IST FP6-004779

PYPY

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

STREP

IST Priority 2

Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects

Due date of deliverable: June 2006

Actual Submission date: Tismerysoft

Start date of Project: 1st December 2005 **Duration: 2 years**

Lead Contractor of this WP: XXX insert submission date here l

Authors: Carl Friedrich Bolz, Armin Rigo

Revision: draft, 24th July 2006

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
2006-06-07	Carl Friedrich Bolz	first draft, describing optimizations
2006-06-27	Carl Friedrich Bolz	draft, added description of garbage collectors
2006-07-14	Carl Friedrich Bolz	draft, added performance figures
2006-07-23	Armin Rigo	draft, improved report, added executive summary
2006-07-25	Carl Friedrich Bolz	publish draft on web page

Abstract

This report describes the implementation of stackless features in the PyPy project, how optimizations are implemented in PyPy's translation toolchain and how PyPy handles garbage collection. We describe the Stackless features, which allow a user-program to use arbitrarily deep recursion and to take control on its own activation records; this can be used for coroutines and green threading. We also show how various optimizations are used together to achieve much higher performance. Finally, we describe how our various garbage collectors are weaved into the program during translation. A key point is that our Python interpreter implementation is not changed by any of these aspects, which provides new flexibility previously not found in practical VM and language implementations. To evaluate the optimizations and the garbage collectors a set of benchmarks results is presented, showing that the implemented optimizations give sizeable speedups and that the various garbage collector give reasonable performance results.

Contents

1 Executive Summary

PyPy's tool-chain is able to translate a subset of the Python language, RPython, by performing whole-program type inference and then specializing to other language backends. In particular, we translate in this way our full Python implementation, which is a "high-level description" of the Python language expressed without explicit reference to lower-level issues like memory management.

All the relevant lower-level issues are resolved during translation: the translation process processes the forest of high-level control flow graphs that represents the input RPython program, and turns them into successively lower-level control flow graphs in which more and more lower-level details are made explicit. The translation process proceeds in this way until the graph's concreteness level matches the target platform's expectations. The flexibility obtained in this way allows us to postpone to translation time decisions about which solution to use for each low-level issue (e.g. which garbage collection strategy). It is also inheritently retargettable, even to platforms with highly different requirements (e.g. C or assembler-level environments versus object-oriented garbage-collected VMs).

The present report describes in detail some of the transformations that we wrote and tried. We had a C-like target environment in mind as our primary goal so far (it is in some sense the most demanding environment, being the lowest-level one). These issues are:

- Stackless Features:

  Corresponding to Task 1 of WP07, so-called "Stackless" support is based on a graph transformation that inputs graphs that contain recursive calls to each other - just like the source RPython program - and breaks these resursive cycles. This is done by turning the graphs into a variant of explicit Continuation-Passing Style (CPS). This trades some efficiency for extra features:

  - The ability to run highly recursive programs in environments with a limited stack depth (which is very common; e.g. most C environments limit the stack to a small fraction of the total amount of RAM).
  - With a custom interface, it enables an RPython program to inspect and control its own execution, by manipulating its own activation records.

  Using these two features in our full Python interpreter enabled us to expose interesting new features to the application-level Python users: massive parallelism, soft-switching, clonable and picklable coroutines, etc.

- Optimizations and Memory Management:

  Working on Task 2 of WP07, we implemented a number of transformations for the purpose of memory management, on the one hand, and general optimizations on the other hand.

  - When the target is C, the graphs of the source RPython program (which assumes automatic memory management) must at one point be enhanced with operations that can inspect and reclaim the consumed memory. We have a number of solutions available for that: we can make the C environment itself manage its own memory with the Boehm conservative garbage collector (GC); we can insert and maintain reference counts in all RPython objects; or we can provide more advanced GCs, written themselves in RPython and whose graphs are merged with the main program's graphs.

- We also experimented with a large range of optimization-only transformations. In this way, we succeeded in reclaiming a reasonable execution performance: the translated PyPy is within the same order of magnitude as CPython, despite the overhead typically associated with writing programs in high-level languages (RPython) instead of low-level ones (C). These transformations include inlining, escape analysis, allocation removal, and pointer tagging. Additionally, we also describe a few manual rewrites that we did in the RPython source code of PyPy.

Most transformations described in the present report are not pure graph-to-graph transformations, but require the addition of new graphs implementing new functionality (e.g. the GC logic). This is achieved by writing the new functionality as RPython code that manipulates objects at the appropriate level (e.g. RPython code manipulating address- and pointer-like objects in the case of the GC). These new "system code" helpers are then sent through the same translation toolchain to produce the extra graphs. The transformation proper only inserts calls to these extra graphs into the original graphs as appropriate (e.g. each memory allocation operation in the original source graphs is replaced by a call to the graph of the corresponding GC helper). The detailled description of this process is beyond the scope of this document, but can be found in (VMC) (in annex).

2 Introduction

2.1 Purpose of this Document

This document describes how massive parallelization features are implemented in the PyPy translation toolchain and weaved into the code and how these features are exposed to the Python developers using modules. It describes the optimizations of the PyPy translation toolchain in technical detail. It also describes how different garbage collection strategies are woven into the translated program during the translation process and how the garbage collectors are implemented.

2.2 Scope of this Document

The document describes the translation features that were implemented in the 0.9 release, which includes a high-performant version of the pypy interpreter, which allows the user to choose between the various garbage collection strategies and whether to include stackless features or not. The document does not describe the translation process in detail.

2.3 Related Documents

This document assumes some basic knowledge about PyPy's translation process, especially the RTyper, graph transformations and low-level helpers. Therefore a close look at the following document is recommended before reading this one:

- PyPy's Approach to Virtual Machine Construction (VMC)

For a theoretical background about the translation process see:

- D05.1: Compiling Dynamic Language Implementation (D05.1)

3 Stackless Features

(preliminary version - this chapter not present)

3.1 The Stackless Transformation

3.1.1 "Unbounded" Stack

3.1.2 Interface to Explicit Stack Manipulations

3.1.3 Resume Points

3.1.4 Performance Impact

3.2 Application Level Interface

3.2.1 Coroutines

3.2.2 Greenlets

3.2.3 Tasklets

3.2.4 Tasklet Pickling

3.2.5 Tasklet Cloning

3.3 Practical Uses of Massive Parallelism

4 Optimizations and Memory Management

One of PyPy's most important goals is to be able to keep the implementation of PyPy's interpreter at a very high level of abstraction and not make it more complex by many manual optimizations and tweaks. To still achieve high performance it becomes necessary to implement optimizations that remove inefficiencies. PyPy also tries hard to avoid making certain decisions at implementation time, such as what garbage collection to use, how certain objects are implemented, etc. Therefore the PyPy interpreter can be seen as an executable specification of the Python language. All the decisions that were not made during the implementation of the interpreter can and indeed have to be made during translation time. Thus various aspects of the implementation are "weaved" into the interpreter during the translation process (see (D05.3) and (D05.4) for more details about translation aspects).

The usefullness of the various optimizations and the performance of the various garbage collectors where evaluated with two benchmarks: The pystone benchmark, which is a port of the Dhrystone 2.0 (DHRY) benchmark to Python which is quite often used in the Python community, and a Python port of the classical richards benchmark.

4.1 Optimizations

When the whole PyPy interpreter was successfully translated to C ((D05.1)) for the first time, the performance of the resulting binary was already much faster than running PyPy on top of CPython. On the other hand the binary was still quite a lot slower than CPython itself, namely between 22 times slower (for the Richards benchmark) and roughly 460 times slower for the pystone benchmark. Subsequently this performance was increased using two different, complementary approaches. On the one hand we tweaked the source of the interpreter itself to introduce specialized code for common cases, remove inefficiencies, etc. On the other hand we implemented a number of general optimizations that transform the flow graphs after they were rtyped but before they are turned into source code of the target platform. These optimizations are described in the following.

One of the biggest sources of inefficiencies is the large amount of time spent managing memory. This is caused by the fact RPython itself is a garbage collected language and idiomatic RPython programs allocate memory extremely frequently. Therefore many optimizations try to help in this area by reducing the amount of objects being allocated or by trying to foresee the lifetime of an object.

In general PyPy proved to be a good environment to work on optimizations. Due to the multistage approach choosen where flow graphs are refined and changed towards the level of the target language in several steps it is possible to choose exactly the right level to do an optimization (see (VMC)). Another advantage is that on all levels the same basic data structures are manipulated, which eases the writing of transformations. Due to the fact that the input of the translation toolchain is a RPython program it is possible to do a lot of optimizations that would not be possible with a low level language as input. The optimizations can assume that no unsafe operations (such as accessing arbitrary memory or accessing memory of the wrong type) are performed because such operations are not possible in Python. In addition the optimizations have full information about types on all levels available which also makes some optimizations possible.

4.1.1 Function Inlining

To reduce the overhead of the many funtion calls that occur when running the PyPy interpreter we implemented function inlining. This is a well-known optimization which takes a flow graph and a callsite and inserts a copy of the flow graph into the graph of the calling function, renaming occuring variables as appropriate. This leads to problems if the original function was surrounded by a `try: ... except: ...` guard. In this case inlining is not always possible. We implemented a simple heuristic that tries to detect whether an exception raised by the called function matches one of the exceptions caught in the exception handler. If that is the case, those are directly matched to each other during inlining.

In addition we also implemented heuristics to decide which function to inline where. For this purpose we assign every function a "size". This size should estimate the increase in code-size which is to be expected should the function be inlined somewhere. This estimate is the sum of two numbers: for one every type of operation is assigned a specific weight, the default being a weight of 1. Some operations are considered to be more effort than others, e.g. memory allocation and calls. Others are considered to be no effort at all (e.g. casts). The first component of the size estimate of a graph is the sum of the weights of all operations occuring in the graph. This is called the "static instruction count". The other component of the size estimate of a graph is the "median execution cost". This is again the sum of the weight of all operations in the graph, but this time the weight of every operation is multiplied with a guess how often the operation is executed. To arrive at this guess we assume that at every branch we take both paths equally often, except for branches that are the end of loops, where the jump back to the end of the loop is considered more likely. This leads to a system of equations which can be solved to get approximate factors for all operations.

After the size estimate for all function has been determined, functions are inlined into their callsites, starting from the smallest functions. Every time a function is being inlined into another function, the size of the outer function is recalculated. This is done until the remaining functions all have a size greater than a predefined limit. Of course only function call sites can be inlined if the called function can be statically known. This is not the always possible, e.g. when function pointers are used. Special care was taken to make the inlining process deterministic, e.g. the same functions should inlined into the same places when the translation is run twice with the same input. This was originally not the case when two function have the same size assigned. In this case one of them was arbitrarily picked over the other. To break such ties we now inline the function with less callers first. In theory it could still be possible to have draws between functions that have the same size estimate but we this seems to happen rarely enough not to be a problem.

Inlining gives interesting speedups. The pystone benchmark becomes 52% faster when using only inlining compared to doing no optimizations at all, the richards benchmark becomes 64% faster.

4.1.2 Malloc Removal

RPython (see (D05.1) for description of RPython) is a garbage collected language and uses memory allocation freely in a lot of places. This leads to memory allocations in places where a more traditional language would not have those. For example a loop of the following form:

```
for i in range(n):
    ...
```

which simply iterates over all numbers from 0 to n - 1 is equivalent to the following in Python:

```
l = range(n)
iterator = iter(n)
try:
    while 1:
        i = iterator.next()
        ...
except StopIteration:
    pass
```

This means that three memory allocations are executed: The range object, the iterator for the range object and the StopIteration instance, which ends the loop.

After a small bit of inlining all these three objects are never even passed as arguments to another function and are also not stored into a globally reachable position. In such a situation the object can be removed (since it would be garbage anyway after the function returns) and can be replaced by its contained values.

This pattern (an allocated object never leaves the current function and thus dies after the function returns) occurs quite frequently, especially after some inlining has happened. Therefore we implemented an optimization which "explodes" objects and thus saves one allocation in this simple (but quite common) situation.

This optimization first computes sets of variables that hold pointers to structures or arrays. Two variables are in the same set if one of them can be passed as a value into the other along a link in the flow graph. For each of these sets, we record the places where a variable in the set is used, as well as the places where a variable in the set is assigned to. A set of variables is a candidate for removal if:

- all the variables in the set were created by one single malloc;
- all the variables in the set are only accessed by operations that read or write fields from or to the structure that the variable points to.

If these conditions hold the variable can be exploded into its components.

This all means that the optimization gives up if a variable created by a malloc can reach a place that a variable created in any other way (result of a function call, another malloc, etc.) reaches. It also gives up when the variable is passed as an argument to a function or returned from the current function - or stored into a field of another data structure. In the later case, though, if the other data structure is itself candidate for removal, it will vanish together with the store-into-field operation, and the former data structure can become candidate for removal again.

This malloc-removing transformation is made useful only by the inliner (described above). Without inlining objects are passed into other functions most of the time, which makes malloc removal give up. On the other hand, after some inlining happened quite a lot of objects can be removed. Indeed inlining was tweaked in such a way that malloc removal can work effectively.

The speedup when using the malloc removal optimization together with inlinining compared to just using inlining is 18% faster for the richards benchmark and 43% faster for the pystone benchmark.

4.1.3 Escape Analysis and Stack Allocation

Another technique to reduce the memory allocation penalty is to use stack allocation for objects that can be proven not to live longer than the stack frame they have been allocated in.

If this is the case it is possible to allocate the object on the stack. This makes allocation faster, since stack allocation is just the increase of a pointer, and makes deallocation basically free since deallocation happens automatically when the function returns. Stack allocation is a well explored technique whith very sophisticated algorithms, see for example (BLANCHET99), (CHOI99).

To enable stack allocation where possible we wrote an analysis, which detects which malloc positions lead to mallocs which "escape" the current function, e.g. have references to them stored into a place where they can be accessed by something outside of the stack of frames starting with the frame where the malloc occured.

For this we choose a naive, pessimistic approach (FRANZ02) which was quite easy to implement, to evaluate whether the general approach gives speedups at all. The analysis assumes that an object escapes if one of the following situation occurs:

- the object is returned
- the object is raised as an exception
- the object is stored into a field of any another object

The algorithm assigns to every variable a set of "creation points" like "malloc", "constant", "returned by function call", etc. The creation points are forward-propagated to all variables that they can reach. If a variable is returned from the current function, raised as an exception or stored into the field of another object the analysis gives up. In this case the creation points that are associated to this variable are marked to be escaping. This is done using a fix-point algorithm until no change occurs anymore.

After using the escape analysis to find malloc sites that don't escape, we replace these mallocs by stack allocations (using local variables). This cannot be done in all cases, namely if the allocated object is variable-sized or if the allocation occurs in a loop. Both cases should be avoided because they make stack overflows more likely. Also objects that have a finalizer cannot be allocated on the stack, since the finalizer might resurrect the object (see section "Memory Management" below).

The resulting performance improvements by this optimization were quite poor when applied together with all other operations. The pystone benchmark became 3% faster, the richards benchmark 7%. This is the case because escape analysis and malloc-removal optimize very similar cases, so after the malloc-removal is done the escape analysis is left with a lot less cases where it can move objects to the stack. This is shown by turning off malloc removal and comparing with and without escape analysis. In this case the speedup is 40% for the pystone benchmark and 30% for the richards benchmark.

4.2 Pointer Tagging

A common technique to decrease the memory allocation overhead when dealing with many integer calculations is the use of tagged pointers. Due to machine word alignment constrains of modern architecture, the addresses of the start of an object are always even numbers (and more likely multiples of 4, at least). That means that one can mix pointers and integers, as long as the lowest bit of the integer is set. This can be used to represent small enough integers by shifting them one bit to the left and setting the lowest bit. These integers have the advantage that they don't need allocation at all, being stored in the pointer itself.

We used this technique of pointer tagging to implement an additional application-level integer type in the PyPy standard object space (D04.2) that can only be used for "small" integers,

where "small" means "fits in N-1 bits, where N is the number of bits of a pointer". Small integers behave like standard integers as far as the interpreted application can tell; the two of them are exposed to the user as a single type, and they are undistinguishable (using PyPy's multimethod-based machinery (D04.2)). Internally, every time a new integer is created anywhere it is checked whether the integer is small enough or not, and the corresponding implementation is used.

Our particular implementation of pointer tagging (an orthogonal, optional 156-lines module in the translation toolchain) only produced interesting speed improvements when combined with a simple constant folding optimization. This optimization folds side-effect-free operations whose arguments can be determined to be constant and replaces the resulting variable with the result of the computation. Although all C compilers already do that, there is a situation where the C compiler lacks an essential piece of information: the virtual method tables generated by the translation toolchain are *constant* structures, so it is valid to constant-fold all reads from these tables. This is critical in the pointer tagging scheme because at runtime, once the lowest bit has been determined to be set, we know the exact class of that object (in the case of the translation of PyPy, we know it is a "small integer" object). Normally, a virtual method call to this object would involve reading the method pointer out of the virtual method table, but in this case the virtual method table pointer itself is a constant, so the indirect method call can be constant-folded to a direct function call.

In this way, we have recreated with minimal efforts a pluggable pointer tagging scheme that produces low-level code very similar to what is more typically achieved with a set of invasive, whole-program C macros.

Without the simple constant folding the tagged integers improved the speed of pystone by 4% and decreased the speed of the richards benchmark by 3%. Together with the constant folding the speed of pystone increased by 7% and the speed of the richards benchmark by 1% compared to a regular pypy build (also with constant propagation enabled).

4.3 Manually Rewritten Code

4.3.1 The Dispatch Loop

The bytecode dispatch of PyPy's bytecoe interpreter (D04.2) was originally implemented as a list of function pointer, into which the dispatch would index using the bytecode and call the found function. This has some drawbacks, the biggest being that none of the functions that implement a bytecode can be inlined into the dispatch loop, despite being very small and simple, 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 ((CPy)) 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 implemented the bytecode dispatch as a long chain of blocks of the following form:

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

Without further changes this would have been even slower, since a potentially large number of comparisons has to be performed. To overcome this problem we implemented a small 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. After that the calls are all direct which means that inlining can do its work and inline the simple opcodes.

With this optimization the richards benchmark became 5% faster, the pystone benchmark didn't change at all.

4.3.2 Multimethod Dispatch

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 in (MRD). It only requires a pair of integers per virtual method table, plus a few small compressed tables. This shrinks the executable of the PyPy interpreter by 2.5 MB. The run-time performance is very similar (only a few percents slower): the previous scheme required a double indirect jump, while the newer 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.

4.4 Memory Management

As mentioned above, RPython is a garbage-collected language. Therefore many parts of PyPy's toolchain (for example the Flow Object Space, the Annotator, the RTyper, see (D05.1) for a descripition) assume that memory will be automatically managed in the RPython program. When targetting low-level languages (for example when using the C or LLVM backend) the target language does not have explicit memory management. In this case the toolchain has to weave in a garbage collector into the flow graph to make memory management explicit. Memory management is therefore a "translation aspect" that can be chosen at compile time (see (D05.3) an (D05.4) for more details about translation aspects). This weaving in is done by the GC transformer (see section "The GC Transformer"_). Note that as with other translation aspects we evolved to program GC integration code as flow graph transformations instead of writing backend-specific code.

Traditionally the choice of memory management strategy was made explicitly, early during the creation of a VM implementation. Indeed, object layout is often among the very first decisions that VM implementors make (this is reflected in the fact that such a layout is typically a major topic in technical introductions to VMs; see e.g. (SQUEAK)). This leads to a deep entanglement of the GC and the implementation. A good example of the resulting issue is CPython, where reference counting was chosen early and now accounts for the fact that operations to increase and decrease reference counts are sprinkled over all the source files, which makes changing this choice very hard.

This is quite different from PyPy's approach. There the implementation language (RPython) itself is garbage collected and different garbage collectors are inserted into the resulting code during translation time. This leads to a great deal of flexibility. It is for example possible to choose a different garbage collector depending on the intended use case or to choose between different size-speed trade-offs. This flexibility also makes PyPy an excellent environment to experiment with and evaluate different garbage collection strategies. It is possible to try

out different garbage collectors for the same program by weaving in different collectors. Additionally PyPy tries very hard to make debugging and testing even of low-level code as easy and painless as possible (see "The GC Construction and Testing Framework"_ section below).

One source of complexity of writing garbage collectors for PyPy is that the collectors have to support finalization. The finalization methods of objects are capable of -- and indeed, cannot be prevented from -- resurrecting the finalized object by storing a reference to it into a globally reachable position. The garbage collectors have to be written in such a way that this does not lead to crashes or similar.

4.4.1 The GC Transformer

The purpose of the GC Transformer is to inserts garbage collection code into the flow graphs during translation. Its input is a flow graph which assumes automatic memory management. This flow graph is transformed in such a way that it contains all operations necessary for explicit garbage collection afterwards. To do this it replaces operations that deal with the allocation and the management of memory by calls to functions that are part of the selected garbage collector. If a GC needs a write (or even read) barrier then operations that write/read a value out of an object are replaced by a call to the necessary barrier function. To increase performance some of these operations can be specified to be always inlined. Usually this is done for the fast path of certain performance critical operations, like memory allocation. This is similar to the control over inlining that MMTK uses (MMTK).

All the operations that are needed by the selected garbage collector are implemented as low level helpers (as described in the (VMC) paper). These helpers are fed into the toolchain to get flow graphs for them. In the original function graph that is transformed some of the operations are replaced by calls to these helper graphs thus lowering the level of the operations in the graph.

4.4.2 Using The Conservative Boehm Collector

The simplest way to transform flow graphs to have memory management is to use the Boehm garbage collector (BOEHM). The Boehm collector is a conservative Mark-and-Sweep collector for C programs. It is used by linking in a library, using a special malloc function and then just not explicitely freeing any objects at all. Therefore the GC Transformer for the Boehm garbage collector is quite simple. It does mostly nothing but replace the allocating operations with the special boehm versions. In addition it has to register finalizers with the Boehm collector for objects that have a `__del__` method or any other form of finalization.

One of the problems that occurs when using the Boehm collector is that since it does not know anything about the data structures the program uses, it has to assume that everything is a pointer. This becomes a problem if an integer with the same bit-pattern as a pointer to a valid object is stored somewhere. If there are no references to this object left it is nevertheless being kept alive by the integer, since Boehm cannot decide whether the integer is really a pointer. This was an actual problem for PyPy, since the default implementation of identity hashing uses the memory address as the hash. That means that if the identity hash of an object is stored somewhere, this object will be kept alive when using the Boehm collector, even if it is not referenced from anywhere.

To solve this problem we changed the way identity hashes are calculated using the Boehm collector to no longer use the memory address but rather the bitwise inverse of the memory address. This still leads to unique hashes, but does not have the described problem.

There were some more problems with the Boehm collector: We observed segmentation faults when trying to allocate very large chunks of memory. Furthermore Boehm seems to sometimes have problems with objects that have finalizers registered and that are in a reference cycle with other objects. Both problems seem to occur rarely in practice, though. On the other hand, Boehm seems to be the fastest garbage collector option we currently have. Using the Boehm GC the richards benchmark using PyPy is 5.8 times slower than on top of CPython, the pystone benchmark is 4.4 times slower.

4.4.3 Reference Counting

A relatively simple memory management strategy is that of reference counting (see (GC-SURV) or (GC) for overviews over standard garbage collection techniques). Reference counting is the strategy that CPython uses. It has many advantages, such as immediate reclamation of resources and ease of implementation. There are also many disadvantages such as relative slowness (especially if the code is not highly tuned) and the fact that reference counting does not reclaim cyclic garbage. CPython overcomes the problem with cyclic trash by having an additional cycle detector which finds inaccessible cycles and breaks them.

The reference counting GC transformer takes a flow graph and inserts calls to the *increase reference* (incref) and *decrease reference* (decref) helpers (which are then inlined). The approach used is quite naive. Incref and decref calls are inserted into all necessary places without checking for redundancy. That means that the result contains a lot of incref/decrefs that would not really be neccessary. This makes the executable both a lot bigger and also adds a lot of unneccessary operations which makes reference counting one of the slowest of our GCs (richards being 7.7 times slower than with CPython, pystone 7.8 times). This slowness makes it also impossible to observe any improved cache behaviour that is sometimes associated with reference counting (DETREVILLE90). In addition we did not implement any sort of cycle detection, which means that cyclic garbage is never reclaimed.

The reference counting garbage collector handles finalization by calling a special function associated with the objects in need of finalization when the reference count reaches zero. If the reference count is positive again after the finalizer has run the object has been revived and is thus not reclaimed. If the reference count stays zero the object is freed.

Currently the refcounting garbage collector does not seem to us a very interesting area to work on since it is a lot easier to get high performance using other algorithms (see section "The Mark-and-Sweep Collector"_). We should note, though, that reference counting is nevertheless a possibly viable option. In the long term, given additional efforts on optimizations - e.g. incref/decref pair removal, reference-borrowing detection - it should be possible to obtain a fair comparison between reference counting and other GCs, to either confirm or refute the informal impression that exists among CPython developers, saying that reference counting is not such a bad choice in that specific context.

4.4.4 The GC Construction and Testing Framework

A central approach of the PyPy project is to use Python as a system programming language (VMC). Therefore we also set out to make it possible to implement the garbage collectors for PyPy in Python itself. An additional goal was to be able to simulate them on top of CPython to make testing easier.

To make this possible we implemented a garbage collection framework and a memory simulator. The GC framework on the one hand provides hooks to the GC implementation so that the GCs can get at information about the layout of objects in memory and about the stack

and static roots. On the other hand the GCs have to implement a certain set of operations so that then it becomes usable for the rest of the program.

The Python code that implements the GC operations manipulate objects that are on a relatively low level (objects that behave like pointers and addresses). This Python code is transformed into flow graphs and calls to these are inserted into the graphs that are transformed to contain garbage collection afterwards. Despite these quite heavy restrictions Python is still more expressive as a programming language than, say, C. The PyPy GC framework was inspired by the memory mamagement toolkit (MMTK) that is used to implement the garbage collectors of the Jikes RVM (JIKES) in Java. Much more work and fine-tuning went into MMTK, though.

The garbage collector needs a way to follow object references on the heap to analyze liveness of objects. The GC also needs a way to get the finalizer associated with an object type. To make this possible the garbage collector has access to some operations that give him information about the layout of objects in memory. These operations always map a typeid (an integer describing the layout of a specific object type) to some information about this type. These operations are implemented by using static tables of data. This is made possible by the fact that we know all types that are used in advance. The tables can therefore be created at compile time.

To run the garbage collection implementations on top of CPython for testing purposes we implemented a memory simulator which provides the simulation of the typical memory operations and objects. These are allocation and freeing of chunks of memory, addresses (pointers) and read and write operations to and from these addresses. In addition the memory simulator checks for typical errors such as reading from non-allocated memory addresses, from the NULL address, freeing an address twice or reading from non-initialized memory. This makes working on the garbage collectors a much more pleasant experience since it does not involve constant segmentation faults.

4.4.5 Explicitely Managed Objects

The datastructures that the GC itself uses need to be explicitly garbage collected. To enable that it is possible to flag classes as needing explicit memory management. If this is the case instances of the class are not managed by the garbage collector but have to be explicitely freed. When running on top of Python accessing a freed object gives again a useful error message to ease debugging on top of CPython.

4.4.6 Finding Roots

One of the hardest problems for a garbage collection framework that works without the cooperation of the low-level compiler is how to find the roots, that means how to find out which objects are accessible from the stack. This is especially hard to achieve using only ANSI C and in a platform independent way. For example the Boehm collector uses a set of platform specific hacks to access the stack.

One suboptimal solution we implemented is to keep an extra stack of roots where all the locations of currently accesible objects are pushed onto and popped from during program execution. This provides a standard-conformant solution to the problem but has the drawback of not overly good performance. To maintain this root stack the GC transformer inserts push/pop operations into all appropriate places in a function graph.

Another solution is to use the unwinding features provided by the stackless transformation (see section The Stackless Transformation above) to unwind the stack before a collection and

then find the roots in the chain of frames which now resides on the heap (and which is in a format we know well). We implemented this strategy but it made the executable slower (15% slower compared to the version that uses explicit push/pops on richards, 13% on pystone). The problem with this approach is that every malloc operation can potentially unwind the stack, which is not the case if we don't rely on stack unwinding to find roots. This makes the size of the executable significantly bigger (70% bigger code segment in the executable, compared to a regular stackless build) so that all possible performance improvements are neutralized by very bad instruction cache behaviour.

4.4.7 GC-Provided Operations

The garbage collector has to implement operations that are inserted by the GC transformer into all the flow graphs. These operations are:

`malloc(typeid, length) --> address`: returns the address of a suitably sized chunk of memory for an object with the type described by typeid.

`collect()`: triggers a garbage collection.

`write_barrier(addr, addr_to, addr_container)`: the implementation of this operation is optional, only some GCs need a write barrier. This operation is called when pointer (`addr`) to an object managed by the GC is written into another object on the heap. The argument `addr_container` is a pointer to the beginning of the object the write happens to.

The `write_barrier` and `malloc` operations can be inlined to increase performance.

4.4.8 The Mark-and-Sweep Collector

The framework GC that is the most developed is a standard mark-and-sweep collector (see for example (GC) or (GCSURV)). At collection time it starts from the stack roots and the static roots and recursively sets a mark bit in objects that are reachable from the roots. After it cannot find any new non-marked objects it walks all allocated objects and frees those that don't have their mark bit set.

The mark-and-sweep collector keeps two words in front of every object for bookkeeping purposes. The word directly in front of the object is used to store the mark bit (the lowest bit of the word) together with the type id of the object (all other bits). The word before that is used to chain all allocated objects together in a linked list.

The mark-and-sweep collector supports proper finalization and resurrection of objects. The collector keeps all objects that need finalization in a different linked list than that for regular objects. If at the end of a collection an object in this list is not marked then all the objects reachable from it are marked and its finalizer called. Afterwards it is considered live and put into the regular linked list. Only if the object is dead at the *next* collection it is actually deleted. This behaviour makes resurrection possible. It also leads to all finalizers being called at most once. This change in behaviour should not matter for most user programs since usually finalizers don't ressurect the object being finalized. In CPython the fact that finalizers are invoked arbitrarily often is a source for interpreter crashes and some CPython developers thought about changing that for CPython too (DEL).

With the mark-n-sweep collector we reach quite reasonable performance, although not quite as fast as with the boehm collector: Richards is 5.8 times slower than CPython, pystone 5.7 times.

4.4.9 Performance & Future work

The following table lists performance measurements for a range of PyPy builds, comared to CPython. The measurements were done on a AMD Opteron 242, 1603 MHZ, 1024 KB cache size with 4GB of RAM. The pystone column lists the time in milliseconds for one iteration of the Pystone benchmark, the richards column lists the time for a full run of the richards benchmark. The "rel. ps" column list the pystone times relative to pypy-c-normal, the "rel. ri" column does the same for the richards benchmark. The pypy-c-normal is the default build of PyPy, using inlining, malloc remocal and the Boehm GC. When not specified, the Boehm GC is used (except for pypy-c-0.7, which uses reference counting).

Executable	pystone	rel. ps	richards	rel. ri
CPython 2.4.3	1800	0.23	1222	0.23
pypy-c-0.7	827390	104.73	26885	5.13
pypy-c-normal	7900	1	5244	1
pypy-c-no-optimizations	17660	2.24	11540	2.20
pypy-c-just-inlining	11650	1.47	7050	1.34
pypy-c-inlining-and-malloc-removal	8100	1.03	5945	1.13
pypy-c-tagged-pointers-constfold	7360	0.93	5174	0.99
pypy-c-tagged-pointers-no-constfold	7590	0.96	5409	1.03
pypy-c-old-multimethod-dispatching	7450	0.94	5080	0.97
pypy-c-escape-analysis	7640	0.97	4881	0.93
pypy-c-escape-analysis-no-malloc-removal	8170	1.03	5425	1.03
pypy-c-old-bytecode-dispatching	7820	0.99	5532	1.05
pypy-c-reference-counting	14010	1.77	9381	1.79
pypy-c-mark-and-sweep	10270	1.30	6813	1.30
pypy-c-mark-and-sweep-stackless-roots	12850	1.63	8150	1.55

As can be seen from the table all our optimizations together more than double the speed of the PyPy interpreter (compare pypy-c-normal with pypy-c-no-optimizations). Also our garbage collectors deliver quite reasonable performance, especially if one takes into account the low amount of tuning that went into them. We hope to improve them in the future.

One of the findings we made when we profiled the translated PyPy interpreter was that a significant amount of time was spent zeroing the allocated memory. At the moment the lltype system (VMC) assumes that all memory allocated by malloc returns zeroed memory. Removing this assumption would potentially give quite some speedup. Another thing to work on is to implement moving garbage collectors such as a semi-space copying collector. This would have the advantage that allocation is extremely fast (basically just incrementing a pointer). This should be relatively little work, some care is needed to still have a consistent identity hash function after objects were moved.

Another area to work on is the optimization of increase/decrease reference operations in the reference counting case as well as the pushing and popping of roots for the framework GCs. Right now quite a lot of superfluous operations are made there, especially for reference counting.

A future possibility to alleviate the root finding problem is to use information provided by our still unfinished just-in-time compiler to find the roots on the C stack. This will be possible since the JIT generated the code so it knows the stack frame layout.

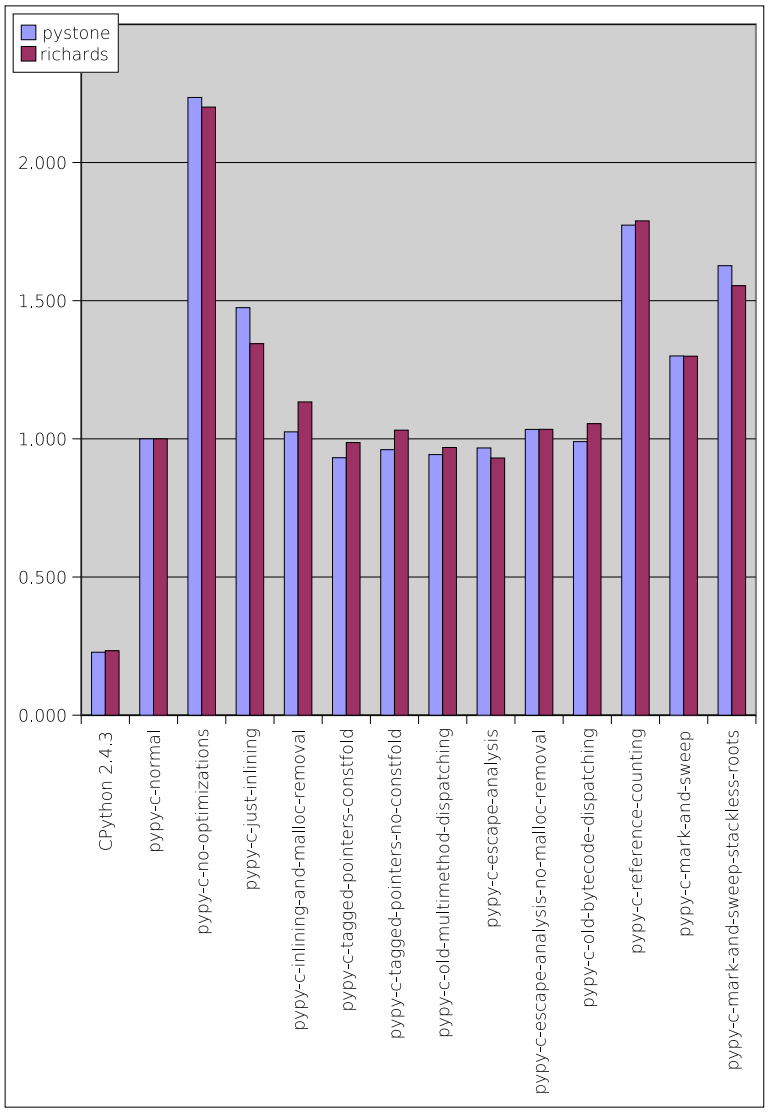

Figure 1: Benchmark results of the various PyPy versions compared to CPython. The results are normalized to pypy-c-normal. The pypy-0.7 data is left out.

4.5 Conclusion

(preliminary version: conclusion about Task 1 missing)

We implemented and experimented with a wide range of optimizations and memory models. Various optimizations were found that increase the performance by significant amounts.

PyPy's architecture has many upsides that make it easy to implement optimizations. One of them is the uniform graph model which can be extended during the process. Indeed we found that the combination of graph transformations and the ability to produce new low level helpers at any time is very powerful and pleasant to work with. One of the possible improvements of this process would be to have a more refined interface for graph manipulations, since this can be tedious at time. This would require some design work to find out what common transformation patterns are. In any case, PyPy's translation tool chain - with relative ease - opens up the possibility for researching optimizations and algorithms in more depth and also in correlation to each other.

Another, less immediately obvious, advantage of PyPy is that the input to the translation process is a typesafe high-level language which ensures that there are no unsafe operations in the produced graphs. This makes many optimizations a lot easier and also a lot more powerful.

On the other hand it is also one of the drawbacks of PyPy's approach to start from a garbage-collected high level language which allocates memory very quickly (and thus produces garbage very quickly) since this increases the pressure on the performance of the GC. However, our experiences and results show that using and optimizing a GC collected language for implementing high level languages is a worthwhile effort and warrants further research and refinements.

5 Glossary of Abbreviations

The following abbreviations may be used within this document:

5.1 Technical Abbreviations:

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.
docutils	The Python documentation utilities.
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.
Graphviz	Graph visualisation software from AT&T.
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 Direct-media Library - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
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.

5.2 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

References

(ARCH) *Architecture Overview*, PyPy documentation, 2003-2005

(D04.2) *Complete Python Implementation*, PyPy EU-Report, 2005

(D05.1) *Compiling Dynamic Language Implementations*, PyPy EU-Report, 2005

(D05.3) *Implementation with Translation Aspects*, PyPy EU-Report, 2005

(D05.4) *Encapsulating Low-Level Aspects*, PyPy EU-Report, 2005

(VMC) *PyPy's approach to virtual machine construction*, Armin Rigo, Samuele Pedroni, draft paper, accepted to OOPSLA, to appear in 2006

(MRD) *Multi-Method Dispatch Using Multiple Row Displacement Tables*, Pang et al, Lecture Notes in Computer Science, 1999

(BOEHM) *Garbage collection in an uncooperative environment*, Boehm, Weiser, Softw. Pract. Exper., 1988

(GCSURV) *Uniprocessor Garbage Collection Techniques*, Paul R. Wilson, Proceedings of the International Workshop on Memory Management, September 1992

(GC) *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*, Richard Jones, Wiley & Sons, 1996

(MMTK) *Oil and Water? High Performance Garbage Collection in Java with MMTk*, Blackburn et al, 2004

(CPy) CPython 2.4.3, Guido van Rossum et al, March 2006, http://www.python.org

(JIKES) *The jalapeno virtual machine*, B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, IBM Systems Journal, 39(1):211-, 2000

(SQUEAK) *Back to the future: the story of Squeak, a practical Smalltalk written in itself*, Dan Ingalls and Ted Kaehler and John Maloney and Scott Wallace and Alan Kay, Proceedings of the 12th ACM SIGPLAN OOPSLA conference, 318-326, 1997

(DHRY) *Dhrystone benchmark: rationale for version 2 and measurement rules*, R. P. Weicker, SIGPLAN Not. 23(8): 49-62, 1988

(CHOI99) *Escape Analysis for Java*, Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, Sam Midkiff, in OOPSLA p. 1-19, 1999

(BLANCHET99) *Escape Analysis for Object Oriented Languages. Application to Java*, Bruno Blanchet, in OOPSLA p. 20-34, 1999

(FRANZ02) *Online Verification of Offine Escape Analysis*, Michael Franz, Vivek Halda, Chandra Krintz, Christian Stork, Technical Report No. 02-21, University of Carolina, Irvine.

(DETREVILLE90) *Experience with Concurrent Garbage Collectors for Modula-2+*, John DeTreville, Technical Report 64, Digital Equipment Corporation System Research Center, 1990

(DEL) Discussion on the python-dev mailing list, Tim Peters, Guido van Rossum, Armin Rigo..., August 2005, summary at http://www.python.org/dev/summary/2005-08-01_2005-08-15/#pep-344-and-reference-cycles