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

D11.1 A Case Study On Using PyPy For Embedded Devices

Due date of deliverable: March 2007

Actual Submission date: March 30th, 2007

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

Lead Contractor of this WP: Logilab

Authors: Ludovic Aubry, David Douard, Alexandre Fayolle

Revision: Final

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
2007-02-19	David Douard	Interim version
2007-03-15	Ludovic Aubry	Updated the rpyhttp part
2007-03-22	Alexandre Fayolle	Updated porting python to rpython section
2007-03-23	Carl Friedrich Bolz	Small reformulation in the translation overview
2007-03-26	Ludovic Aubry	Final version
2007-03-26	Carl Friedrich Bolz	Publish Final version on the Web Page

Abstract

In this document we explore the numerous possibilities PyPy can offer for embedded application developers. We show that the existing implementation of PyPy can already be helpful to some kind of application development process. We also show that PyPy has great potential for future methods of development.

Purpose, Scope and Related Documents

The purpose of this document is to provide the reader with an understanding of the pros and cons of using PyPy as a full or partial toolchain for embedded systems development.

This document describes:

- The type of embedded applications that PyPy can hope to target
- Different ways of using PyPy for embedded application development
- The advantages and challenges of using the RPython language to develop embedded applications
- The process of developing a small embedded web server using the standard Python library and PyPy

Related Documents:

- D05.1 Compiling Dynamic Language Implementations, PyPy EU-Report, 2005 [D05.1]
- D05.2 A Compiled Version of PyPy, PyPy EU-Report, 2005 [D05.2]
- D05.3 Implementation with Translation Aspects, PyPy EU-Report, 2005 [D05.3]
- D06.1 Core Object Optimization Results, PyPy EU-Report, 2007 [D06.1]
- D07.1 Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects [D07.1]
- D10.1 Aspect-Oriented, Design-by-Contract Programming and RPython static checking [D10.1]

Contents

1 Executive Summary

Embedded systems are a wide and increasing domain. Developing software for such systems involves dealing with a number of specific constraints, ranging from real time requirements to computing resource limitations (CPU and memory). The traditional programming languages used to develop such systems are C and Assembly.

The PyPy project can be used to develop applications for some classes of embedded systems, typically those with soft real time constraints and not too little physical memory, using the RPython subset of the Python language to write code that is then translated to C. We demonstrate this use of PyPy by developing some code targeted at Axis hardware.

It should be possible with future versions of PyPy to use the framework in other ways in the embedded context, for instance by shipping a restricted version of the interpreter, dedicated to the interpretation of a given set of programs.

2 Embedded System Challenges

Embedded systems are a wide and increasing domain. Wide because very different hardware and software systems can be called embedded. Actually, they are dominating the CPU market in quantity. But the very wide diversity of embedded applications and devices makes generalizations difficult.

Embedded systems are generally special-purpose systems in which the CPU and all the required secondary resources are bundled on the same chip (which are then named micro-controllers for smaller devices, or System-on-Chip for bigger ones) or on a very small factor printed circuit board (PCB). The embedded device often consists of (most of the devices enumerated here may or may not be present):

- CPU
- Memory (RAM, Flash, Hard drive, etc.)
- Power management system
- Human Interface devices (LEDs, keys, keyboard, LCD, screen, etc.)
- Specific bus systems (CAN, I2C, etc.)
- Specific diagnostic/programming ports (JTAG)
- Communication ports (network, UARTs, etc.)
- A/D or D/A signal converters
- Specific devices (timers, PWM controllers, etc.)
- Sensor and actuators (generally, through A/D and D/A interfaces)
- Specific chips (FPGA, DSP, etc.)

They often aim at reacting on external stimuli in a precise and fully predictable way. The origin of these stimuli may be external (user interaction, signals from detectors or switches, etc.), but can also come from the device itself (timers, interrupts, etc.).

Embedded devices mainly target mass markets (but can also target very small production volume markets), and thus must be as small and cost effective as possible for the task they are designed for. When the production volume is as important as millions of devices (in cars, industrial control systems, etc.), every cent count.

Another very specific constraint of embedded devices is the fact that, being embedded in devices that may run non-stop for years, they should not have software failures, nor require any maintenance operations. Their software can generally not be upgraded either (for technical or economic reasons).

2.1 Hardware Constraints

Embedded devices can be:

- Very small systems:
  - 4 or 8-bits micro-controllers, in which there is no OS like environment, no MMU or so, no protection.
  - Can be found in almost every day to day devices (from coffee machine to cars, as device controller in computers, battery controllers, sensors, etc.). Most used (in quantity) hardware.
  - Main design constraints are cost, then reliability.
  - Developments are done mainly in C and assembly.
- Micro-controllers:
  - 8, 16 or 32 bits micro-controllers, in which very small OS can be used, but still have very limited RAM, ROM and CPU power, and have no MMU.
  - Can be found in items like phones, cars (injection pump calculator, ABS control systems, etc.), airplanes...
  - Main design constraints are cost, CPU power, then memory.
  - Development can be done with various tools and languages (C, C++ (embedded), Java, Basic, assembly). There are also some custom languages.
- Small systems with quite standard architectures:
  - System built around ARM, Freescale, Geode, etc. CPU acts more/less like a small computer. Can run a complete OS (Linux, VxWorks, eCos, QNX, etc.).
  - The limitations are the small amount of RAM (compared to desktop computers), limited CPU, sometimes power consuption, etc.
  - Very common in printers, in network devices (routers), in PDAs, in GPS devices, in 3G or more mobile phones, in cars.
  - Depending on the application, the main design constraint is either cost, CPU power or power consuption.
  - No main programming language, developers generally use standard Unix tools. When the target platform is too slow, cross compilers to generate binaries for the target platform can be used.
- DSP based systems.
  - These are generally coupled with a CPU. Can be has different as a MP3 player, a VOIP echo removal system...
  - Main development tools are C, assembly and/or DSP specific C or Java compilers.
- FPGA based systems.
  - Originally designed for very specific tasks, FPGA are becoming more and more versatile.
  - No OS.
  - Developments are mainly done in VHDL and Verilog.

Examples of embedded applications and their typical constraints (from [KOOP96]):

	Signal processing	Mission critical	Distributed sensor network	Small
CPU power	1GFlops	10-100 MIPS	1-10 MIPS	0.01-1 MIPS
Memory size	32-128 MB	16-32 MB	1-16 MB	1 kB
Lifetime	15-30 years	20-30 years	25-50 years	10-15 years
Maintenance	Frequent repair	Aggressive fault detection/maintenance	Scheduled maintenance	"Never" breaks
Product variants	1-5	5-20	10-10000	3-10

2.2 Software Constraints

On the other side, software for embedded devices potentially must match constraints according the environment in which the device is deployed. The most frequent constraint is timing limits; it is often desired or vital that the device achieve a task -- as a response to a stimulus -- in a limited and predictable amount of time. Common examples include control systems for nuclear power plants or ABS (Anti-lock Brake System).

2.2.1 Real-Time Systems

A real-time system can be defined as "a system in which the information is still meaningful after it has been acquired and processed".

There are commonly 2 kinds of real-time systems:

- Soft real-time systems: the time constraints are quite fuzzy; a delay of half a second is fast enough; the fact the system loose a piece of information is not critical.
- Hard real-time systems: the time constraints are much narrower, and often must not be exceeded at any price; time scales are around 10 ms; each piece of information may be critical, and must then be properly managed by the system.

2.2.2 Software Safety And Validation

The fact that typical embedded applications must not fail, and cannot easily be upgraded, makes it critical to be able to verify behavioural code correctness *before* deploying the device, as much as possible. For very small and small devices, where software is often written in assembly or C language, this is achieved mainly by testing on emulation systems, and by enforcing good coding practices.

There are also attempts to address this problem providing specific languages. The proposed solutions generally reside on static checking performed at compile time. For example, the Control-C language [KDA02] is a subset of C with key restrictions, designed to ensure that memory safety of code can be completely verified at compilation time by static checking, .

This area has been recently investigated in order to provide high level and object oriented languages on those very limited platforms (micro-controllers). Virgil [TIT06] is a lightweight object-oriented language designed with careful consideration for resource-limited domains. However, as for today, very few solutions have been explored to bring "formal" solutions to software development in very constrained systems such as small micro-controllers.

For systems a bit less constrained, many commercial and open source solutions exists. However, the problems encountered in very small devices remain present, together with more potential software safety problems related to the functionalities brought by dynamic memory allocation and OS-like functionalities. On the one hand, there

is a specific version of the Java platform for embedded devices [JEMB]. The language itself is a limited subset of the Java language, the Java virtual machine specifications have been adapted for constrained embedded devices. Many commercial solutions propose Java environment for embedded devices, asserting reduced hardware resources and real-time constraints. On the other hand, there are many solutions specifically designed for embedded development. These are often small footprint OS with real-time capabilities, with a development tool suit, allowing to develop using C or C++.

As long as the system allows dynamic memory allocation, some issues arise with respect to safety and fault tolerance. There are papers addressing these issues, e.g.:

- Address segment protection for embedded devices with no virtual memory protection [SMB05].
- Memory safety without garbage collection [DKAL05].
- Code safety without run time checks [KDA02].

3 PyPy For Embedded Systems

PyPy, being a very flexible and versatile environment around the Python language can be considered as a tool suit for embedded devices.

There are several ways one can think of using PyPy in this domain:

1. Create a Python interpreter using the PyPy tool suit for the specified platform. One can think of a framework built on PyPy to design a Python interpreter for any embedded platform easily by implementing a specific backend for the device family.
2. Create a lightweight Python interpreter using the PyPy tool suit for the specified platform, which is able to interpret a reduced version of the Python language, the way Java for embedded platforms is a reduced subset of the Java SE platform.
3. Create a RPython interpreter. This is a stripped down objective of the previous one (the way Java has specific JVM and language specifications for embedded devices, RPython could be seen as the subset of the Python language for embedded devices, with a specific VM that can be built using the PyPy tool suit).
4. Use PyPy extension abilities to provide specific language aspects (especially constraint programming, see WP09) to improve software safety.
5. Use the translation mechanism of PyPy to convert RPython code into C or LLVM (or any other backend supported by PyPy).

3.1 Using PyPy Instead Of C-Python For Embedded Systems

If the first option seems attractive, it is for now not feasible, due to the CPU power and memory requirements to have a working Python interpreter generated with the PyPy tool suit.

In fact at the time of this writing a generated PyPy interpreter is still bigger and slower than it's handwritten (CPython) counterpart. On the other hand, over the past three years, size and speed have improved considerably, and some builds today are less than 2x slower. However, reducing the generated code size has never been a goal per se.

3.1.1 Status

Example of generated code size and approximate speed for a specific build (Feb, 23rd 2007):

Size	Code Size	Options	Richards	Pystone
1015360	879453	CPython 2.4.4	1.0	1.0
946284	823915	CPython 2.3.5	1.2	0.8
7186692	3944448	llvm-39324-faassen-x86	2.3	3.1
7122732	3399680	llvm-39324-faassen-c-prof	2.0	2.5
6991972	3649536	llvm-39324-faassen-c	2.2	2.8
8695808	N/A	cli-39324	57.4	70.3
7940976	4673536	c-39324-stackless--objspace-std-withmultidict--prof	3.8	4.7
5301380	2174976	c-39324-prof--objspace-opcodes-CALL_LIKELY_BUILTIN	3.4	4.2
5301296	2174976	c-39324-objspace-std-withmultidict--prof	3.7	4.5
5393632	2211840	c-39324-objspace-std-withfastslice	5.8	6.7
6855068	2842624	c-39324-gc=framework--objspace-std-withmultidict--prof	3.8	4.2
6124960	2940928	c-39324-faassen	2.6	3.4
5349176	2174976	c-39324	6.0	6.4

These statistics are collected daily and are available from the daily PyPy benchmarks

As we can see in the above table, the size of the executable is an order of magnitude larger.

A lot of work done as part of WP6 and described in [D06.1] has improved this status.

3.1.2 Areas Of Improvement

By examining the generated code, one can pinpoint some areas where code size could be reduced:

3.1.3 Exception Handling

In the C backend, using the lltypesystem, exceptions are handled by returning a `NULL` (like in C Python) which makes the backend generate testing code after each function call.

By using a C++ compiler, we could take advantage of the native C++ exception handling mechanism to reduce code size and overhead in this area.

Other options could use the C functions `setjmp` and `longjmp` to set and restore exception handling points. This technique will be difficult to implement with the reference counting garbage collector, but other collectors could probably handle this cleanly.

3.1.4 Garbage Collection

PyPy can be compiled with one of several memory management implementations. They are described in detail in [D07.1]. In terms of code size, the *Boehm* garbage collector is probably the lightest available. The *framework* garbage collector provided with PyPy is more efficient during collection but, in its current implementation, adds a significant overhead to each function call. The reason is that it needs to maintain the list of reachable live objects which Boehm just guesses. The current implementation saves the live objects on a stack before each function call. The stack is used as part of the root objects. This seems to induce a significant overhead both in terms of code size and performance.

3.2 Providing Embedded Specific Extensions To PyPy

There are two aspects in providing specific extensions to PyPy for embedded development.

First, provide run time extensions useful for embedded systems. Such extensions already exists, since PyPy provides a pluggable garbage collection framework [D07.1] which allows selecting the memory collection strategies of the interpreter.

However this cannot really be considered as long as there is no real solution to have a Python interpreter compiled using the PyPy tool suit that fits the embedded requirements (in terms of CPU power requirements and memory usage).

the second aspect is providing extensions to the compiler that will allow it to do more checking and validation on the source code. For example, computing the maximum stack depth of an application, and detect recursion. The stackless mode of PyPy can even remove such recursions at the cost of a larger usage of heap allocated memory.

The Stanford checker is an example of what kind of checking support PyPy could provide to embedded application developers (and other developers as well). Such applications include but are not limited to:

- Checking locking rules in multi-threaded applications
- Checking error reporting, detecting unhandled exceptions
- Checking various program-specific constraints

RPylint also provides a step in this direction but the infrastructure and design goals of Pylint won't allow all the checks (like stack depth computation) that PyPy itself can potentially provide.

3.3 Implementing An RPython Interpreter

According the way the PyPy tool suit actually works, implementing an RPython interpreter for embedded devices is not an option. The RPython language is a subset of the Python language. The PyPy tool suit takes RPython programs and transforms them into other languages (C, LLVM, etc.). The transformation is a complex process and RPython was not designed with an interpreted goal in mind: It is the (largest possible) static subset of the Python language (+ restrictions) that can be transformed by the translation process. This process consist in mainly 3 steps (see [D05.1], [D05.2] and [D05.3] for more details on the process):

- Annotation: a recursive process that generates the flow graphs and infers the types of the code execution. One cannot know beforehand how many iterations this process will require to reach a fixed point.
- RTyping: using the previously generated flow-graph and type information, the RTyper will transform the flow-graph to generate a low-level flow-graph. Note that it may be required to go back to the annotation step in order to complete RTyping.
- Code generation: from the transformed low-level flow-graph, target code can be generated (as long as a backend exists).

So this complex procedure is far away from being potentially considered for use in embedded devices.

3.4 Implementing A Lightweight Python Interpreter With PyPy

This option might be an interesting way to investigate. But nothing in the current code of PyPy allows doing this out-of-the-box.

The goal would be to remove the parser and only keep the bytecode interpreter, thus excluding `eval` and `exec` builtins. Pushing things further, one could remove support for less useful objects like complex, sets, unicode strings, long integers and/or floats. This would reduce code significantly.

There is no obvious support for doing that right now, but PyPy's modular architecture should allow that to be done more easily than with CPython. As it is easy to derive the standard object space to provide a new one with more types, builtin functions or different semantics (see for example Object Space and the Taint Object Space), it's not much more difficult to provide a reduced object space with less functionnalities.

3.5 Using RPython As A Development Language For Embedded Applications

The fifth option can be view as using the PyPy translation tool suit to generate code that will be executed by the lightweight device. The current state of the PyPy tool suit makes this option pretty much possible. The idea is to propose a tool suit for the programmer that lets him write code in RPython, and produce C (or Java) code that will be executed by his target device. This option can be investigated and used with very minimal effort with the current version of PyPy.

4 Coding In RPython

Mastered and efficient development cycle being one of the major issue of embedded market software development, using a high level language like Python can be really valuable. However, for now, Python as is can not be used in most of the embedded systems.

RPython can be a good candidate. Being a subset of the Python language with restrictions, it is not as versatile as pure Python, but many of the nice aspects of the Python language remain available. Today, the PyPy toolchain (PyPy version 1.0 is on the way) can be used to generate C code from RPython snippets. Some Python standard libraries are not yet available in RPython, some others are not complete, but as this work progresses, the idea and the feasibility of using the PyPy translation mechanism to produce C code for the target embedded platform becomes more and more realistic.

4.1 Advantages

The biggest advantage of using RPython and the PyPy toolchain is that RPython is much easier to program with, and the resulting program will be translated to C and compiled and executed on the target:

- there is no need to declare variables and the PyPy translation process will detect the correct typing and also any type checking errors that may be introduced.
- RPython provides a lot of high-level data structures that are not available in plain C, and, although those structures exist in C++, are not as flexible. These are lists, dictionaries, tuples. Furthermore, their implementation can be tuned at compile time to provide the best suited implementation for the application.

A second key point is portability. The RPython code is completely hardware and OS agnostic. It is the responsibility of the backend, and then of the compiler to make the link with the hardware and OS layers.

This approach is not much different than programming in Java for embedded applications. The only real advantage is that RPython is probably easier to program with, although the current implementation lacks a lot of standard library functions.

4.2 Drawbacks

Some key points prevent to use the actual PyPy translation framework for generating code for embedded devices.

First of all, the lack of a modular enough compilation tool suit. The PyPy translation tool suit is for now very parameterizable, but some key points cannot be chosen, like to be able to link a very minimalistic executable, with no `libm`, `libpthread`, and so on, using a minimalistic version of the `libc` (eg. `uclibc`).

The memory management, using a garbage collector, can be chosen among a set of implementations (Boehm, reference counting, etc.), but none of them has been designed specifically for the embedded devices, using for example a GC that behaves nicely with regards to real-time constraints or heavily constrained memory. On the other hand, PyPy provides a garbage collector framework allowing development of custom collectors in RPython, which means that some important infrastructure is already in place to allow this.

A very practical problem of the current implementation of the PyPy tool suit is the fact that the error messages reported during the translation process are sometimes very cryptic. Another potential limitation is the fact that the translation stops at first error, which can make the code development difficult, loosing the interest of using a high level language like RPython. The Python language itself has the same kind of limitations (syntax errors are reported one at a time), but this is mitigated by the absence of a compile step in the toolchain.

The documentation, which is today quite important, is not really easy to use with the goal of writing RPython code.

Last, if one wants to use the PyPy toolchain for creating software for embedded devices, it is very probable that modules will be required to access the specific hardware devices on the board, like the several bus systems (CAN, I2C, etc.).

However, none of these points are impossible to address, and most of them are merely a matter of the PyPy project maturing beyond the state of a research project. One of the hardest points are modular compilation and the real-time garbage collector.

4.3 Ideas To Overcome The Shortcomings

The first thing to do, and probably the easiest one, is to provide a very complete documentation on what can be done with RPython, which modules from the standard python library can be used, and which modules are available in the RPython library (rlib). This will become easier once PyPy 1.0 is released, when the RPython specification has become more stable.

One important aspect is to provide helper tools for the developer. RPython being a subset of the Python language -- with some limitations -- it is not always easy to find out what goes wrong in ones code snippets. Task 3 of WP10 [D10.1] has provided a specific checker for the Pylint static code checker: this checker tries to assess the correctness of a piece of RPython code. It is not perfect by far, and still has a number of false negatives and a few false positive warnings, but it provides a tremendous help when writing RPython code with a Python background, or when porting Python code to RPython. As such, part of the expected target audience for this tool is software developers writing RPython code to translate it to C for embedded devices.

5 Work Case: Using PyPy To Build Embedded System UI For Axis Communication devices

5.1 Axis Communication Presentation

Axis Communication is a swedish company that develop solutions for network video and print servers. It sells embedded devices all over the world. Axis' products and solutions are focused on applications such as security, surveillance, remote monitoring and document management. They are based on in-house developed chip technology, which is also sold to third parties.

Most of the embedded devices developed by Axis Communications are built around a specific CPU architecture, the ETRAX chips, with the system on chip version (ETRAX 100LX multi chip), being used at the heart of almost all devices designed and sold by Axis Communications.

All their devices are running a specific version of the Linux kernel (2.4 and 2.6 are available), with support for full MMU, standard `glibc` or `uclibc`, shared libraries and Linux and POSIX threads. Device drivers for Ethernet, serial and parallel ports, GPIO, USB and IDE buses are also present in the freely downloadable SDK for their ETRAX based platforms.

The ETRAX board is built around the ETRAX 100LX SiC. The typical specifications of the board are:

- a 100 MIPS CPU with MMU
- 2 to 8 Mb of Flash memory
- 8 to 32 Mb of RAM
- 1 or 2 Ethernet transceivers
- RS232 serial ports
- RS422 ports
- a GPIO bus
- several switches and LEDs

Axis Communication has shown interest in the PyPy project, mainly on how to get tools to facilitate software development and simplify the development cycle for the kind of application they design. These are not critical applications for quite big embedded devices, as described above.

To follow are a number of application test cases that are typical applications that run on Axis Communications devices. These have been written in RPython, to demonstrate the feasibility of using the PyPy translation toolchain to write code that can be run on these embedded devices.

5.2 Simple Web Server In RPython

The choice of this application was guided by the fact that most SOHO (small office, home office) embedded appliances are manageable through a web interface.

This is the case of Axis products and this is also true of almost every firewall, and network attached storage on this market. These kind of products are similar in terms of power requirements, memory and solid state memory. Some NAS appliances, however, use part of the hard drive they serve to store their operating system.

In the following section we will describe the process we followed to turn a simple HTTP server example written in Python into an executable compiled with PyPy, which will be refered to as rpyhttp in the rest of this document.

As we will see the process is still a bit rough and the following section describe what would be the expected process for using PyPy as an embedded software development SDK.

5.2.1 Compiling `SimpleHttpServer`

The Python standard library contains a lot of useful modules for developing web applications. One of the module defines a class `BaseHTTPServer` which can be used as a basis for building a fully scriptable web server: one can implement a regular file serving HTTP server or intercept the URL and generate pages dynamically.

The module `SimpleHTTPServer` is a demonstration module using `BaseHTTPServer` that provides a simple web server that serves the files in the current working directory

There is very little code required to use this module with CPython. Here is the relevant excerpt from `demo_rhttp.py`:

```
import SimpleHTTPServer
Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

def entry_point(argv):
    port = 8000
    if len(argv)>1:
        port = int(argv[1])
    httpd = SocketServer.TCPServer(socket.INETAddress("", port), Handler)
    os.write( 1, "serving at port %s" % port )
    try:
        httpd.serve_forever()
    except:
        pass
    return 0
```

When providing a custom implementation of a web server, it is only necessary to override methods of `SimpleHTTPRequestHanc`

The next step is to try and compile it with PyPy. In order to do that we need to provide a `target` for the PyPy translator.

In our case, it is also simple to do; the `targethttp.py` file is listed here:

```
import os, sys, stat
import pypy.interpreter.gateway # needed before sys, order of imports !!!

# ___________  Entry point  ___________
from demo_rhttp import entry_point

# _____ Define and setup target ___

def target(*args):
    return entry_point, None
```

Then, compiling the application is a simple matter of running:

```
cd pypy/translator/goal
./translate.py targethttp.py
```

This alone will not work, since large large parts of the CPython library are not written using the restricted rules of RPython. However, with RPython and PyPy improving, it has become significantly easier to turn existing code into RPython compliant code.

In the following paragraphs we will describe some of the steps we went through in order to be able to turn the SimpleHTTPServer example into an RPython application.

5.2.2 Python To RPython Adaptations

The `socket` Module

> The first obvious change was to port the code in the `SocketServer` module to use the `pypy.rsocket` module instead of the standard `socket` module. The port was very straightforward, as the implementation of the `rsocket` module exposes the same symbols as the standard library module except for the `socket` class which is renamed `RSocket` and the `error` exception which is renamed `SocketError`.

String Manipulations

Most programs make heavy use of character strings, and Python's excellent support for this data type, through the various methods available on string objects and the powerful indexation and slicing syntax, is one on the most obvious benefits of the language over its competitors. RPython's string support is not as good, and a lot of high level operations have to be rewritten using lower level primitives, or adapted to a weaker form.

One key restriction has to do with string slicing and indexing. RPython does not allow negative indices other than `-1`, for instance, so code such as this one, found in `BaseHTTPServer`:

```
def parse_request(self):
    ...
    if requestline[-2:] == '\r\n':
        requestline = requestline[:-2]
    elif requestline[-1:] == '\n':
        requestline = requestline[:-1]
    ...
```

has to be rewritten as:

```
def parse_request(self):
    ...
    L = len(requestline)
    L2 = L-2
    L1 = L-1
    if L2>=0 and requestline[L2:] == '\r\n':
        requestline = requestline[:L2]
    elif L1>=0 and requestline[L1:] == '\n':
        requestline = requestline[:L1]
    ...
```

Another significant restriction is on the `str.split(substring, maxcuts)` method: the Python version accepts an arbitrary string as first argument, and an optional integer specifying the maximum number of splits to perform. The RPython version accepts at most one character to split on, and it is not possible to specify a maximum number of splits. Therefore, some Python one liners must be rewritten, and:

```
def parse_request(self):
    ...
    base_version_number = version.split('/', 1)[1]
    ...
```

becomes:

```
def parse_request(self):
     ...
     ver_split = version.split('/')
     base_version_number = "/".join( ver_split[1:] )
     ...
```

String formatting in RPython also has some restriction over the CPython version. Some formats are not supported (`%r`, `%x`...), and the named string substitution, where the values to insert in a string are read from a dictionary is not available: code using this feature must be adapted to use the standard positional substitution method.

String Substitution

RPython support for operator `%` on string is limited to positional arguments only. So substitutions like:

```
d = { 'arg1': 1, 'arg2': 2 }
s = "%(arg2)s some text %(arg1)s" % d
```

had to be replaced by it's positional counterpart (ie removing usage of the dictionary):

```
s = "%s some text %s" % (2, 1)
```

Variable Arguments Functions

The `*args` and `**kwargs` facility is not available in RPython. This is used in `BaseHTTPServer` to pass a string format and values to be substituted to the `log_message()` method. This had to be adapted to call the `%` operator to perform the substitution before calling that method throughout the code.

Dynamic Method Invocation

A very common CPython idiom consists in dispatching a call to a method with a name built at run time. The `SimpleHTTPServer` module uses this to dispatch the processing of an HTTP request depending on the value of the command in the request: a `GET` request gets processed by the `do_GET` method of the server, etc. The code is:

```
def handle_one_request(self):
        ...
        mname = 'do_' + self.command
        if not hasattr(self, mname):
            self.send_error(501, "Unsupported method (%r)" % self.command)
            return
        method = getattr(self, mname)
        method()
        ...
```

This is not RPython, and needs to be rewritten as:

```
def handle_one_request(self):
        ...
        if self.command == 'GET':
            self.do_GET()
        elif self.command == 'POST':
            self.do_POST()
        elif self.command == 'HEAD':
            self.do_HEAD()
        else:
           self.send_error(501, "Unsupported method (%s)" % self.command)
        ...
```

Inputs And Outputs

Only low level file operation are supported in RPython at the moment. That means one has to cope with calling `os.write` and `os.read` on file descriptors instead of manipulating file objects like regular Python files.

Usage of `print` is also unsupported and has been replaced by calls to `os.write`. For the same reasons, writing to `sys.stdout` and `sys.stderr` was replaced by calls to `os.write(1, ...)` and `os.write(2, ...)` respectively.

We chose to simply replace file objects by file descriptors in that case because the file operations were simple enough and the modifications were straightforward.

On the other hand, `SocketServer.py` uses file-like socket objects as `wfile` and `rfile` attributes of the handler. These objects are used in a lot of places and it would have been too tedious to change the code everywere. So the module implements a file-like object that is used in place of the original socket file. The only trouble with this object was that it doesn't provide full support for `seek`.

Fortunately, the code in `rfc822.py` only uses `seek` in a very limited way. A tricky part of the port was deeling with the fallback strategy of the module which checks for the presence of a method called `unread` as a `seek` replacement. This check had to be removed since RPython doesn't support dynamic method lookup.

Unsupported And Partially Supported Modules

Some functions had to be reimplemented using low level primitives available in RPython. These include:

- `os.listdir()`
- `os.path.islink()`
- `os.path.isdir()`
- `os.path.splitext()`
- `os.path.split()`
- `urlparse.urlparse()`
- `urllib.quote()`
- `cgi.escape()`

Other unavailable functions were dropped and functionality has been reduced:

- `time.gmtime`: was used to format the timestamp on generated pages

5.3 Managing `/bin/init` In RPython

CPython is already widely used in applications that were once built as a set of shell scripts, since it provides enough functionalities to be able to replace any kind of shell environment.

With this in mind, the idea of replacing `/bin/init` for an embedded application becomes quite natural since basic system management, configuration and logging can be done entirely in Python. Since the `fork` system call is available, it also makes sense to develop the application entirely as one Python application. It can be tested and developed on any Unix system, and then compiled and installed as a single binary on the target system.

This approach offers several advantages over standard development methodologies:

- First of all, this can be a simplification over a standard SDK: most development kits have to provide at least partial POSIX support (generally using busybox), and a method to build a minimal Unix system. With Python and its (restricted) library as a development language, most of the standard Unix command line tools become redundant. For example:

  - the `sed` and `grep` commands are replaced by the `re` module which is already available in RPython
  - most system calls are available though the `os` and `ros` modules, which allows for file system handling and socket programming

- This is an approach similar to that of busybox, which provides a lot of command line tools built as a single static executable, so that every single line of code that can be shared among several commands is really shared.

  This approach is going one step further by building the whole application this way so that code sharing can be done between operating system tools and the applications.

- Some applications which require fast start up time will benefit from a single application forking instead of boot scripts executing: the Linux `exec` system call usually maps the executable in memory and lets the virtual memory manager swap in the code on demand. Note however that forking on Linux is quite cheap, as code segments are shared, read-only mappings.

- Testing cycles can be reduced, as only one application needs to be rebuilt and reinstalled, and this application can first be tested without compilation by running it on top of CPython.

The main disadvantage of this approach is the need to recompile the entire application each time one needs to install a new binary on the target system, but this is compensated by the fact that testing can be much easier. As a matter of fact, the dynamic nature of Python can be used at its fullest during testing: for instance, if an embedded application needs access to a special device driver that is available only on the development platform, Python makes it really easy to provide a replacement driver library to emulate the hardware.

5.4 Evaluation Of Implemented Solutions

To evaluate the simple HTTP server we produced from the standard python library modules, we will compare it against Boa . Boa is a lightweight and high performance web server often used in embedded systems based using free software.

It's a single-tasking HTTP server. That means that unlike traditional web servers, it does not fork for each incoming connection, nor does it fork many copies of itself to handle multiple connections. It internally multiplexes all of the ongoing HTTP connections, and forks only for CGI programs (which must be separate processes). Preliminary tests show boa is capable of handling several hundred hits per second on a 100 MHz Pentium.

5.4.1 Development Speed

Developing the HTTP server took about 7 days. 60% of that time was spend on transforming standard library modules to allow them to be compliant with RPython, and 35% were devoted to tune/improve RPython to allow it to accept some common patterns found in the library modules, viz. strip with more than one character, tuple comparison, and negative indexes. Writing the HTTP server code itself using standard python modules took five minutes.

Even if, in this case, the compilation time is quite acceptable, testing the application can be done by running the code directly with CPython.

5.4.2 Generated Code Size

Boa's binary executable is significantly smaller than rpyhttp, but the text segment is roughly the same size for both executables.

The following figures are computed on an intel x86 platform:

text	data	bss	dec	hex	filename
87427	79032	484	166943	28c1f	targethttp-c
70754	2172	14720	87646	1565e	/usr/sbin/boa

5.4.3 Efficiency

The application developed is not significant nor complete enough to allow correct benchmarking. So the number given here are just meant to prove that performance are within comparable range when compared to an application written in C.

In the following table we show two simple tests done with boa and our simple server. The first test retrieves the root `/` which involves generating a list of files from the root `www/` directory since there is no `index.html` file.

The second test just retrieves a static file from the disk.

Both tests are done by forking 10000 instances of wget on the client retrieving the same url.

The client script is executed on a 4-way SMP machine connected with 1Gb Ethernet to the server which is a single processor x86 (standard P4).

url	client time wall clock	server user cpu time	server sys cpu time
/ (boa)	62s	3.7s	17.6s
/ (rpyhttp)	25s	1.7s	1.4s
/file.txt (boa)	18s	0.4s	1.0s
/file.txt (rpyhttp)	21s	1.7s	1.2s

As we can see, the user CPU time and System cpu time are comparable. Boa seems slower generating the index page, but is much faster on retrieving static files. That's expected since boa probably does more file system checking than the naive implementation of rpyhttp does (ownership for example).

The application has also been run on a PEPLINK Manga and a Soekris net4801-60. But both of these do not provide enough network bandwidth to really consume cpu time.

The Soekris is a 266Mhz Geode cpu with 256Mb ram. The Manga is based on the ks-8695 (from Micrel) which is a System On Chip (SoC) composed of an ARM922T (~166MHz, without FPU), 32Mb of RAM and two Ethernet controllers with integrated PHY (one of them being a 4 port switch).

The main reason is not the Ethernet link (which is 100Mb/s) but the implementation of the PCI bus. In fact a simple test doing bridged transfer across two network interfaces of the Soekris shows that this machine is limited to about 15Mbps bandwith. It seems that this is a limitation of the PCI bridge and the way interrupts and DMA are managed on the North Bridge.

6 Going Further

We have shown that using the PyPy translation tool suit can be a valuable and viable solution for embedded applications on not too small devices. Of course PyPy is currently not suitable for embedded application development for regular Python programmers. The case study we described earlier clearly shows two main reasons for that:

- RPython is still a bit tedious to develop with,
- there isn't enough (yet) libraries available *as is* from RPython.

However, the power of PyPy and the potential of the PyPy toolchain make it suitable as a base for more advanced applications of the PyPy tool suit for embedded targets.

The following are other leads that could be followed to improve the process of application development for smaller systems.

6.1 microPyPy

The idea is to propose a stripped down version of PyPy. It would be an interpreter that can run Python code, using a very small footprint interpreter. It would not be capable of running any Python, but only a limited subset of the Python language, but less limited than RPython. The idea is actually the same as what has been done in the Java world with the Java for embedded devices platform.

6.2 Using PyPy As A Code Validation Process

It can also be useful to make use of the very advanced features of the RPython process toolchain (Annotator, RTyper, code generator) to not only use the RPython translation process to generate target code, as proposed previously. The toolchain could also be used to ensure code safety.

6.3 Other Ideas

One could also think of a real-time garbage collector to target embedded devices with real-time constraints. It would probably not be possible to assert real-time with very straight time constraints, but some interesting results could be obtained for a wide range of embedded applications.

7 Glossary Of Abbreviations

The following abbreviations may be used within this document:

7.1 Technical Abbreviations:

AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree
CPython	The standard Python interpreter written in C. Generally known as “Python”. Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
CCLP	Concurrent Constraint Logic Programming.
CPS	Continuation-Passing Style.
CSP	Constraint Satisfaction Problem.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime.
docutils	The Python documentation utilities.
F/OSS	Free and Open Source Software
GC	Garbage collector.
GenC backend	The backend for the PyPy translation toolsuite that generates C code.

GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
GenCLI backend	The backend for the PyPy translation toolsuite that generates CLI code.
Graphviz	Graph visualisation software from AT&T.
IL	Intermediate Language: the native assembler-level language of the CLI virtual machine.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.
Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.
Pygame	A Python extension library that wraps the Simple DirectMedia Layer - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
pypy-c	The PyPy Standard Interpreter, translated to C and then compiled to a binary executable program
ReST	reStructuredText, the plaintext markup system used by docutils.
RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.
VM	Virtual Machine.

7.2 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

8 References

References

[TIT06] *Virgil: Objects on the Head of a Pin*, Ben L. Titzer, In Proceedings of the 21st Annual Conference on Object-Oriented Systems Languages, and Applications (OOPSLA '06). October 2006.

[JEMB] http://java.sun.com/javase/embedded/index.jsp

[SMB05] *Segment Protection for Embedded Systems Using Run-time Checks*, Proc. of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'05), San Francisco, CA, September, 2005

[DKAL05] *Memory Safety Without Garbage Collection for Embedded Applications*, Dinakar Dhurjati, Sumant Kowshik, Vikram Adve and Chris Lattner ACM Transactions in Embedded Computing Systems (TECS), February 2005.

[KDA02] *Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems*, Sumant Kowshik, Dinakar Dhurjati & Vikram Adve, CASES 2002, Grenoble, France, Oct 2002.

[KOOP96] "Embedded System Design Issues", Proceedings of the International Conference on Computer Design (ICCD 96)

[D05.1] *Compiling Dynamic Language Implementations*, PyPy EU-Report, 2005

[D05.2] *A Compiled Version of PyPy*, PyPy EU-Report, 2005

[D05.3] *Implementation with Translation Aspects*, PyPy EU-Report, 2005

[D06.1] *Core Object Optimization Results*, PyPy EU-Report, 2007

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

[D10.1] *Aspect-Oriented, Design-by-Contract Programming and RPython static checking*, PyPy EU-Report, 2007