, http://www.python.org/sf/709744

[RESIZABLE] Brodnik, A., Carlsson, S., Demaine, E.D., Munro, J.I., Sedgewick, R. 1999. *Resizable Arrays in Optimal Time and Space. Workshop on Algorithms and Data Structures*, 37-48.

[ROPES] Boehm, H.-J., Atkinson, R., Plass, M. 1995. *Ropes: An alternative to Strings. Software - Practice & Experience*, Volume 25, Issue 12, 1315-1330. John Wiley & Sons, Inc., New York, NY, 216-227.