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

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: July 31, 2006

Start date of Project: 1st December 2004 Duration: 2 years

Lead Contractor of this WP: Tismerysoft

Authors: Carl Friedrich Bolz (merlinux GmbH), Armin Rigo (HHU)

Revision: Final (pending official approval)

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
2006-10-16	Armin Rigo	added description of stackless transformation
2006-11-08	Michael Hudson	added related work sections
2006-11-25	Armin Rigo et al	added app-level interface and microbenchmarks
2006-12-07	Armin Rigo et al	final review
2006-12-15	Carl Friedrich Bolz	publish near final version on web page
2007-02-25	Carl Friedrich Bolz	peer review suggestions of Christopher Armstrong

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 of its own stack frames; 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 woven 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 collectors give reasonable performance results.

Contents

1 Executive Summary **5**

2 Introduction **7**

2.1 Purpose of this Document . . . 7
2.2 Scope of this Document . . . 7
2.3 Related Documents . . . 7

3 Stackless Features **8**

3.1 Introduction: Stackless Features for RPython Programs . . . 8
3.2 The Stackless Transformation . . . 8
3.2.1 Saving and Resuming Frames . . . 9
3.2.2 "Unbounded" Stack . . . 11
3.2.3 Interface to Explicit Stack Manipulations . . . 11
3.2.4 Resume Points . . . 13
3.2.5 Performance Impact . . . 13
3.3 Application Level Interface . . . 15
3.3.1 Coroutines . . . 16
3.3.2 Tasklets and Channels . . . 17
3.3.3 Greenlets . . . 18
3.3.4 Coroutine Pickling . . . 19
3.3.5 Coroutine Cloning . . . 21
3.3.6 Composability . . . 22
3.3.7 Microbenchmarks of Stackless Features . . . 25
3.4 Practical Uses of Massive Parallelism . . . 27
3.5 Related Work . . . 28

4 Optimizations and Memory Management **31**

4.1 Optimizations . . . 31
4.1.1 Function Inlining . . . 32
4.1.2 Malloc Removal . . . 33
4.1.3 Escape Analysis and Stack Allocation . . . 34
4.1.4 Pointer Tagging . . . 35
4.2 Memory Management . . . 36
4.2.1 The GC Transformer . . . 36
4.2.2 Using The Conservative Boehm Collector . . . 37

4.2.3 Reference Counting 37
4.2.4 The GC Construction and Testing Framework 38
4.2.5 The Mark-and-Sweep Collector 40
4.2.6 Performance & Future work 40
4.3 Related Work 41

5 Conclusion **43**

6 Glossary of Abbreviations **44**
6.1 Technical Abbreviations: 44
6.2 Partner Acronyms: 45

1 Executive Summary

PyPy's tool-chain is able to translate a subset of the Python language, RPython (D05.1), to other (typically lower-level) languages and platforms. It performs whole-program type inference and then specializes the code to match the target language's type system and features. In particular, we translate in this way our full Python implementation, which is a high-level implementation 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 toolchain processes the forest of high-level control flow graphs that represent 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. Translation proceeds in this way until the graphs' level of abstraction matches the expectations of the target platform. The flexibility obtained with this approach allows us to postpone decisions about which solution to use for each low-level issue until translation time (for example what strategy we use for garbage collection). It is also inherently retargetable, 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 have written and tried. Our initial focus has been on C-like target environments, which are in some sense the most demanding environments, being the most low-level. Those environments are also the focus of this report, as the transformations described below have so far only been implemented for them. This is only due to a choice of priorities; the core of our toolchain can now target the very different .NET environment as well and more backends (e.g. for the Java VM) are in progress, proving the soundness of our approach to retargetability.

In more detail, we discuss:

- Stackless Features:

  Corresponding to Task 1 of WP07, the so-called "Stackless" support is based on a graph transformation that takes as input graphs that contain recursive calls to each other – just like the source RPython program – and breaks these recursive cycles in the following sense: all cycles have at least one edge where a check is made to see whether the C stack is about to overflow. If this is the case, the stack is unwound (see below). This unwinding is made possible 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 (which is very common; for example 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 stack frames.

  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 and others.

- Optimizations and Memory Management:

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

  - When the target is C, the graphs of the source RPython program (which assumes automatic memory management) must at some point be imbued with operations that

can inspect and reclaim the consumed memory. We have a number of solutions available for this: 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 heap-allocated RPython objects; or we can provide more advanced GCs, written themselves in RPython and whose graphs are merged with those of the input program.

- 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 well 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.

Most transformations described in the present report are not pure graph-to-graph transformations but require the addition of new graphs implementing new functionality (for example 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 detailed 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 woven into the code and how these features are exposed to the Python developers using modules.
- The optimizations of the PyPy translation toolchain in technical detail.
- 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-performance version of the PyPy interpreter and 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 theoretical background about the translation process see:

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

3 Stackless Features

The term *Stackless* used throughout the present document has a complex history. It origin lies in the Stackless Python project, which was "A Python Implementation That Does Not Use The C Stack" (SLP0). The usage of the term has gradually evolved since then, along with Stackless Python itself. Currently, *a Stackless feature* means a feature that cannot be implemented without some form of explicit stack control, either by being very careful about the C code (e.g. avoiding recursive calls) or by saving and restoring pieces of the C stack by a brute force 'memcpy' approach. In PyPy, we use a variant of the former approach: a *stackless build of pypy-c* is a PyPy binary compiled from C code that was generated with careful systematic tweaks.

The motto of a Stackless PyPy is probably: "A Python Implementation That Uses the C Stack as a Cache". Indeed, the C code supports saving its own stack of frames away into the heap, and restoring it from there, frame by frame. So the C stack is really just a cache for the heap. If the cache is full (stack overflow), we save some more frames into the heap to free some cache space. If a context switch occurs (coroutine switch) the cache is invalidated (entierely flushed to the heap) and the new context's frames are copied from the heap to the cache as needed (the function's frames are resumed one by one).

3.1 Introduction: Stackless Features for RPython Programs

The Python interpreter of PyPy uses interpreter-level recursion to represent application-level calls. The reason for this is that it tremendously simplifies the implementation of the interpreter itself. Indeed, in Python, almost any operation can potentially result in a non-tail-recursive call to another Python function; this makes writing a non-recursive interpreter extremely tedious (Stackless Python version 1 was such an attempt – see the Related Work section). This is in contrast to many other languages, for which it is reasonable to write recursion-free interpreters (e.g. Squeak offers to applications continuation-like control over their own stack in this way). Instead, in PyPy we rely on lower-level transformations during the translation process to control this recursion. This is the Stackless Transformation, which is at the heart of PyPy's support for stackless-style concurrency.

The Stackless Transformation is the core of all the features exposed in the present chapter. It is a translation-time "aspect" which modifies the control flow graphs of an RPython program to enable new capabilities: the RPython program can save, restore, manipulate and even build from scratch its own stack of stack frames. This is described in detail in the Stackless Transformation section.

This gives any RPython program powerful continuation-like control over its execution. We have used this feature to build RPython coroutines and PyPy's full Python interpreter exposes these coroutines to application-level code in a varied choice of interfaces: we have three variants of coroutines, plus a purely application-level "software microthread" built on top of them. We describe the application-level interface available to users of PyPy in the Application Level Interface section.

3.2 The Stackless Transformation

The Stackless Transformation is an optional pass in our translation tool-chain that modifies the control flow graph of a subset of all functions in a systematic way. From a high-level point of view, this transformation adds to the graphs a new capability (or "aspect") not originally

present in the RPython source of the function: the input program gains the ability to suspend and restore its own execution by saving and restoring the state of the local variables between the (implicit, low-level) stack and an explicit heap-based data structure.

This is a form of stack introspection and manipulation that enables features such as continuations. It is more expressive than the traditional continuations, though, because it gives the RPython program explicit control over each frame of its own stack.

The Stackless Transformation is not the only way these features could have been implemented, but it has the significant advantage that the graphs that it produces place no more demands on the backend than the graphs it started with. In other words, from a low-level point of view, the transformed graphs are still regular functions performing regular calls to each other. These functions simply contain additional lightweight bookkeeping logic to be able to save and resume themselves, as described below.[1]

3.2.1 Saving and Resuming Frames

As with most of PyPy's transformations, the Stackless Transformation is actually two parts:

- a pure, one-graph-at-a-time transformation that introduces the necessary logic in each control flow graph, after each call; and
- a new set of helper functions to control the process: specific functions that initiate the state-saving process in some circumstances, as well as a top-level "driver" function at the bottom of the call stack that stops the state-saving process and controls the time at which the suspended function calls should be resumed.

The basic idea of the Stackless Transformation is the introduction of a new exception class, `UnwindException`. Its purpose is to "unwind" the complete low-level stack of calls, as a normal uncaught exception would, but with the difference that all interrupted functions should store away enough information to be able to restart themselves in the same state as they were. So whenever a function `f()` calls another function `g()`, the latter is allowed to (directly or indirectly) raise an UnwindException. When `f()` receives this exception, it should catch it, save away any relevant information about its state, and re-raise the exception.

The heap-based data structure mirrors the "real" stack (i.e. the low-level stack provided by the backend's platform) in that it is a chain of frames. The heap version uses a linked list of normal, garbage-collected structures that each contain the following information:

- a pointer to the previous frame in the call stack;
- an integer that uniquely identifies both the function and the location within the function at the point where the present frame was saved (the Stackless Transformer maintains a single, compact table that maps this index back to the function, and also gives any other information required to resume at this point later);
- any number of extra fields to hold the values of the local variables that were alive at the point where the present frame was saved (the number and types of the extra fields depend on the resume point, which means that each resume point needs in theory a different structure layout. As this explosion is a problem e.g. for GCs that maintain layout information, we mitigate it by type-erasing and sorting the extra fields, which reduces the number of different layout variants to reasonable levels).

[1] The transformation can be expressed in a simple way in the formalism of monads; however, in the present report, we will only present an informal practical description.

In each function, the save-and-resume points are put immediately *after* the calls. In this way, each heap-based frame only needs to remember the values of the local variables that are still used after the call. When a function wishes to save away the current stack, e.g. for the purpose of inspecting it or switching to another previously-saved stack chain, it raises UnwindException, thus causing each of its callers – one after another – to save their state in the heap, by allocating and filling a frame structure. The top-level driver then transfers control back to a saved frame: it can be either the function that raised UnwindException in the first place or an entirely different frame, depending on the effect desired.

The resuming occurs as follows: we never restore a complete heap-based stack chain into the low-level stack; instead, only the top (innermost) frame is restored. This is done by calling the function that was originally executing the saved frame in a special mode, where it jumps to the appropriate resuming point and reloads all relevant local variables from the heap-based frame. The function then continues its normal execution. When it returns, it should conceptually return to its original caller; but the original caller is no longer present in the low-level stack, so it returns to its "real" current caller, i.e. the driver. The driver then emulates the original return. This is done by popping the next frame from the saved frame chain, and resuming this function, in such a way that it will jump just after the call and reload its own local variables. The driver also transports the return value from one function to the next, in a place where the resuming logic of the original caller can find it (unless the callee raised an exception, which the driver catches and re-raises in the caller).

In other words, saved frames are not moved back to the "real" stack before they are actually needed there. This is the reason why we need each function to have resume points just *after* the calls. Another approach has been tried recently (PICOIL) in which the resume points are put *before* the calls, and resuming is done by calling the bottom (outermost) function in such a way that it will resume just before – and perform again – the call to the next function, itself resuming just before – and performing again – the call to the following function, and so on, thus completely moving the saved stack back into the "real" one. For our purposes, the approach we selected instead has two main advantages: it handles stacks whose depth can exceed the limit of the low-level stack, as there is no need for the whole saved frame chain to be present in the low-level stack all at once (see Unbounded Stack below); and it makes switching between coroutine-like microthreads faster. Indeed, in our approach, restoring a frame chain is an amortized constant-time operation – even in the presence of deep stacks – in the common case where the restored frame chain quickly switches away to yet another saved chain (where "quickly" means "before it needs to return to many levels of callers in the current chain"). This in turn means that *saving* the frame states is an amortized constant-time operation (if the current call stack is deep, most of it is likely to be already in the heap). This is particularly important for our main use case of many massively parallel microthreads frequently switching control.

Note also that we have described a model where all functions need special re-entry code, and a special handler for UnwindException after each call. A simple optimization that we have implemented is to avoid inserting resume points at all in the following two circumstances: if the call is a tail call (because the caller's frame is no longer useful when the caller returns); and if we can detect that the call is to a function that cannot possibly raise the special UnwindException. Which functions can or cannot raise UnwindException depend on the desired effect, i.e. on the specific features that we want to enable – or not – with the Stackless Transformation. This is detailed in the following sections. In the common case, many functions do not need any resume point at all, and thus no special re-entry code; particularly the leaves of the call graph and the functions near the leaves, which is good since these are typically the performance-critical functions.

3.2.2 "Unbounded" Stack

This is the only completely transparent application of the Stackless Transformation: it allows a highly recursive RPython program to execute without any other limit than that of the available RAM, even though most common platforms put a much lower limit to their program's low-level stack space. The effect is obtained simply by raising the UnwindException when appropriate, which causes the whole low-level stack to be saved into the heap. Execution then continues normally. As described above, only the innermost frame is restored into the "real" low-level stack. This frame might call many other functions, thus causing the low-level stack to fill up again, at which point another UnwindException can cause this new part of the stack to be saved into the heap as well, and so on. Whenever a function returns past the last frame currently present in the low-level stack, the previous frame is restored and continues its execution.

Frames are restored one by one in a lazy fashion, but it does not mean that the process is continually saving and restoring frames: once a frame has been restored, it stays in the "real" stack until it either returns or needs to be saved again because it triggered so many nested calls that the "real" stack is now full again. One point of view is that the "real" stack (below the currently running frame) is but a cache for the top section of the current heap-based frame chain. We flush the whole "cache" stack into the heap at once when it is full. We load a frame from the heap to the "cache" stack only when it is needed, and it can usually stay in the cache then.

An interesting side-effect is that the heap-based structures are typically more compact than the platform's low-level stack frames, thus making this approach interesting even if only for its memory-compression effect.

The checks "is the low-level stack full?" are automatically inserted at key points in the RPython program. The algorithm that chooses these points ensures that all loops in the call graphs must meet at least one such point. In more precise terms, we pick all the "back edges" in the call graph and insert a check just before the corresponding calls, in the callers. This has proven to be a good way to avoid inserting too many checks, and it also avoids checks on the "fast paths", e.g. the non-recursive case of recursive functions.

3.2.3 Interface to Explicit Stack Manipulations

The Stackless Transformation gives an RPython program explicit access to its own stack via a set of built-in functions, which are implemented as helpers manipulating the UnwindException and its associated data directly.

The present section describes the primitives used for simple stack inspection and coroutine-like manipulation. The next section describes the more advanced primitives that support explicit stack reconstruction.

The simpler built-ins that we implemented are:

- `stack_unwind()` – unwinds the low-level stack by raising UnwindException, but has no effect otherwise. This is what is called when the "is the low-level stack full?" checks mentioned in the Unbounded Stack section return True.
- `stack_frames_depth()` – for testing purposes, returns the number of frames in the current call stack. The implementation simply raises UnwindException and then counts the number of frames by following the linked list in the heap.

To implement coroutine-like behavior, we have implemented a very primitive built-in function `yield_current_frame_to_caller()` and an opaque RPython type `frame_stack_top`. Their purpose is, respectively, to initiate a new coroutine and to reference the current (saved) state an inactive coroutine. They work as follows:

- The built-in function `yield_current_frame_to_caller()` causes the current function's state to be captured in a new `frame_stack_top` object that is returned to the parent. Only one frame, the current one, is captured in this way. The current frame is suspended and the caller continues to run. Note that the caller is only resumed once: when `yield_current_frame_to_caller()` is invoked in the callee. See below.
- A `frame_stack_top` object can be jumped to by calling its `switch()` method with no argument.
- `yield_current_frame_to_caller()` and `switch()` return a new `frame_stack_top` object: the freshly captured state of the caller of the source `switch()` that was just executed, or None in the case described below.
- the function that called `yield_current_frame_to_caller()` also has a normal return statement, like all functions. This statement must return another `frame_stack_top` object. The latter is *not* returned to the original caller; there is no way to return several times to the caller. Instead, it designates the place to which the execution must jump, as if by a `switch()`. The place to which we jump this way will see a None as the source frame stack top.
- every frame stack top must be resumed once and only once. Not resuming it at all causes a leak. Resuming it several times causes a crash.
- a function that called `yield_current_frame_to_caller()` should not raise. It would have no implicit parent frame to propagate the exception to.

The following example prints the numbers from 1 to 7 in order:

```
def g():
    print 2
    frametop_before_5 = yield_current_frame_to_caller()
    print 4
    frametop_before_7 = frametop_before_5.switch()
    print 6
    return frametop_before_7

def f():
    print 1
    frametop_before_4 = g()
    print 3
    frametop_before_6 = frametop_before_4.switch()
    print 5
    frametop_after_return = frametop_before_6.switch()
    print 7
    assert frametop_after_return is None

f()
```

As this example shows, the provided interface is extremely primitive. For practical purposes it must be wrapped, typically in RPython classes, to provide some better-documented and easier to grasp coroutine-like interface. We provide a classical Coroutine wrapper, where each Coroutine instance stores its coroutine's current `frame_stack_top` and wraps the logic to start a new Coroutine. Again, it is interesting to realize that this class is written in plain RPython on top of the `yield_current_frame_to_caller()` primitive; as it is not part of the Stackless Transformation but only an application of it, we will describe it later (see Coroutines). Also, as the RPython interface only depends on this primitive, it will be possible to use a different implementation when targeting a platform that has more flexibility with respect to stack introspection (for example, this should be possible for the Squeak backend or a hypothetical scheme backend) or even an implementation for C that uses platform specific assembly to copy pieces of the low-level stack around.

3.2.4 Resume Points

We have implemented a set of very powerful primitives that allow a program to build from scratch a complete emulated call stack for itself, and "resume" it. It allows a small, independent part of a program to put the rest of the program in some state from which execution can be (re)started. We know of no equivalent of this feature in the literature; the best term we can use is "stack reconstruction". The PyPy interpreter uses this feature to provide Coroutine Pickling.

This feature requires a few *named resume points* to be places in the program. Remember that the Stackless Transformation normally inserts resume points after function calls; named resume points are regular resume points that are associated with a (global) label. They can appear anywhere, but they are most useful just after calls.

A named resume point is created by "calling" the built-in function `resume_point("label", var1, ..., varN)`. The call itself has no effect; it is used to associate the name given by the "label" to this specific resume point. (The association is global and statically precomputed by the Stackless Transformer.) The extra arguments `var1` to `varN` are the variables that are alive across this resume point. They must all be listed; the point of this declaration is to give them an order. Given a named resume point, a heap-based frame can be created from scratch, anywhere in the program, by calling `resume_state_create(back_frame, "label", value1, ..., valueN)`. The "label" specifies where this frame is supposedly suspended; the values are used to fill the local variables `var1` to `varN`.

If a named resume points is just after a call, it should be declared as `resume_point("label", var1, ..., varN, returns=varR)`, where `varR` is the variable holding the return value of the call in the caller. This is the variable that is given special treatment by the top-level driver, that is it is transported from callee to caller when the caller's frame needs to be moved from the heap to the low-level stack. To create from scratch a chain of multiple frames, we call `resume_state_create()` for each frame to create, omitting the value for the `varR`, which will come from the callee's frame real return value.

The built-in function `resume_state_invoke(return_type, frame_top)` invokes a freshly created frame chain. It returns whatever the last (outermost) function returns. For typing reasons, the expected type must be specified in the call to the built-in.

3.2.5 Performance Impact

The main design goal of the Stackless Transformation was to avoid preventing compiler optimizations while still producing portable C code. The latter restriction means that it is impossible

to inspect the run-time stack at the C level (let alone modify it or build it from scratch). A naive approach would have been to store all relevant local variables in a place where the stack inspection code can see them. This can be achieved in basically two ways: keeping the variables as locals (either individually or in a single big local struct) and storing pointers to them in a place where the stack inspection can find them; or not using local variables altogether and working with a custom stack. The latter solution can again be subdivided in two variants: using a custom stack only, or keeping all variables as local but copying them back and forth between the C and the custom stack at specific point, e.g. across function calls.

Note that our custom framework Garbage Collector (described in the Memory Management section below) can be configured to use the latter approach for the stack roots (see Finding roots): in this case, they are saved and restored across each call. We did not try out this approach for the Stackless Transformation, because in that case it would not solve everything – we would still need some way to resume functions at these points. Moreover the run-time overhead would be larger than in the GC case, because we would need to save all data to the custom stack (and not just object references, as in the case of the GC), plus some kind of tag to enable inspection.

The approach that we selected instead is to use local variables only, with special saving and restoring modes where the function copies between these variables and the heap. It means not only that the C compiler sees normal code in the common path, using local variables, but we never takes a pointer to any local variable – the saving code simply reads the locals and save them to the heap, while the restoring code performs the following steps:

- read from the heap
- write to the locals
- jump to the place within the function where execution should be resumed

Because the writes to the locals are immediately followed by the jump, we expect the compiler to be able to produce the following effect: it should perform register allocation as usual in the common path parts of the code, and then implement the restoring writes as a write directly to the register allocated to the local variable *immediately after the jump*. The additional writes in the restoring code, and symmetrically the additional reads in the saving code, can all directly refer to whichever register the variable was allocated to at this point in the normal code of the function, and thus should have no impact on the graph of constraints for register allocation.

To verify the above theory, we have measured the impact (in code size and performance) of enabling the Stackless Transformation. The benchmarks are pystone, which is a port of the Dhrystone 2.0 (DHRY) benchmark to Python which is quite often used in the Python community, and the classical Richards benchmark ported to Python. The increase in code size is due to the new code inserted in each function, most notably the state-saving and state-restoring code. As these benchmarks do not use the new stackless features, the performance overhead comes only from the extra state-checking instruction inserted at the beginning of every function by the transformation, and probably from the cache effects of the increase in code size.

The numbers in parenthesis give the increase when compared to the equivalent non-stackless version. The revision 34906 is from 2006, the 23rd of November. The machine is a Gentoo Linux laptop with an Intel(R) Pentium(R) M processor at 1.73GHz, with 1GB of RAM and 2MB of L2 cache. The term "framework" refers to a pypy-c running our own framework Garbage Collector; the non-framework pypy-c's are using the conservative Boehm GC (BOEHM). The times are in millisecond for one run.

Executable	Code size (a)	pystone	Richards
CPython 2.4.3	916730 (b)	1320	905
pypy-c-34906	1990332	5820	3889
pypy-c-34906-stackless	4700304 (2.36x)	6930 (+19%)	4578 (+17%)
pypy-c-34906-framework	2987948	6430	4618
pypy-c-34906-stackless-framework	6256376 (2.09x)	8121 (+26%)	5927 (+28%)
pypy-c-34906-thread (c)	2000228	6080	3895

(a) Not including data size, link tables, etc.

(b) Estimate only. It is difficult to do a meaningful code size comparison involving CPython because the latter contains a number of dynamically-linked libraries implementing modules, not all of which have a one-to-one correspondence with a PyPy module. (All PyPy modules are built-in.)

(c) For comparison purposes only – no stackless variant available. Our Stackless features are not thread-safe so far (in the sense of OS threads), although this is not a deep problem and should be fixed in a production release.

The table shows that indeed the Stackless Transformation has a reasonable overhead in the range of 17%-28%, which shows that the C compiler is still able to perform most of its optimizations. For comparison, we tried to compile pypy-c in a mode where all local variables are stored in a structure, which is what would occur if we had gone for a different approach relying on pointers to local variables. This pypy-c shows a performance hit of 60% on Pystone and 64% on Richards. Small RPython benchmarks show an even higher slow-down.

We report on the performance of stackless features (as opposed to the performance hit incurred on the whole program even when not using them) in the Microbenchmarks of Stackless Features section.

3.3 Application Level Interface

A stackless PyPy contains a module called `stackless`. The interface exposed by this module has not been much refined, so it should be considered in-flux (as of 2006).

So far, PyPy does not provide support for `stackless` in a threaded environment. This limitation is not fundamental, as previous experience has shown, so supporting this would probably be reasonably easy.

An interesting point is that the same `stackless` module can provide a number of different concurrency paradigms at the same time. From a theoretical point of view, none of above-mentioned existing three paradigms considered on its own is new: two of them are from previous Python work, and the third one is a variant of the classical coroutine. The new part is that the PyPy implementation manages to provide all of them and let the user implement more. Moreover – and this might be an important theoretical contribution of this work – we manage to provide these concurrency concepts in a "composable" way. In other words, it is possible to naturally mix in a single application multiple concurrency paradigms, and multiple unrelated usages of the same paradigm. This is discussed in the Composability section below.

3.3.1 Coroutines

A Coroutine is similar to a very small thread, with no preemptive scheduling. Within a family of coroutines, the flow of execution is explicitly transferred from one to another by the programmer. When execution is transferred to a coroutine, it begins to execute some Python code. When it transfers execution away from itself it is temporarily suspended, and when execution returns to it it resumes its execution from the point where it was suspended. Conceptually, only one coroutine is actively running at any given time (but see Composability below).

The `stackless.coroutine` class is instantiated with no argument. It provides the following methods and attributes:

- `stackless.coroutine.getcurrent()`

  Static method returning the currently running coroutine. There is a so-called "main" coroutine object that represents the "outer" execution context, where your main program started and where it runs as long as it does not switch to another coroutine.

- `coro.bind(callable, *args, **kwds)`

  Bind the coroutine so that it will execute `callable(*args, **kwds)`. The call is not performed immediately, but only the first time we call the `coro.switch()` method. A coroutine must be bound before it is switched to. When the coroutine finishes (because the call to the callable returns), the coroutine exits and implicitly switches back to another coroutine (its "parent"); after this point, it is possible to bind it again and switch to it again. (Which coroutine is the parent of which is not documented, as it is likely to change when the interface is refined.)

- `coro.switch()`

  Suspend the current (caller) coroutine, and resume execution in the target coroutine `coro`.

- `coro.kill()`

  Kill `coro` by sending an exception to it. (At the moment, the exception is not visible to app-level, which means that you cannot catch it, and that `try: finally:` clauses are not honored. This will be fixed in the future.)

Example

Here is a classical producer/consumer example: an algorithm computes a sequence of values, while another consumes them. For our purposes we assume that the producer can generate several values at once, and the consumer can process up to 3 values in a batch – it can also process batches with less than 3 values without waiting for the producer (which would be messy to express with a classical Python generator).

```
def producer(lst):
    while True:
        ...compute some more values...
        lst.extend(new_values)
        coro_consumer.switch()
```

```
def consumer(lst):
    while True:
        # First ask the producer for more values if needed
        while len(lst) == 0:
            coro_producer.switch()
        # Process the available values in a batch, but at most 3
        batch = lst[:3]
        del lst[:3]
        ...process batch...

# Initialize two coroutines with a shared list as argument
exchangelst = []
coro_producer = coroutine()
coro_producer.bind(producer, exchangelst)
coro_consumer = coroutine()
coro_consumer.bind(consumer, exchangelst)

# Start running the consumer coroutine
coro_consumer.switch()
```

3.3.2 Tasklets and Channels

The `stackless` module also provides an interface that is roughly compatible with the interface of the `stackless` module in Stackless Python: it contains `stackless.tasklet` and `stackless.channel` classes. Tasklets are also similar to microthreads, but (like coroutines) they don't actually run in parallel with other microthreads; instead, they synchronize and exchange data with each other over Channels, and these exchanges determine which Tasklet runs next.

For usage reference, see the documentation on the Stackless Python website (SLP).

Note that Tasklets and Channels are implemented at application-level in *pypy/lib/stackless.py* on top of coroutines. You can refer to this module for more details and API documentation.

The stackless.py code tries to resemble the stackless C code as much as possible. This makes the code somewhat unpythonic.

Bird's eye view of tasklets and channels

Tasklets are a bit like threads: they encapsulate a function in such a way that they can be suspended/restarted any time. Unlike threads, they won't run concurrently, but must be cooperative. When using stackless features, it is vitally important that no action is performed that blocks everything else. In particular, blocking input/output should be centralized to a single tasklet.

Communication between tasklets is done via channels. There are three ways for a tasklet to give up control:

1. call `stackless.schedule()`
2. send something over a channel

3. receive something from a channel

A (live) tasklet can either be running, waiting to be scheduled, or blocked by a channel.

Scheduling is done in a strictly round-robin manner. A blocked tasklet is removed from the scheduling queue and reinserted when it becomes unblocked.

Example

Here is a many-producers many-consumers example, where any consumer can process the result of any producer. For this situation we set up a single channel where all producer send, and on which all consumers wait:

```
def producer(chan):
    while True:
        chan.send(...next value...)

def consumer(chan):
    while True:
        x = chan.receive()
        ...do something with x...

# Set up the N producer and M consumer tasklets
common_channel = stackless.channel()
for i in range(N):
    stackless.tasklet(producer, common_channel)()
for i in range(M):
    stackless.tasklet(consumer, common_channel)()

# Run it all
stackless.run()
```

Each item sent over the channel is received by one of the waiting consumers; which one is not specified. The producers block until their item is consumed: the channel is not a queue, but rather a meeting point which causes tasklets to block until both a consumer and a producer are ready. In practice, the reason for having several consumers receiving on a single channel is that some of the consumers can be busy in other ways part of the time. For example, each consumer might receive a database request, process it, and send the result to a further channel before it asks for the next request. In this situation, further requests can still be received by other consumers.

3.3.3 Greenlets

A Greenlet is a kind of primitive Tasklet with a lower-level interface and with exact control over the execution order. Greenlets are similar to Coroutines, but have a slightly different interface: greenlets put a greater emphasis on a tree structure – the various greenlets of a program form a precise tree, which fully determines their order of execution.

For usage reference, see the documentation of the Py lib greenlets (PYLIB). The PyPy interface is identical. You should use `py.magic.greenlet` instead of `stackless.greenlet` directly, because the Py lib can now give you the latter when you ask for the former on top of PyPy.

PyPy's greenlets do not suffer from the cyclic GC limitation that the CPython greenlets have: greenlets referencing each other via local variables tend to leak on top of CPython (where it is mostly impossible to do the right thing). It works correctly on top of PyPy.

3.3.4 Coroutine Pickling

Coroutines and tasklets can be pickled and unpickled, i.e. serialized to a string of bytes for the purpose of storage or transmission. This allows "live" coroutines or tasklets to be made persistent, moved to other machines, or cloned in any way. The standard `pickle` module works with coroutines and tasklets (at least in a translated `pypy-c`; unpickling live coroutines or tasklets cannot be easily implemented on top of CPython).

To be able to achieve this result, we have to consider many objects that are not normally picklable in CPython. Here again, the Stackless Python implementation has paved the way, and we follow the same general design decisions: simple internal objects like bound method objects and various kinds of iterators are supported; frame objects can be fully pickled and unpickled (by serializing a reference to the bytecode they are running in addition to all the local variables). References to globals and modules are pickled by name, similarly to references to functions and classes in the traditional CPython `pickle`.

The "magic" part of this process is the implementation of the unpickling of a chain of frames. At any point in time, a chain of Python-level frames corresponds to a chain of interpreter-level frames (e.g. C frames in pypy-c), where each single Python-level frame corresponds to one or a few interpreter-level frames – depending on the length of the interpreter-level call chain from one bytecode evaluation loop to the next (recursively invoked) one.

This means that it is not sufficient to simply create a chain of Python frame objects in the heap of a process before we can resume execution of these newly built frames. We must recreate a corresponding chain of interpreter-level frames. To this end, we have inserted a few *named resume points* (see the Resume Points section) in the Python interpreter of PyPy. This is the motivation for implementing the interpreter-level primitives `resume_state_create()` and `resume_state_invoke()`, the powerful interface that allows an RPython program to artificially rebuild a chain of calls in a reflective way, completely from scratch, and jump to it.

Example

Consider a program which contains a part performing a long-running computation:

```
def ackermann(x, y):
    if x == 0:
        return y + 1
    if y == 0:
        return ackermann(x - 1, 1)
    return ackermann(x - 1, ackermann(x, y - 1))
```

By using pickling, we can save the state of the computation while it is running, for the purpose of restoring it later and continuing the computation at another time or on a different machine. However, pickling does not produce a whole-program dump: it can only pickle individual coroutines. This means that the computation should be started in its own coroutine:

```
# Make a coroutine that will run 'ackermann(3, 8)'
coro = coroutine()
```

```
coro.bind(ackermann, 3, 8)

# Now start running the coroutine
result = coro.switch()
```

The coroutine itself must switch back to the main program when it needs to be interrupted (we can only pickle suspended coroutines). Due to current limitations this requires an explicit check in the `ackermann()` function:

```
def ackermann(x, y):
    if interrupt_flag:      # test a global flag
        main.switch()       # and switch back to 'main' if it is set
    if x == 0:
        return y + 1
    if y == 0:
        return ackermann(x - 1, 1)
    return ackermann(x - 1, ackermann(x, y - 1))
```

The global `interrupt_flag` would be set for example by a timeout, or by a signal handler reacting to Ctrl-C, etc. It causes the coroutine to transfer control back to the main program. The execution comes back just after the line `coro.switch()`, where we can pickle the coroutine if necessary:

```
if not coro.is_alive:
    print "finished; the result is:", result
else:
    # save the state of the suspended coroutine
    f = open('demo.pickle', 'w')
    pickle.dump(coro, f)
    f.close()
```

The process can then stop. At any later time, or on another machine, we can reload the file and restart the coroutine with:

```
f = open('demo.pickle', 'r')
coro = pickle.load(f)
f.close()
result = coro.switch()
```

Limitations

Coroutine pickling is subject to some limitations. First of all, it is not a whole-program "memory dump". It means that only the "local" state of a coroutine is saved. The local state is defined to include the chain of calls and the local variables, but not for example the value of any global variable.

As in normal Python, the pickle will not include any function object's code, any class definition, etc., but only references to functions and classes. Unlike normal Python, the pickle contains frames. A pickled frame stores a bytecode index, representing the current execution position. This means that the user program cannot be modified *at all* between pickling and unpickling!

On the other hand, the pickled data is fairly independent from the platform and from the PyPy version.

Pickling/unpickling fails if the coroutine is suspended in a state that involves Python frames which were *indirectly* called. To define this more precisely, a Python function can issue a regular function or method call to invoke another Python function – this is a *direct* call and can be pickled and unpickled. But there are many ways to invoke a Python function indirectly. For example, most operators can invoke a special method `__xyz__()` on a class, various built-in functions can call back Python functions, signals can invoke signal handlers, and so on. These cases are not supported yet.

3.3.5 Coroutine Cloning

In theory, coroutine pickling is general enough to allow coroutines to be *cloned* within a process; i.e. from one suspended coroutine, a copy can be made – simply by pickling and immediately unpickling it. Both the original and the copy can then continue execution from the same point on. Cloning gives much of the expressive power of full *continuations*.

However, pickling has several problems in practice (besides a relatively high overhead). It is not a completely general solution because not all kinds of objects can be pickled; moreover, which objects are pickled by value or by reference only depends on the type of the object. For the purpose of cloning, this means that coroutines cannot be pickled/unpickled in all situations, and even when they can, the user does not have full control over which of the objects currently reachable from a coroutine will be duplicated, and which will be shared with the original coroutine.

For this reason, we implemented a direct cloning operation. After some experiments, we determined that the following behavior is usually considered correct: when cloning a coroutine C, we duplicate exactly those objects that were created by C (i.e. while C was running). The objects not created by C (e.g. pre-existing, or created outside) are not duplicated, but directly shared between C and its new copy. This heuristic generally matches the intuition that the objects created by C are also the ones "owned" by C, in the sense that if C is cloned, the clone needs its own copy – to avoid mutating the same shared object in conflicting ways. Conversely, objects of a more "global" nature, like modules and functions, which are typically created before the coroutine C started, should not be duplicated; this would result in unexpectedly invisible side-effects if they are mutated by the clone of C.

The implementation of the above heuristic is based on support from the garbage collector. For this reason, it is only available if both stackless and our own framework GC are compiled together in pypy-c:

```
cd pypy/translator/goal
python translate.py --stackless --gc=framework
```

In this mode, our garbage collector is extended to support the notion of "pool": a pool is a linked list of allocated objects. All objects allocated go to a "current" pool. When the stackless module switches execution between two ClonableCoroutine objects, it switches the GC's current pool as well, so that the allocated objects end up segregated by coroutine. Cloning is implemented by another GC extension which makes byte-level copies of allocated objects. Pointers to objects inside the current pool cause the target objects to be recursively copied; pointers outside the current pool are simply shared.

Two interfaces are available to clone a coroutine:

- `coro.clone()`

  Clones a suspended coroutine, returning the new copy.

- `fork()`

  This global function (from the `_stackless` module) clones the currently running coroutine, returning the new copy – which, in this case, is a child of the original coroutine (i.e. its parent is the original coroutine). When the parent or any other coroutine eventually switches to the child, execution appears to return from the same `fork()` call, this time with a `None` return value (this is similar to the UNIX `fork` system call).

Example

Forking is a natural way to implement backtracking. Consider a coroutine other than the main one (the main coroutine cannot be cloned or forked) which issues a call to the following function:

```
def zero_or_one():
    subcoro = fork()
    if subcoro is not None:
        try:
            subcoro.switch()      # in the parent: run the child first
        except Fail:
            pass
        return 1                  # then proceed with answer 1
    else:
        return 0                  # in the child: answer 0
```

It clones the current coroutine, and switches to the new child. In the latter, `zero_or_one()` returns 0, so the caller sees the value 0 first. But at any later point in time (even from a completely unrelated place, as long as it is still in the same coroutine) the caller may decide that 0 is not a suitable value, and backtrack and try again with the value 1. To do so, it simply raises a Fail exception. The exception stops the child coroutine, which comes back to the parent, where the Fail is caught and the value 1 is returned from `zero_or_one()`.

The tests in *pypy/module/_stackless/test/test_clonable.py* contain an example using exactly the above function to search for a sequence of zeroes and ones that has a specific property, by trial-and-error. This allows a style that looks similar to what is traditionally reserved to logic programming languages.

3.3.6 Composability

Although the concept of coroutines is far from new, they have not been generally integrated into mainstream languages, or only in limited form (like generators in Python and iterators in C#). We can argue that a possible reason for that is that they do not scale well when a program's complexity increases: they look attractive in small examples, but the models that require explicit switching, by naming the target coroutine, do not compose naturally. This means that a program that uses coroutines for two unrelated purposes may run into conflicts caused by unexpected interactions.

To illustrate the problem, consider the following example. First, a simple usage of coroutine:

```
main_coro = coroutine.getcurrent()     # the main (outer) coroutine
```

```
data = []

def data_producer():
    for i in range(10):
        # add some numbers to the list 'data' ...
        data.append(i)
        data.append(i * 5)
        data.append(i * 25)
        # and then switch back to main to continue processing
        main_coro.switch()

producer_coro = coroutine()
producer_coro.bind(data_producer)

def grab_next_value():
    if not data:
        # put some more numbers in the 'data' list if needed
        producer_coro.switch()
    # then grab the next value from the list
    return data.pop(0)
```

Every call to grab_next_value() returns a single value, but if necessary it switches into the producer function (and back) to give it a chance to put some more numbers in it.

Now consider a simple reimplementation of Python's generators in term of coroutines:

```
def generator(f):
    """Wrap a function 'f' so that it behaves like a generator."""
    def wrappedfunc(*args, **kwds):
        g = generator_iterator()
        g.bind(f, *args, **kwds)
        return g
    return wrappedfunc

class generator_iterator(coroutine):
    def __iter__(self):
        return self
    def next(self):
        self.caller = coroutine.getcurrent()
        self.switch()
        return self.answer

def Yield(value):
    """Yield the value from the current generator."""
    g = coroutine.getcurrent()
    g.answer = value
    g.caller.switch()

def squares(n):
    """Demo generator, producing square numbers."""
    for i in range(n):
        Yield(i * i)
squares = generator(squares)
```

```
for x in squares(5):
    print x       # this prints 0, 1, 4, 9, 16
```

Both these examples are attractively elegant. However, they cannot be composed. If we try to write the following generator:

```
def grab_values(n):
    for i in range(n):
        Yield(grab_next_value())
grab_values = generator(grab_values)
```

then the program does not behave as expected. The reason is the following. The generator coroutine that executes `grab_values()` calls `grab_next_value()`, which may switch to the `producer_coro` coroutine. This works so far, but the switching back from `data_producer()` to `main_coro` lands in the wrong coroutine: it resumes execution in the main coroutine, which is not the one from which it comes. We expect `data_producer()` to switch back to the `grab_next_values()` call, but the latter lives in the generator coroutine `g` created in `wrappedfunc`, which is totally unknown to the `data_producer()` code. Instead, we really switch back to the main coroutine, which confuses the `generator_iterator.next()` method (it gets resumed, but not as a result of a call to `Yield()`).

As part of trying to combine multiple different paradigms into a single application-level module, we have built a way to solve this problem. The idea is to avoid the notion of a single, global "main" coroutine (or a single main greenlet, or a single main tasklet). Instead, each conceptually separated user of one of these concurrency interfaces can create its own "view" on what the main coroutine/greenlet/tasklet is, which other coroutine/greenlet/tasklets there are, and which of these is the currently running one. Each "view" is orthogonal to the others. In particular, each view has one (and exactly one) "current" coroutine/greenlet/tasklet at any point in time. When the user switches to a coroutine/greenlet/tasklet, it implicitly means that he wants to switch away from the current coroutine/greenlet/tasklet *that belongs to the same view as the target*.

The precise application-level interface has not been fixed yet; so far, "views" in the above sense are objects of the type `stackless.usercostate`. The above two examples can be rewritten in the following way:

```
producer_view = stackless.usercostate()    # a local view
main_coro = producer_view.getcurrent()     # the main (outer) coroutine
...
producer_coro = producer_view.newcoroutine()
...
```

and:

```
generators_view = stackless.usercostate()

def generator(f):
    def wrappedfunc(*args, **kwds):
        g = generators_view.newcoroutine(generator_iterator)
        ...

        ...generators_view.getcurrent()...
```

Then the composition `grab_values()` works as expected, because the two views are independent. The coroutine captured as `self.caller` in the `generator_iterator.next()` method is the main coroutine of the `generators_view`. It is no longer the same object as the main coroutine of the `producer_view`, so when `data_producer()` issues the following command:

```
main_coro.switch()
```

the control flow cannot accidentally jump back to `generator_iterator.next()`. In other words, from the point of view of `producer_view`, the function `grab_next_value()` always runs in its main coroutine `main_coro` and the function `data_producer` runs in its coroutine `producer_coro`. This is the case independently of which `generators_view`-based coroutine is the current one when `grab_next_value()` is called.

Only code that has explicit access to the `producer_view` or its coroutine objects can perform switches that are relevant for the generator code. If the view object and the coroutine objects that share this view are all properly encapsulated inside the generator logic, no external code can accidentally temper with the expected control flow any longer.

In conclusion: we will probably change the app-level interface of PyPy's stackless module in the future to not expose coroutines and greenlets at all, but only views. They are not much more difficult to use, and they scale automatically to larger programs.

3.3.7 Microbenchmarks of Stackless Features

We performed microbenchmarks of stackless features on the same executables as the ones listed in the Performance Impact section. Refer to that section for a description of the executable names and the benchmarking conditions. All times below are in microsecond per operation.

The first benchmark is measuring *Coroutine switches*. It switches back and forth between the main coroutine and another coroutine whose application-level stack depth is fixed at *N*. The results show that the time taken does not depend on *N*, i.e. our `pypy-c` does indeed not save and restore the whole stack of the coroutine at each switch.

Stack depth	**pypy-c** 34906 stackless	**pypy-c** 34906 stackless framework
0	6.72	7.00
10	6.71	6.82
20	6.70	6.89
30	6.29	6.88
40	6.32	6.95
50	6.44	6.99
60	6.58	6.93
70	6.78	7.01
80	6.96	6.61
90	7.08	6.67
100	7.31	6.48

The next benchmark is similar, but uses the greenlet interface. We can now compare the results with the same tests running on top of CPython and using the Py lib greenlets (PYLIB). The CPython timings are linear in *N*, reflecting the fact that each switch copies the whole stack of the greenlet in and out of the machine stack in the Py lib greenlets.

Stack depth	**CPython** 2.4.3	**pypy-c** 34906 stackless	**pypy-c** 34906 stackless framework
0	1.81	14.41	16.30
10	2.82	14.49	16.92
20	4.10	14.58	16.48
30	8.71	14.53	16.18
40	12.03	14.72	16.08
50	14.22	14.72	16.02
60	16.52	15.09	15.77
70	18.83	15.10	15.66
80	21.16	15.51	16.04
90	23.52	14.66	15.49
100	25.93	15.16	15.63

The rest of the tests are grouped in a single table, as they do not involve a varying parameter.

Benchmark description *times in microsecond, (mem. usage in bytes)*	**CPython** 2.4.3	**pypy-c** 34906 stackless	**pypy-c** 34906 stackless framework	**pypy-c** 34906 thread
Calling an empty function with one argument (for reference)	0.37	5.10	7.12	4.40
Instantiate and execute a trivial coroutine		35.55 (404b/dead coroutine)	38.03 (1268b/dead coroutine)	
Instantiate and execute a trivial greenlet	3.10	55.49 (990b/dead greenlet)	59.48 (2164b/dead greenlet)	
Instantiate, switch in and switch out of a coroutine		58.88 (1748b/live coroutine)	68.02 (2600b/live coroutine)	
Instantiate, switch in and switch out of a greenlet	9.37	66.86 (2136b/live greenlet)	72.57 (3424b/live greenlet)	
Instantiate a thread and wait until it finishes (but see below)	13.26 (15.8kb)			36.39 (25.6kb)
Pickling a coroutine		7320	9050	
Unpickling a coroutine		10280	12250	
Cloning a coroutine			67.89	

Forking a clonable coroutine and stopping the parent			58.39	

In general, when comparing the different columns with each other, remember to take into account the fact that these different executables have different performance to start with. Their relative performance when executing non-stackless-based code can be found in Performance Impact section.

Pickling and unpickling are implemented mostly at application-level, which explains their slowness. We still believe that coroutine cloning can be much faster than any pickling/unpickling-based implementation.

The OS thread line is included for comparison, but such numbers can only be obtained by measuring a thread start *and* stop. It is not possible to start many threads without stopping them. Indeed, under a default Gentoo Linux installation, each thread uses 8.4MB of virtual memory, and thus on a 32-bit machine the default address space of 3GB is exhausted after only 382 threads. (Each thread only consumes on the order of 15-25kb of resident memory, though, which is the number that we reported in the table.)

3.4 Practical Uses of Massive Parallelism

The style of massive parallelism offered by the facilities presented in this document are most suited to applications with large numbers – on the order of thousands or more – of at least somewhat independent, simultaneously active conceptual threads of control.

Two kinds of application with these demands are simulations with large numbers of agents and network servers with large numbers of concurrent clients.

A relatively new application that combines aspects of both of these situations is that of the Massively Multiplayer Online Role-Playing Game, where thousands or tens of thousands of players interact in an imaginary universe. This particular application is relevant to this document, because CCP Games of Reykjavík, Iceland used (and sponsored the development of) Stackless Python, the direct forerunner of this work, to create their massively multiplayer game EVE Online.

Hilmar V. Petursson, CEO of CCP Games, said:

> "When embarking on the creation of EVE Online, a single shared persistent world we realized two things: We could not given constraints of time and commercial reality do this in a compiled language and we needed innovative concurrency control for such a large scale shared state simulation across tens of thousands of CPUs (EVE Clients included). After many experiments with various combination of existing scripting languages and NT fibers we arrived at Stackless Python. Stackless Python offered us the power of Python coupled with a vastly superior concurrency control mechanism over anything we had seen before, first as continuations and later with an innovative channel based API. CCP's commercial success today is built on the single decision of selecting Stackless Python as our foundation."

When faced with a problem requiring a large degree of concurrency, there are broadly speaking four approaches available in mainstream programming languages today:

- use the preemptive, shared-memory threads supplied by the operating system.

- use some form of cooperative user-land or "green" threads.
- use an event based framework such as (SEDA) or (Twisted).
- a message-passing model such as described by CSP or used by Erlang.

The main disadvantage of the first approach is that of overhead, both in space and time; although advances have been made recently in reducing this it is still significant, especially when it comes to simultaneously active threads. In addition, using fully pre-emptive threads introduces the additional overhead of requiring locking or extremely careful programming to avoid the usual problems of deadlocks, livelocks and race conditions associated with threaded programming. The unique advantage of operating system level threads is exploiting multiple processor cores which are finally becoming commonplace – but for optimal use of N cores, the number of operating system threads should be N or $N + 1$, so using such a thread for each agent in a thousand-agent simulation is not a good way of achieving optimal performance.

PyPy's "tasklets and channels" interface is inspired by Limbo (LIMBO) programming language, and provides a CSP-like interface on top of a cooperative thread like model.

The main advantage of the first two approaches as opposed to the latter two is that they allow the conceptual flow of control to more closely resemble that of the mental model of the programmer (von-Behren-et-al). A commonly presented example is that of a web application that receives input by presenting the user with a series of forms (Seaside), (PLTWebserver).

This last example gives a motivation for another feature of PyPy: coroutine pickling. To avoid "breaking the back button" or allow bookmarking, these application servers must be able to resume computation from a previous point. "Full" (i.e. not one-shot) continuations allow for this by mapping a URL to a particular continuation as a continuation can be resumed multiple times, but for a bookmark to survive a server restart, this continuation must be persisted somehow.

It seems that the use of coroutine pickling for persistence may not be practical, as the pickled execution state necessarily depends on the finest details of the code being executed (AB-OC), but it still has uses in checkpointing, migrating computation in a cluster and the flexibility that this facility requires and implies surely has applications that cannot be imagined today.

A final use for massive parallelism is PyPy's approach to logic programming, which implements choice by cloning coroutines and having each copy explore a different possibility. This implementation makes two particular demands on the coroutine implementation:

1. The requirement of being able to clone the coroutine implies that every detail of the coroutine implementation must be under the control of the PyPy runtime (or at least known: while cloning an operating system thread might be possible on a given platform, it must surely be impossible to do portably), and
2. As the number of pending choices is effectively unbounded, the coroutines must be lightweight so that the copies representing as-yet unexplored choices do not consume excessive system resources.

PyPy's implementation ably satisfies both these requirements.

3.5 Related Work

Our task of implementing non-standard control flow in a potentially uncooperative environment is similar to that facing implementers of the Scheme (R5RS) programming language.

Our approach has many aspects in common with the less theoretical parts of Pettyjohn et al. (CGSI), in particular with Section 4.2 which illustrates how to implement "Continuation Marks" (MAS) on the CLI virtual machine. This in turn is similar to the approaches taken by Sekiguchi et al. (PICOIL) (probably the most directly similar work to ours) and Tao (PMTPM).

The level of performance and code size overhead is comparable to that reported in the above work.

There are a few differences between our work and the above:

- Our transformation, by the nature of being part of a globally analysing compilation process, operates at a different, arguably more natural, level (the above papers describe transformations of either source code or byte code).
- Relatedly, when PyPy's Standard Interpreter is translated with these features enabled, the application-level developer does not need to worry about mixing use of stackless-enabled features with use of Python builtins – all interpreter functions that can recursively lead to the execution of application-level code will be stackless transformed and so will know how to save their state. This differs from some of the above work, and also from the original Stackless Python. The only case in which the application level programmer needs to be careful is when using a third party library that features callbacks, something that is not yet supported.
- The manner in which stack frames are lazily restored to the low-level stack is not included in any of the above papers (though our approach here can be seen as a much simplified version of that of Dybvig et al.(RCPFCC)_).
- We use a different primitive (`yield_current_frame_to_caller`, see above) to build our more usable abstractions upon.
- We have made no effort to formally prove the correctness of our transform, as in the context of our work we have primarily an implementation focus. In the basic Stackless Transformation – as usual in our approach – the underlying theory is reasonably straightforward; for the most novel aspects, though, we plan to eventually formalize and publish the results.

From a less academic point of view, our work is obviously much inspired by the existing Stackless Python (SLP) project. The first released version of Stackless Python (retrospectively named "Stackless 1") amounted to a rewrite of parts of the interpreter in a style where the bytecode interpreter does not call itself recursively when it interprets a function call, but rather pushes the current execution into a heap structure and begins interpreting the called function (this can be seen as a form of Continuation Passing Style). This version therefore never uses the C stack (at least in the simple case of Python calling Python) and this was the inspiration for the name "Stackless", a name which does not strictly apply to any of its successors. Stackless 1 exposed full continuations to the programmer, although most users used a widely distributed third party module implementing lightweight threads on top of these continuations.

This first version of Stackless Python required extensive modifications to support continuing past anything other than a direct Python to Python call, which made tracking the development of the Python language extremely difficult. This lead to the development of "Stackless 2" which used small pieces of platform-dependent assembly to move the low-level stack around as necessary. This version of stackless was also the first to introduce the tasklets and channels model of concurrency, modelled after the Limbo language (LIMBO).

The most recent version of Stackless Python, "Stackless 3.0", combines both these approaches, avoiding the use of the low-level stack when it can and falling back to stack copying when

it cannot. It is the first version to support the pickling of tasklets, with the limitation that only stack-less tasklets can be pickled.

Armin Rigo took the assembly written for Stackless 2 and used it to build a extension module for CPython that could implement a similar (but not the same) control flow control mechanism; this is where the greenlet application level interface design comes from.

While the implementation of PyPy's core stackless features does not resemble that of any version of Stackless Python, the interface ideas and higher level code have been incorporated into PyPy. This is where the tasklet application-level interface comes from, together with other aspects like the details of tasklet pickling. The coroutine interface is intended to be as classical as possible. As far as we know, the ideas and the design sketched in the Composability section above are new.

The fact that this kind of state-capturing approach allows a form of coroutine pickling is far from a novel observation, and indeed this is the motivation of the work of Tao (PMTPM) referenced above.

The approach of using the GC to allow a form of coroutine cloning is, to the best of our knowledge, novel.

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 strategy to use, how certain objects are implemented and so on. This is consistent with the view that the source of our Standard Interpreter is 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 "woven" into the interpreter during the translation process (see (D05.3) and (D05.4) for more details about translation aspects).

The usefulness of the various optimizations and the performance of the various garbage collectors were 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, 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 – this is described in the report for WP06. 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 – this is the subject of the current section.

One of the biggest sources of inefficiencies is the large amount of time spent managing memory. This is caused by the fact RPython itself has automatic memory management and idiomatic RPython programs allocate memory at an extremely fast rate. Therefore many optimizations try to help in this area by reducing the number of objects being allocated or by trying to predict the lifetime of an object.

In general PyPy proved to be a good environment to work on optimizations. Due to the multistage approach chosen where flow graphs are moved towards the abstraction level of the target language in several steps it is possible to choose exactly the right level to perform 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 perform many 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. Some more optimizations are enabled by the fact that they have full information about types, at all levels.

4.1.1 Function Inlining

To reduce the overhead of the many function 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 call-site and inserts a copy of the flow graph into the graph of the calling function, renaming occurring variables as appropriate. Clearly, this optimization only applies to call sites whose target function can be statically known. Our primary motivation for this optimization is to enable further optimizations (notably malloc removal, described below); by itself, its interest is limited by the fact that most compilers for the low-level languages we target can perform inlining themselves.

A drawback of our exception model is that it makes it difficult to inline a function inside another if the original call site is surrounded by a `try: ... except: ...` guard. More precisely, although in Python source an exception handler surrounds a region of code, we chose a more explicit representation for our flow graphs: inside a single function, all the jumps from each potentially-raising operation to the corresponding handler are explicit. In a call site surrounded by an exception handler, a single such link is enough – for the call operation; but after inlining, the call is replaced with many operations. A comprehensive treatment of this case would require checking again each inlined operation for which exceptions it can raise, and rewiring it if needed. We settled for a compromise, where a simple heuristic tries to detect whether an exception raised by the called function directly 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 the more complicated cases, inlining is not performed.

In addition we also implemented heuristics to decide which function to inline where. For this purpose we assign to 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 occurring 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 by a guess of how often the operation is executed in the presence of branches and loops. To arrive at this guess we make the simple (and probably naive) assumption that both paths of every branch are equally likely to occur, with the exception of branches involved in loops, where we consider staying in the loop to be more likely than exiting it. This leads to a system of linear equations, whose solution is our "median execution cost".

After the size estimate for all functions has been determined, functions are inlined into their call-sites, 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. Special care was taken to make the inlining process deterministic, e.g. the same functions should be 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 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. As no other optimizations are performed these numbers probably show that – at

least in the case of small benchmarks – the GCC compiler itself could benefit from performing more aggressive inlining.

4.1.2 Malloc Removal

RPython is a garbage collected language and uses memory allocation freely in many places. This leads to memory being allocated in places where a more traditional language would not. 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 object are allocated on the heap: The range object, the iterator for the range object and the StopIteration instance which ends the loop.

After a small bit of inlining all three objects are never passed as arguments to another function and are 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 inlining. 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 is also reached by a variable created in any other way (result of a function call, another malloc, etc.). 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.

On the default build of PyPy's Standard Interpreter the malloc removal phase removes about 20% of the malloc operations. The speedup when using the malloc removal optimization together with inlining compared to just using inlining is as follows: 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 detect 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 with very sophisticated algorithms; see for example (BLANCHET99), (CHOI99).

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 controlled parts of the stack of frames. For this we chose a naive, pessimistic approach (FRANZ02) which was quite easy to implement, for the goal of evaluating 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, in a fix-point process. 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 with this variable are marked to be escaping.

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 optimizations. 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 many fewer 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.

4.1.4 Pointer Tagging

The following optimization is more specifically targeted to virtual machines for languages with boxed primitive types. It decreases the memory allocation overhead incurred by boxing. For example, when interpreting code performing many integer calculations, each intermediate result typically requires an allocation; this is the kind of allocation that can be avoided by the common technique of tagged pointers.

Due to machine word alignment constraints of modern architectures, the address of the start of an object is always an even number (and more likely a multiple of 4, at least). That means that no valid pointer can have its lowest bit set; the presence of that bit can be used to distinguish between valid pointers and pseudo-pointers that merely encode information in their own bit pattern. This can be used to represent small enough integers by shifting them one bit to the left and setting the lowest bit. These integers 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 indistinguishable (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 is an orthogonal, optional 156-line module in the translation toolchain. The interesting part is that the RPython source code contains a regular "boxed small integer" class. The class has a single (read-only) field storing the integer, but also many methods, some of which override methods from its base class in the usual way. It is only during translation that instances of this class become represented as a tagged pointer. Method calls do not need special support, because they are based on the operation of reading the dynamic type of the instance and finding the method pointer in the vtable; we only had to modify this "read dynamic type" operation to check the tag bit of the pointer. When the tag bit is set, the operation returns a pointer to the (normal) vtable of the particular "boxed small integer" class.

We got interesting speed improvements by combining the above transformation with a simple constant folding optimization. The latter 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 effort 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. It would only take us a few extra hundreds of lines to implement tagging for additional object types, e.g. for very short strings or "round enough" floats.

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).

4.2 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 description) assume that memory will be automatically managed in the RPython program. When targeting 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) and (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 implementers 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 at the very least does not lead to crashes or similar.

4.2.1 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.

4.2.2 Using The Conservative Boehm Collector

The simplest way to transform flow graphs to have memory management is to use the Boehm-Demers-Weiser 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 explicitly 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 bit-wise 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.

4.2.3 Reference Counting

A relatively simple memory management strategy is that of reference counting (see (GCSURV) 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 necessary. This makes the executable both a lot bigger and also adds a lot of unnecessary 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, which is that reference counting is not such a bad choice in that specific context.

4.2.4 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 GCs 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 management toolkit (MMTK) that is used to implement the garbage collectors of the Jikes RVM (JIKES) in Java. Much more work and fine-tuning has gone 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 it 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 oper-

ations 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.

More recently we have moved away from the byte-level memory simulator. Instead, we have extended the approach that we use in the rest of the translation toolchain, which is to have an implementation of C structures and pointers as Python objects. We have added an implementation of address-like Python objects. They can refer to the start of a structure, to a field within it, to a header before it, or to an array or a specific array item. Address offsets are represented purely symbolically – they do not have a concrete integer value in the world of the Python objects. Operations like adding an address and an offset are implemented at the symbolic level. Other Python objects represent arenas, from which objects can be allocated, and which for testing purposes are considered to be exhausted after some number of allocations. This allows us to write and test "type-safe" GCs.

Explicitly managed objects

The data structures 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. Instances of such classes are not managed by the garbage collector but have to be explicitly freed. When running with simulating Python objects, accessing a freed object gives again a useful error message, easing the debugging on top of CPython.

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 - i.e. 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 accessible 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.

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. (The length is used for arrays.)

`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 a 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.

4.2.5 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. It is only if the object is dead at the *next* collection that it 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 resurrect the object being finalized. In CPython, finalizers are invoked arbitrarily often - a fact that has been a source for interpreter crashes, and that some CPython developers thought about changing (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.

4.2.6 Performance & Future work

The following table lists performance measurements for a range of PyPy builds, compared to CPython. The measurements were done on a AMD Opteron 242, 1603 MHZ, v1024 KB cache size with 4GB of RAM. The pystone column lists the time in milliseconds for a full run of the Pystone benchmark, and 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 removal and the Boehm GC. When not specified, the Boehm GC is used (except for pypy-c-0.7, which uses reference counting). This table is built from the 12th of July revision of PyPy.

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.

Another thing to work on is to implement support for 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). While the GC itself, in its simple form, should be relatively little work, some care is needed to still have a consistent identity hash function after objects were moved. We are also facing the issue that moving objects (e.g. string objects) would break the current implementation of `rctypes`, the part of PyPy that deals with calls to external C functions.

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.

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.

4.3 Related Work

The optimization techniques of inlining, malloc removal, escape analysis and stack allocation are all classical. The pointer tagging scheme itself is classical as well. To the best of our knowledge, though, the automatic insertion of pointer tagging as a translation-time optimization is novel in PyPy.

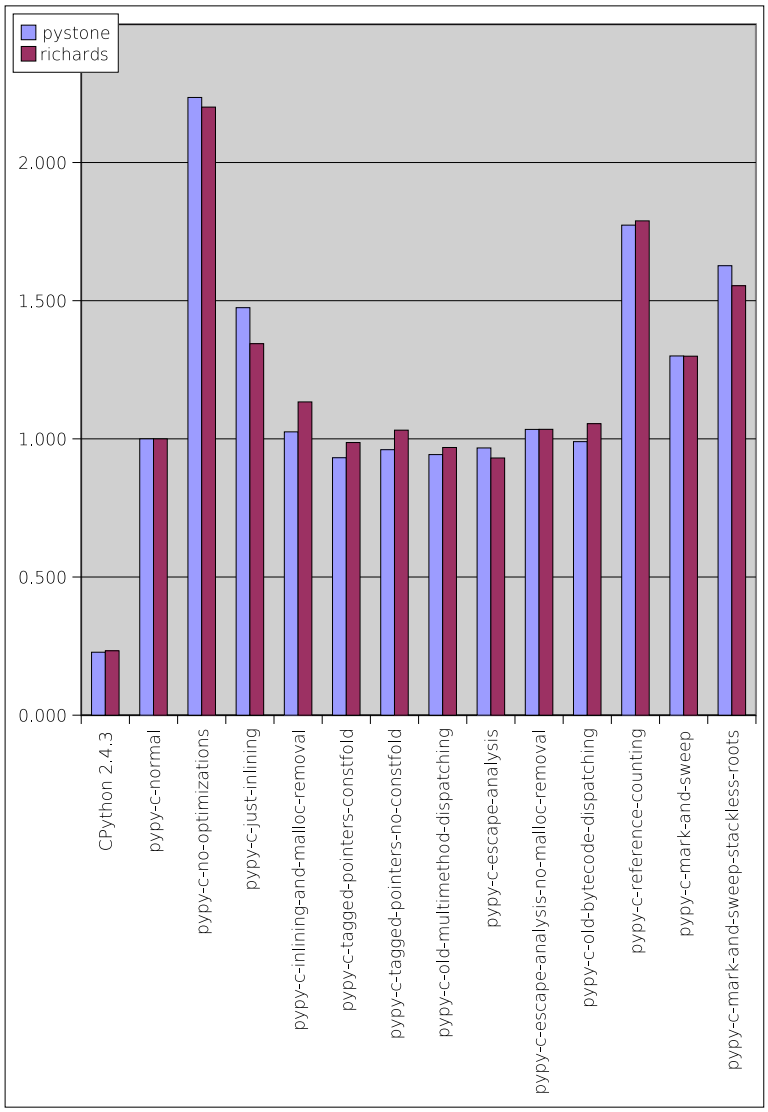

Figure 1: Benchmark results of the various PyPy versions compared to CPython. The results are normalized to pypy-c-normal. The pypy-0.7 data is left out.

Blackburn et al (MMTK) was a source of inspiration for the design of our GC construction framework. PyPy's GC implementations themselves are completely classical. Unlike the MMTK project, advanced tuning of high-performance GCs is not our primary focus; we are investigating future solutions along the lines of automatically porting some of MMTK's GCs into our GC framework for reuse in PyPy.

As mention in the Stackless Features section, the approach of using the GC to allow a form of coroutine cloning is, to the best of our knowledge, novel.

5 Conclusion

Both sections of this report showcase the flexibility of PyPy's architecture.

In the first section, the basic result that we described is the transformation of the flow graphs of RPython programs, extending their semantics in ways that give the RPython program complete control over its frame stack – both for inspection and for reconstruction. This was done mostly without sacrificing the overall performance.

We then used this result as a building block for various concurrency primitives in PyPy, allowing them to be implemented in ways that minimize the impact on the rest of the PyPy code base. These primitives are made available to the application-level user, following pre-existing interfaces when applicable. Building on the flexibility of this approach, we also experimented with new interesting operations like coroutine cloning, and we designed ways to make different concurrency models co-operate in the same application.

In the second section of the present report, we described how 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.

To a greater or lesser extent each of the features described above has been implemented in modules at various abstraction levels and supporting each other. The extreme example is probably coroutine cloning, which uses GC support to duplicate objects within a region ("pool") of memory; it relies on the Stackless Transformation to ensure that the relevant stack

frames are first moved to the heap, where they can be cloned by the GC, and it builds on top of the coroutine abstraction to switch the pools in which the GC should allocate new objects. This example showcases the interest of the PyPy approach, i.e. an open translation toolchain in which we can express code and transformations at the level at which they are the most natural (VMC) and in ways that are open to further transformations (no C code was involved in any of this).

6 Glossary of Abbreviations

The following abbreviations may be used within this document:

6.1 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.
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 Direct-media Library - 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

(AB-OC) Ongoing Continuations in "HREF Considered Harmful", the weblog of Avi Bryant

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

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

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

(CGSI) Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S., and Felleisen, M. 2005. Continuations from generalized stack inspection. In Proceedings of the Tenth ACM SIGPLAN international Conference on Functional Programming (Tallinn, Estonia, September 26 - 28, 2005). ICFP '05. ACM Press, New York, NY, 216-227. DOI= http://doi.acm.org/10.1145/1086365.1086393

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

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

(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

(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

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

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

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

(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

(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

(LIMBO) The Limbo Programming Language, Dennis M. Ritchie, http://www.vitanuova.com/inferno/papers/limbo.html

(MAS) Clements, J., Flatt, M., and Felleisen, M. 2001. Modeling an Algebraic Stepper. In Proceedings of the 10th European Symposium on Programming Languages and Systems (April 02 - 06, 2001). D. Sands, Ed. Lecture Notes In Computer Science, vol. 2028. Springer-Verlag, London, 320-334.

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

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

(PICOIL) Sekiguchi, T., Sakamoto, T., and Yonezawa, A. 2001. Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling. In Advances in Exception Handling Techniques (the Book Grow Out of A ECOOP 2000 Workshop) A. B. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, Eds. Lecture Notes In Computer Science, vol. 2022. Springer-Verlag, London, 217-233.

(PLTWebserver) Krishnamurthi, Hopkins, McCarthy, Graunke, Pettyjohn , Felleisen (Journal of Higher-Order and Symbolic Computing) Implementation and Use of the PLT Scheme Web Server

(PMTPM) Tao, W. 2001 A Portable Mechanism for Thread Persistence and Migration (Mobile Agent). Doctoral Thesis. UMI Order Number: AAI3005121.

(PYLIB) *The Py lib*, Holger Krekel et al, http://codespeak.net/py/

(R5RS) Adams, N. I., Bartley, D. H., Brooks, G., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rozas, G. J., Steele, G. L., Sussman, G. J., Wand, M., and Abelson, H. 1998. Revised5 report on the algorithmic language scheme. SIGPLAN Not. 33, 9 (Sep. 1998), 26-76. DOI= http://doi.acm.org/10.1145/290229.290234

(RCPFCC) Hieb, R., Dybvig, R. K., and Bruggeman, C. 1990. Representing control in the presence of first-class continuations. In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (White Plains, New York, United States). PLDI '90. ACM Press, New York, NY, 66-77. DOI= http://doi.acm.org/10.1145/93542.93554

(SEDA) SEDA: An Architecture for Well-Conditioned, Scalable Internet Services, Matt Welsh, David Culler, and Eric Brewer. In Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18), Banff, Canada, October, 2001

(SLP) *Stackless Python*, Christian Tismer, http://www.stackless.com/

(SLP0) http://www.stackless.com/index_old.htm

(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

(Seaside) Stéphane Ducasse, Adrian Lienhard and Lukas Renggli, "Seaside – a Multiple Control Flow Web Application Framework," Proceedings of ESUG International Smalltalk Conference 2004, September 2004, pp. 231-257.

(Twisted) "Twisted is an event-driven networking framework written in Python and licensed under the MIT license.", http://twistedmatrix.com/trac

(VMC) *PyPy's approach to virtual machine construction*, Armin Rigo, Samuele Pedroni, in OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications, pp. 944-953, ACM Press, 2006

(von-Behren-et-al) Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea for high-concurrency servers. In 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003. http://usenix.org/events/hotos03/tech/full_papers/vonbehren/vonbehren_html/