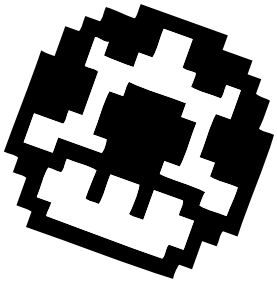


PYGIRL

DEVELOPMENT AND DEBUGGING OF A WHOLE-SYSTEM VM IN RPYTHON

CAMILLO BRUNI

SUPERVISION: TOON VERWAEST

BACHELORS,
UNIVERSITY OF BERN, SWITZERLAND, NOVEMBER 2008

Abstract

Hardware Virtual Machines are generally written in a low-level style with close resemblance to the actual working principle of the original system. We show a different approach by creating a whole-system VM for a hardware gaming device using a high-level language and a high-level representation, which increases readability and maintainability. By creating a fully functional VM model which then can be compiled for different architectures, we postpone low-level optimizations to the compilation step. Our high-level VM model written in Python is translated using the PyPy toolchain, a sophisticated high-level compiler.

Contents

1 Introduction

On one hand there are whole-system VMs, built with close resemblance to the original hardware. This simplifies the initial programming of such VMs, but the resulting code remains unmaintainable, dealing with lots of low-level details. On the other hand there are the high-level language VMs which have no physical counterpart. Those VMs cannot be built according to existing plans, instead new approaches and optimizations are used to create fast programs.

Although both domains use similar techniques, only recently have those two fields converged. As a clear example of this fact we see that modern VM books talk about both fields in one breath [6].

In order to create a portable VM we chose to implement a high-level VM model which is then compiled to different architectures. Instead of creating variants for each backend we let a compiler handle the target specific changes and optimizations. Since our model is written in Python it is fully executable and debuggable during the whole development. We then use the high-level PyPy toolchain to create binaries for different backends of our VM model. PyPy inserts target specific transformation and performs automatically optimizations such as inlining or just-in-time-compilation to create fast binaries.

To show the advantages of this new approach, we port an existing whole-system implementation in Java which is written in a low-level manner. Using high-level optimizations we create a well structured VM model out of the initial procedural-style implementation.

The thesis is structured as follows: In Section 2 on the following page an introduction to the PyPy project is given. Section 3 on page 5 covers the technical details of the emulated gaming device, followed by the actual implementation details of PyGirl in Section 4 on page 6 and debugging details in Section 5 on page 16. In Section 6 on page 18 the performance of the different implementations are compared. An outlook of our future work is provided in Section 7 on page 21. Then finally a brief overview of our achievements is given in Section 8 on page 22.

2 PyPy in a Nutshell

In this section we describe the translation toolchain PyPy. A more detailed project information is available on the PyPy website[1].

The goal of PyPy was to write a full featured, customizable and fast interpreter for Python written in Python, in order to have the language described in itself. Of course, running an interpreter on top of another interpreter results in slow execution, and still requires a first interpreter written in another language below it. For this reason it was decided to build a "domain specific compiler", a toolchain which translates high-level specifications of interpreters (VMs) in Python down to low-level backends, such as C/Posix [4]. Like the interpreter, again, the toolchain itself is written in Python. This effort is similar to other self-sustaining systems like Squeak where the VM is written in Slang [3]. The major difference between Slang and PyPy is that Slang is a thinly veiled Smalltalk-syntax on top of the semantics of C, whereas PyPy focuses on making a more complete subset of the Python language translatable to C. This difference is clearly visible in the level of abstraction used by programs written for the respective platforms.

2.1 The Interpreter

The starting point for PyPy was to create a minimal but full interpreter for Python written in Python itself. With minimal we mean that all interpreter implementation details such as garbage collector and optimizations are not implemented. For these features the PyPy interpreter rather relies on the garbage collector and optimizations available in the environment running it. This results in a very clean and small implementation of the interpreter which models how the language works without obscuring implementation details.

2.2 The Translation Toolchain

Since high-level executable models of VMs will not run fast by themselves, PyPy also implements a "domain specific compiler", which can translate VMs in Python down to low-level code. This translator is designed as a flexible toolchain where front- and backends can be replaced so that it can generate VMs for different languages which will run on different platforms. Not only the front- and backend can be changed, but also different aspects of the translation itself, resulting in fast and portable VMs.

First, the toolchain builds a modifiable and dynamic flow graph from the target program's sources. From there on, this model is transformed in several steps until the final binary results. Using the same intermediate representation for a great part of the translation and applying transformations in small steps allows us to add custom modifications for every translation aspect.

2.2.1 Translation Steps

In the first step the translator loads the source-code it will translate. Unlike standard compilers which would start by parsing the source code, the PyPy toolchain never even sees the Python source code directly. Since the input code for the translator is RPython code (which is a subset of Python code), and the toolchain itself is running on top of a Python interpreter, the toolchain can use its hosting Python interpreter to load and parse the input files. While loading, the Python interpreter will evaluate all top-level statements like decorators and add the loaded method and class definitions to the global Python memory. After loading, the toolchain uses the the globally loaded `main` function as the starting point for its input graph. This setup allows the input model to apply

[1] `http://codespeak.net/pypy/`

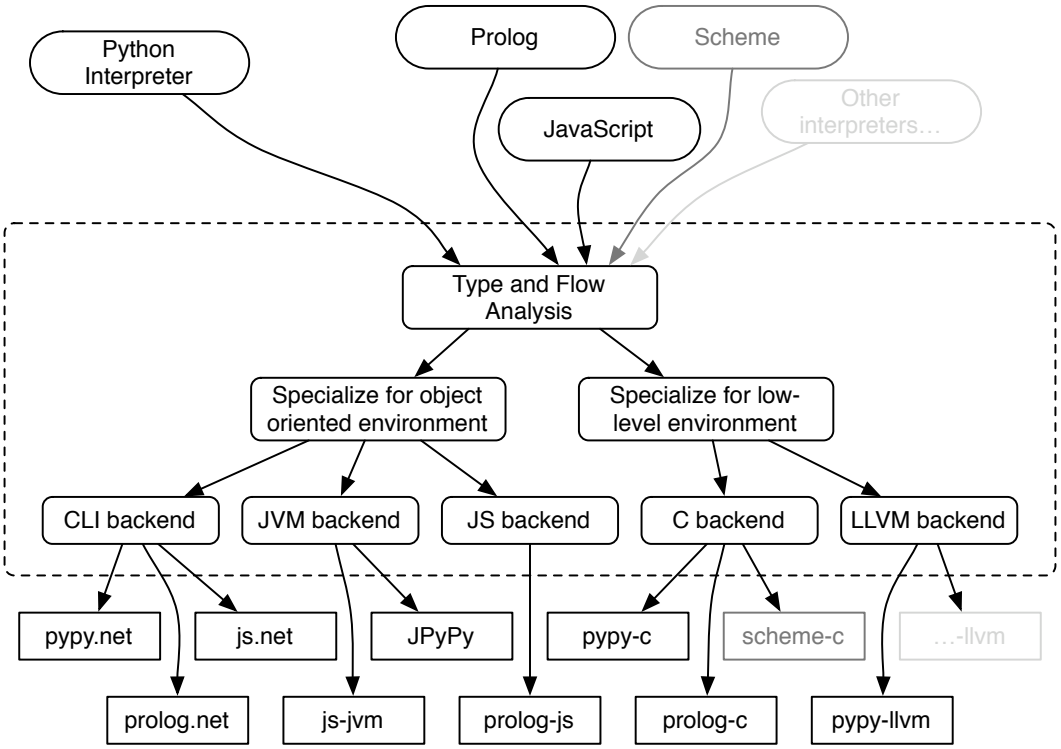


Figure 1: PyPy translation toolchain architecture

metaprogramming in plain Python code at preprocessing time, as long as the resulting graph in memory (generated program) is RPython compatible (Listing 4 on page 9 shows an example of this feature). From the code objects resulting from loading the sources, a control flow graph is generated.

Because PyPy mostly targets statically typed backends, the graph is annotated with inferred types. Starting with a specified entry point, the type inference engine works its way through the object flow graph and tries to infer the most specific types. If multiple types are possible for a certain node, the type inference engine will try to select the common superclass as type. If the most specific type found for a node would be the most common type object type in the system, an error is thrown to show that no specialization was possible. The same happens when two types are incompatible, like booleans and objects. If such type-errors arise, they have to be solved by the programmer of the VM, often by introducing a new common superclass or moving a method higher up the hierarchy. In other cases they really are semantical errors and require restructuring (Section 4.3 on page 11 covers some aspects of resolving errors discovered by the toolchain). Here we see that the compiling process has an impact on the structure of the input source code. It effectively restricts the expressiveness of the input language somewhat. Because of these restrictions we call the input language understood by the PyPy toolchain RPython instead of Python, as we will see in Section 2.3 on the next page.

The annotation step is followed by the conversion from a high-level flow-graph into a low-level one. There is currently a converter which specializes towards low-level backends and one which specializes towards object-oriented backends (Figure 1).

On the low-level flow-graph, optional backend optimizations are performed. These optimizations are rather similar to optimizations found in standard compilers, like function inlining and escape analysis.

After the low-level flow-graph is optimized, it gets specialized for a specific backend. The preparation for code generation covers the following points:

- Insertion of explicit exception handling
- Adding memory management details. Different garbage-collection strategies are available[2]. Note that these garbage collectors themselves are also written in Python code. They also get translated and woven into the VM definition.
- Creation of low-level names for generated function and variables

Finally the language-specific flow-graph is transformed into source files. These source files are then again compiled or assembled to binaries by a language-specific compiler or assembler, which can perform further domain-specific optimizations. For example generated C source files are compiled with GCC with the `-O3` flag.

2.3 RPython

In order to boost the performance of the Python interpreter written in Python, the PyPy translation toolchain was created. The translation from a dynamically typed language like Python to a statically typed language like C is however not straightforward. As mentioned before, in order to do this in a semantically correct way, we are enforced to limit the expressiveness of the input language. For this reason, when we talk about the language understood by the translation toolchain, we do not call this Python but rather RPython or restricted Python. The language is rather imprecisely defined as Python with the following restrictions applied:

- Variables need to be type consistent
- Runtime reflection is not supported
- All globals are assumed to be constants
- All code has to be type inferable

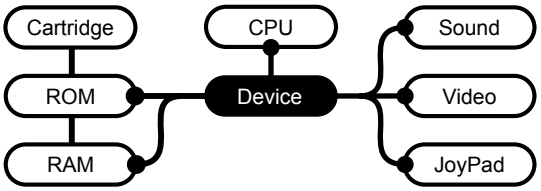
The complete definition of RPython itself is fuzzy as it is defined by the evolution of the translator and the community[3].

Although these restrictions seem to be substantial for a dynamic language such as Python, it is still possible to use high-level features like single inheritance, mixins and exception handling. More importantly, since RPython is a proper subset of Python, it is possible to test and debug the input program with the known Python tools before trying to translate it using PyPy. As such the VM models we build for PyPy are executable by themselves, thus giving a great speedup against a classical compile-wait-test cycle.

[2] `http://codespeak.net/pypy/dist/pypy/doc/garbage_collection.html`
[3] `http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html#restricted-python`

3 Gaming Device Hardware Technical Details

In this section we list the technical details of the gaming hardware. An official documentation is available on the producer's website[4].



3.1 Hardware Pieces

The system is composed of 6 essential pieces which are accessible through shared memory. External events are supported through a 8 bit maskable interrupt channel. The use of an 8 bit processor makes the handling of opcodes compact and maintainable. There are two kind of opcodes:

- First order opcodes are executed directly
- Second order opcodes fetch the next instruction for execution. The combined opcode doubles the range of possible instructions at the cost of execution speed. The second order opcodes are mostly used for bit testing and bit setting on the different registers.

The following list shows some more specific details of the different parts of our gaming device.

- The 8 bit CPU is a slightly modified version of the Zilog 80 with a speed of 4.19 MHz. It supports two power-saving mechanisms, which both work in a similar way. After a certain interrupt, the CPU is put into a low power consumption mode which is left only after another interrupt has occurred.
- The cartridge contains the ROM with the embedded game and possibly additional RAM and/or other devices such as cameras of printers. The size of the RAM depends on the type of cartridge. Some types of cartridge support an additional battery to store game-state. Switchable memory banks are used to extend the 8 bit limited address range. A checksum in the header and a startup procedure are used to guarantee that the device is working correctly and the cartridge is not corrupted.
- The supported Resolution is 160×144 Pixels with 4 shades. It is possible to show maximally 40 Sprites of 8×16 or 16×16 pixels at the same time. The video chip has two tile-map registers, one for the background and one for the foreground. A masking window can be used to crop the background thus enabling basic low-level support for scrolling.
- A serial connection can be used to communicate with another device. This is useful for multiplayer games.
- The sound chip supports stereo sound and has four internal mono sound channels. Sound can be either read directly from the RAM, thus creating arbitrary sound at the cost of its calculation, or it can be produced via a noise-channel or via two different wave pattern generators.

[4] `http://www.nintendo.co.uk/NOE/en_GB/support/game_boy__pocket__color_559_562.html`

4 PyGirl Implementation

In this section we will show some specific implementation details and discuss some problems, which raised during the project.

4.1 Source Implementation

We chose the open source project jMario [2] as basis for our implementation. jMario was built to support different Java runtime environments, thus the core depends on abstract drivers, which are then implemented for each backend differently. jMario features backends for the Standard Edition and for Applets. Each hardware piece is mapped to a class, a design template which we wanted to extend by abstraction.

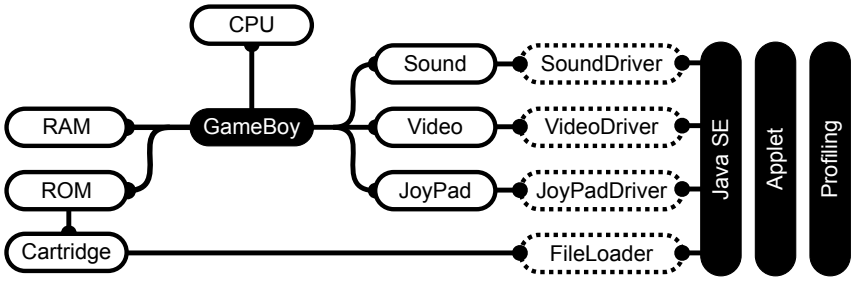


Figure 2: Architecture of the jMario VM with driver sets for different backends.

To handle the different execution times of the original hardware pieces, the emulation uses ticks to synchronize each piece. Each piece has a certain number of ticks available. The central gameboy controller executes the minimum number of available ticks on each piece. If the ticks have been eaten up, a new cycle is started by adding a certain number of ticks to the available ticks. This number has to be a multiple of the smallest number, to guarantee the syncing between the parts.

While the code is written in an object-oriented manner at first glance, many parts of the Java system strictly follow the low-level execution details of the hardware. On top of this, the implementation is cluttered with local optimizations. Two types of optimization strategies clearly stand out: manual inlining of code and manual unrolling of loops. Both strategies result in an overly expanded code-base, obscuring the overall design and semantics.

For example, the CPU class is cluttered with such speed optimizations. The reason is that a CPU is a very low-level general-purpose device which provides fewer possibilities for abstraction than others. Even the video chip is implemented in a non-abstract procedural way. This is the case even though there are more components ready for abstract representation, such as *sprites*, *background* and *foreground* (see Section 4.5 on page 13).

4.2 From Java to Python

I started by porting the Java system without applying any optimizations or Python specific refactorings at first. Although I could have used a tool which transformed the Java sources to Python automatically, I intentionally chose the manual way. This way I could easily learn Python, by converting the code stepwise to a more Python-like sources.

The following sections covers different optimizations and refactorings, which moved the project towards a high-level VM.

4.2.1 Memory Usage Considerations

The Java code is cluttered with type-casts between bytes and integers. Bytes are used to represent the 8 bit hardware architecture, whereas the integers are used in the VM for all kind of calculations. Instead of using integers whenever possible, the Java version focuses on reducing the memory footprint of the running emulator and even more, it focuses on the implementation details of the original hardware. Only at very few places in the code was the use of bytes justified by the resulting two's-complement interpretation of the numbers. Even the consideration of reducing memory is irrelevant on a modern computer, since the original device uses a insignificant amount of memory. Considering the use of four to eight times as much memory as the original device, which corresponds to an expansion from 8 to 32 or even 64 Bit, the resulting footprint won't exceed 20Mb.

In order to increase the readability and generate more flexible code, we removed casts and low-level byte operations wherever possible. The following example shows a part of the original code, using 2s-complemented[5] numbers.

```
public final void jr_nn() {
        byte offset  = (byte) this.fetch();
        this.pc      = (this.pc + offset) & 0xFFFF;
        this.cycles -= 3;
}
```

Listing 1: Java: CPU code for a relative jump, working with byte casting and masking.

The corresponding code in Python with a helper method:

```
def process_2s_complement(value):
        return (value ^ 0x80) - 128

def relative_jump(self):
        self.pc.add(self.process_2s_complement(self.fetch()))
        self.cycles += 1
```

Listing 2: Python: CPU code for a relative jump, working with register objects and a helper function. See Appendix F on page 28 for a short introduction to 2s complemented numbers.

The Python example is much simpler to read than the Java version, and it also reduces a possibly misleading interpretation of the fetched value, which has to be treated as a 2s complemented number, since there is only a single place where the compact notation of bit operation is used to decode those 2s complemented number. The Python version handles the cycle consumption in each object separately. Only exceptions have to be manually specified, unlike the Java version which specifies the consumption in each method separately.

4.2.2 The God Switch

The most prominent candidate for refactoring duplicated and inlined code is the CPU class. In the original version the class is around 4000 loc long, featuring an unpleasant 1700 loc switch which delegates the incoming opcodes.

[5] An example of how to use 2s-complemented numbers can be found in Appendix F on page 28

```
public void execute(int opcode) {
    switch (opcode) {
    case 0x00:
        this.nop();
        break;

    ...

    case 0xFF:
        this.rst(0x38);
        break;

    default:
        throw new RuntimeException(ERR);
    }
}
```

On top of that, there is a nested switch of 800 loc, which handles the second order opcodes.

We build most parts of the switch dynamically at compile-time. In the following excerpt from the original Java code we can see how bytecodes directly encode their semantics in a structured way

```
public void execute(int opcode) {
    ...
    case 0x78:
        this.ld_A_B();
    case 0x79:
        this.ld_A_C();
    ...
}

public final void ld_A_B() {
    this.a = this.b;
    this.cycles -= 1;
}
...
```

Listing 3: Java: Grouped opcode mappings. It is clearly visible that somehow the opcodes encode the argument of the load method. `0x78` corresponds to `B` and `0x79` to `C`

All this hand written repetitive code can be reduced to more abstract patterns. Since the original code encodes parts of the function logic in the opcodes, we could group all the similar functions and iterate over the opcodes, generating function closures. This way the arguments of the more abstract functions are directly mapped from the opcode, instead of using a switch to retrieve this information. Even though the Java methods are only few lines of code each, the dynamic generation reduced the size significally.

The opcodes in Listing 3 encode the arguments which should be passed to the `load` function. Instead of separate functions PyGirl uses a general `load` for different registers so that it can be reused for more than one opcode.

```
def load(self, register1, register2):
   register1.set(register2.get())
```

To refactor Listing 3, we created such reusable functions for all the different types of operations, then simply replaced the whole switch with a compact lookup in an opcode table which is generated at compile-time.

```
def execute(self, op_code):
    OP_CODES[op_code](self)
```

Instead of hardcoding the mapping to the respective functions, we used metaprogramming to compute the definition at translation time. The mapping of opcodes to functions similar to the example Listing 3 on the previous page can be replaced with the following flexible definition.

```
def create_op_codes(table):
    op_codes = []
    for entry in table:
        op_code  = entry[0]
        step     = entry[1]
        function = entry[2]
        for getter in entry[3]:
            op_codes.append(
                (op_code,
                 register_lambda(function,
                                 getter)))
            op_code += step
    return op_codes

def register_lambda(function, register_or_getter):
    if callable(register_or_getter):
        return lambda s: function(s,
                          register_or_getter(s))
    else:
        return lambda s: function(s,
                          register_or_getter)


REGS = [CPU.get_bc, ..., CPU.get_sp]
SET  = [
    (0x01,  0x10, CPU.fetch_double_register, REGS),
    (0x03,  0x10, CPU.inc_double_register,   REGS),
    (0x09,  0x10, CPU.add_hl,                REGS),
    (0x0B,  0x10, CPU.dec_double_register,   REGS),
    ...
    (start, step, func,                      REGS)
    ...
]

OP_CODE_TABLE += create_op_codes(SET)
```

Listing 4: RPython: Opcode generation at preprocessing time using metaprogramming

The `create_op_codes` -function creates a part of the final opcode table from a list of inputs. Each input entry consists of a start opcode, the offset to add to get the following opcode and finally, an ordered list of registers. The input `SET` is a clean and compact notation for the variety of opcode mappings. We could use the static creation of opcode table entries for most of the opcodes. In total we were able to use metaprogramming for about 450 out of all 512 opcodes.

It may seem that the resulting code depending on the lookup in the opcode table and the dynamic function call is slower than a simple switch, but this is only the case for the directly interpreted version of PyGirl. When translating this code, PyPy is able to

take the preprocessed opcode table and translate it back into an optimized switch. So the source code stays compact and maintainable, without loss in performance. Even better, PyPy should be able to optimize the switch at runtime so that often used branches are scheduled first. It is also possible to achieve this already at compile time. PyPy has a dynamic optimization feature, which profiles compiled applications in a first step and creates an optimized program in a second compile run. PyPy can then also inline small methods. Although the author of the Java version didn't inline any method, it would have been the next logical step to optimize the code in a classical way (like in the Java implementation). By letting PyPy handle the optimizations, the sources are more readable and flexible, but still result in a fast program.

4.2.3 Abstraction

Since the Java implementation tries to avoid methods calls, different internal subparts of the hardware device were implemented as simple byte or integer fields rather than separate high-level objects. As mentioned in the previous section we tried to reduce the application of bit operations and repetitive code; thus we could introduce simple objects for handling those original integer fields. Instead of just creating a wrapper around an integer value with some accessor methods, we directly used booleans, which represent each bit of the integer field, giving a logical meaning to each bit.

```
class FlagRegister(Register):
    def __init__(self):
        self.is_zero        = False
        self.is_subtraction = False
        self.is_half_carry  = False
        self.is_carry       = False
        self.p_flag         = False
        self.s_flag         = False
        self.lower          = 0x00

    def get(self):
        value  = 0
        value += ( int(self.is_carry)       << 4)
        value += ( int(self.is_half_carry)  << 5)
        value += ( int(self.is_subtraction) << 6)
        value += ( int(self.is_zero)        << 7)
        return value + self.lower

    def set(self, value):
        self.is_carry       = bool(value & (1 << 4))
        self.is_half_carry  = bool(value & (1 << 5))
        self.is_subtraction = bool(value & (1 << 6))
        self.is_zero        = bool(value & (1 << 7))
        self.lower          = value & 0x0F
```

Listing 5: Python: Example implementation of a flag register which is represented as a single integer field in the Java version.

Although this is quite a lot of code compared to a simple integer field, it brings all the comfort of an object, its named fields and helper methods. The following example shows the code of two flag-related CPU methods in Java and the corresponding ones in Python. The `ccf`-method inverts the carry-flag and resets all the other flags but the zero-flag. The `scf`-method resets all flags except for the zero-flag and sets the carry-flag:

```
public final void ccf() {
    this.f = (this.f & (Z_FLAG | C_FLAG)) ^ C_FLAG;
}

public final void scf() {
    this.f = (this.f & Z_FLAG) | C_FLAG;
}
```

The corresponding code in PyGirl is not that compact, but more readable.

```
def complement_carry_flag(self):
    self.flag.partial_reset(keep_is_zero=True,
                            keep_is_carry=True)
    self.flag.is_carry = not self.flag.is_carry

def set_carry_flag(self):
    self.flag.partial_reset(keep_is_zero=True)
    self.flag.is_carry = True
```

4.3 Translation

This part describes some of the problems we encountered while using the PyPy translator. It has to be said, that I started to use the translator at a rather mature state of the emulator. This had two reasons: on the one hand I thought that it might make more sense to run the translator on fully working emulator, not just on one part. On the other I didn't have much experience in using the translator at the beginning. The first tryouts resulted in a bunch of errors like in Listing 6 on the following page, because unsurprisingly the code didn't compile, due to type incompatibilities. After I had an introduction to the translator at the PyPy sprint in Berlin[6], it was matter of using the Python debugger and a small set of translator specific commands, to find exactly what was going wrong.

Most of the initial errors during translation were caused by typos and inconsistently ported code. Since I started translation rather late and I already tried to adapt the code to the Python programming language, I introduced more dynamism by classes and function closures. Exactly those parts of the code were more difficult to translate, because PyPy has to reduce all dynamic calls for a typed backend. Sometimes the translator was rather helpful to check the code for possible semantic errors, since not all parts of the implementation were working fully.

4.4 Float Injection in a Integer World

Python and Java don't share exactly the same set of basic system functions, amongst others the time function. In Java one simply gets a long value in milliseconds, whereas the Python version returns a floating point number in seconds. At a certain point we finished the implementation of the Timer object, which was a stub returning only a fixed value so far. By accident, we didn't convert the return value to an integer. Instead we injected a float into a fully integer based system. The translation toolchain was unable to locate the error, but instead produced a rather unrelated error (Listing 6 on the next page):

[6] http://morepypy.blogspot.com/2008_05_01_archive.html

```
Blocked block -- operation cannot succeed
 v0 = getitem(([<unbound method CPU.n...9dcb0>]), op_code_0)
            self.instruction_counter += 1
            self.last_op_code = op_code
==>         OP_CODES[op_code](self)
Known variable annotations:
 op_code_0 = SomeFloat()

Blocked block -- operation cannot succeed
 v1 = and_(flags_0, (64))
==>                 if ((flags & 0x40) != 0):
Known variable annotations:
 flags_0 = SomeFloat()
```

Listing 6: PyPy: Translation error output

The injection of the float caused a chain reaction in the integer system. Every contact with the new float value changed an integer field to the more general type float. In the end it is impossible for the toolchain to narrow down the choice of a possible error source.

At some point in the code the integer fields were used again for integer only operations. Exactly at these points the toolchain produced such blocked block error messages. Instead of manually checking the sources, we used PyPy's Flow Graph browser to follow the call graph visually. Like that we could trace back a type ambiguity to its origin, in this case the `Timer` class.

4.4.1 Call Wrappers

We handled operations on registers and other CPU functions elegantly by just passing around function closures, allowing us to reuse methods for multiple actions (see Section 4.2.2 on page 7). A very common example is the following `load` method:

```
def load(self, getter, setter):
    setter(getter())

...

load(self.flag.get, self.a.set)
load(self.fetch, self.a.set)
```

`load` is called with different arguments. In the first example it is used to copy the values from the flag-register into the register `a`. The second example loads the next instruction into register `a`. Due to the different arguments passed to the function closure, the toolchain was unable to transform this into a typed counterpart. As solution we created call-wrappers, as one would do in a typed language, with a common superclass. Introducing call-wrappers for the passed function closures added some overhead, but it was a simple way to keep the code minimal and close to the original idea of passing around closures. The following code shows the same function calls using wrappers for each passed type of closure:

```
class CallWrapper(object):
    def get(self, use_cycles=True):
        raise Exception("called CallWrapper.get")

    def set(self, value, use_cycles=True):
        raise Exception("called CallWrapper.set")


class RegisterCallWrapper(CallWrapper):
    def __init__(self, register):
        self.register = register

    def get(self,  use_cycles=True):
        return self.register.get(use_cycles)

    def set(self, value, use_cycles=True):
        return self.register.set(value, use_cycles)


class CPUFetchCaller(CallWrapper):
    def __init__(self, CPU):
        self.CPU = CPU

    def get(self,  use_cycles=True):
        return self.CPU.fetch(use_cycles)
```

Listing 7: Python: Callwrappers for handling function closures in RPython label

`RegisterCallWrapper` takes a register and calls `get` or `set` on it. The `CPUFetchCaller` is used as an abstraction for the `fetch` method of the CPU so that it can be used as a possible argument for the `load`-method. These are two out of the 5 total call-wrappers we used to handle closure passing. The superclass `CallWrapper` makes it possible for PyPy to infer a common type for the argument of every method. Something important which is not that clearly visible in this code is the fact that every `get`-method of the call-wrapper returns an integer. Thus all methods have the same signature. To support the call-wrappers we replaced the closure calls by a `get` or `set` on the call wrappers. Applied to the `load` method this resulted in the following code

```
def load(self, getCaller, setCaller):
        setCaller.set(getCaller.get())
```

Then the method-calls from the original example look now like the following example:

```
load(RegisterCallWrapper(self.flag),
    RegisterCallWrapper(self.a))

load(CPUFetchCaller(self),
    RegisterCallWrapper(self.a))
```

4.5 Video and SDL

In this section we will show the implementation details of the video chip. The first part covers the high-level optimizations of the video chip and the second shows how the video driver is connected to a backend specific video library.

4.5.1 High-Level Abstraction

As previously mentioned in Section 4.2.3 on page 10 we tried to take advantage of inheritance to simplify the code. Unlike the CPU, the video chip is not a mere procedural calculation machine, rather it is based on an abstract idea. We mapped each part, such as *sprites*, *background* and *foreground*, to a separate object.

The Java implementation works like the original hardware implementation on integer fields as registers, and uses basically two sequential memory areas. The first one holds the graphical data (Tiles) for displaying. The second one (Object Attribute Memory) stores the attributes for each displayed Sprite, such as the position and the tile. The painting mechanism sorts and filters the sprites depending on their position and attributes. To express this procedure, the Java implementation has to iterate several times over the OAM and mask out the necessary values. Since the attribute set of a sprite covers 4 consecutive bytes, not only one, each access is complicated again. To simplify and clean up the code the newly created objects `Sprite` and `Tile` directly kept these parts and attributes themselves.

A second refactoring step covered the different states of the video chip. For example, during the painting of the sprites and the swapping of the video ram contents, the video memories cannot be accessed. There are certain restrictions during each of these states, which last only for a specific amount of time. The Java version handles the different modes by accessing the status register and cluttering the code at several points with switches for handling the different functionality. We tried to push as much as possible into separated `Mode` objects. The modes are built up similar to a single linked list. Each mode can jump to the next one by a single call. This method selects the next mode, depending on the current status, and activates it.

4.5.2 Lib SDL

To support graphics for the C backend, the PyPy people chose to use libSDL[7]. In order to have the functionality available at the interpretation level and in the final binary a special setup file, which specifies the mapping of types, constants, and functions added by the external library, is required.

The following example (Listing 8) shows an excerpt from the `RSDL` mapping file, used by the PyPy toolchain to enable graphics support for the C backend:

```
SurfacePtr     = lltype.Ptr(lltype.ForwardReference())
PixelFormatPtr = lltype.Ptr(lltype.ForwardReference())

class CConfig:
    Uint32       = platform.SimpleType('Uint32', rffi.INT)

    Surface      = platform.Struct('SDL_Surface',
                                   [ ('w', rffi.INT), ('h', rffi.INT),
                                     ('format', PixelFormatPtr),
                                     ('pitch', rffi.INT),
                                     ('pixels', rffi.UCHARP)])

     PixelFormat = platform.Struct('SDL_PixelFormat',
                                   [ ('BitsPerPixel', rffi.INT),
                                     ('BytesPerPixel', rffi.INT),
                                     ('Rmask', rffi.INT), ('Gmask', rffi.INT),
                                     ('Bmask', rffi.INT),('Amask', rffi.INT)])
```

[7] `http://www.libsdl.org/`

```
SurfacePtr.TO.become(Surface)
PixelFormatPtr.TO.become(PixelFormat)

CreateRGBSurface = external('SDL_CreateRGBSurface',
                            [ Uint32, rffi.INT, rffi.INT, rffi.INT,
                              Uint32, Uint32, Uint32, Uint32
                                                          ],
                            SurfacePtr)
```

Listing 8: RSDL: Excerpt from the `pypy.rlib.rsdl.RSDL` module

The `CConfig` class defines the structs used by SDL, defined by a name and list of fields. The `XXXPtr.TO.become()` methods are used to tell each Pointer to which type it is pointing. After the Pointers are set, the external functions are defined, by setting a name and a list of types for the arguments.

Externals defined in such a way can be directly accessed from the interpreted Version and are translated directly to the C counterparts, when compiling to the C backend.

The code in Listing 8 on the previous page has to be considered as mapping from Python types and functions to C counterparts and vice versa. This part required some deeper knowledge about PyPy, thus we depended on the support of the community in order to get an example implementation. Thereafter the rest of the `RSDL` implementation was a simple mapping task, requiring the documentation of libSDL related to the type and function definitions.

4.6 State of the Implementation

The current version supports simple test ROMs without sound and moving sprites. Static images and bit-blitting are supported. This means that most games are able to show an intro screen but there is no animation. In the current version, the video part is heavily redesigned, using high-level definitions for all the logical parts in the video chip. Because we mostly focused on the graphics part, the sound chip has not yet been implemented for PyGirl. This is the case even though PyPy's C-backend would already support sound via libSDL. The CPU is fully functional.

5 Debugging

In this section we discuss the debugging techniques which were used.

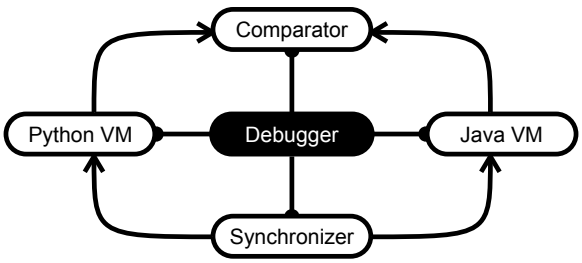
5.1 Double Testing

When I was implementing the CPU I didn't have tests at the beginning. Thus I was more or less blindly implementing the CPU in Python. Later on, with the system running for the first time, I had to start writing tests for the CPU in order to eliminate bugs. The testless early development cycles weren't a brilliant choice, since I apparently reduced not only duplicated code but also code that looked nearly the same but wasn't. Thus some parts were simplified too much and reused again. This and the fact, that there was already quite some abstraction introduced by using more object oriented features, made it really hard to track the errors by hand. At this point I could only rely on the unit tests and no more on the visual comparison of the source files. Since at that point I had only little knowledge of the gaming device, the first series of tests for the CPU didn't cover the flag register, which normally stores exceptional behaviours like overflows by setting a bit. As the project reached a more mature state, allowing us to run simple ROMs, the carelessness resulted in strange responses of the emulation. This caused hours of exhausting debugging, since the visual feedback of the emulation was no more directly connected to the execution of an operation and the errors normally occurred first after 2500 or up to 32000 opcodes executed.

Finally we decided to create a second test set with slightly different tests, built from a simple specification text, which was a merge of all sort of freely available documentation in the web. In contrast to the existing tests, we now explicitly tested the flag output for each operation.

5.2 Remote Testing

After the CPU was double tested, but the system as whole still remained buggy, we tried another approach. Instead of checking the sources and comparing line per line with original Java version, we relied on the existence of a working version. In order to check which parts of the Python implementation were defective we wanted to run both implementations side by side and compare their memory and register values after each step, thus we created a remote debugger. This debugger suite consists of 5 essential parts. There are the two versions of VM's we compare, the original one in Java and the implementation in Python. Those two parts are started by the Debugger which is programmed in Python, working as an RPC-XML server for the Java version. After the Java version has started up, it connects to the debugger and syncs its state through RPC-XML calls.



Since the Python version runs in interpreted mode, it is much slower than the Java version (see Section 6 on page 18), thus to sync these two versions, the Java version has to wait after each execution. After each step, the Java version checks whether the Python version is ready or not. Since the RPC-XML calls are blocking, this is a fairly

easy task. The Java version calls and the synchronizer loops until the Python version is ready, then the call returns and the Java version can resume its execution. The most important part is the comparator. Since Java and Python cannot interchange types directly, each register and memory area has to be serialized on the Java side. After transmitting the data, it is compared with the local values of the Python version.

```
loading rom:    pypy-dist/pypy/lang/gameboy/rom/rom9/rom9.gb
python:                 waiting for client to start
------------------------------------------------------
Mario GameBoy (TM) Emulator Version 0.2.4 (Build 2006-09-23)
Copyright (C) 2006-2007 Carlos Hasan. All Rights Reserved.
------------------------------------------------------
Title: rom9 Type: MBC1+RAM ROM: 32KB RAM: 32KB
------------------------------------------------------
python: called start
python:    waiting for client to send rom
================================================================================
0   EXEC: 0 | 0x0 | nop
--------------------------------------------------------------------------------
>>  enter skip, default is 0:
 a:   1 |  f: 128 |  b:   0 |  c:  19 |  d:   0 |  e: 216 |  h:   1 |  l:  77 | sp: 65534 | pc: 257 |
      1 |     128 |       0 |      19 |       0 |     216 |       1 |      77 |     65534 |     257 |
fetch: 195

================================================================================
1   EXEC: 195 | 0xc3 | jump
--------------------------------------------------------------------------------
>>  enter skip, default is 0:
 a:   1 |  f: 128 |  b:   0 |  c:  19 |  d:   0 |  e: 216 |  h:   1 |  l:  77 | sp: 65534 | pc: 336 |
      1 |     128 |       0 |      19 |       0 |     216 |       1 |      77 |     65534 |     336 |
fetch: 243

...

================================================================================
2750   EXEC: 240 | 0xf0 | store_memory_at_axpanded_fetch_address_in_a
--------------------------------------------------------------------------------
python: !! video line value at 0x2a expected: 0 got: 1 !!
python: !! video line value at 0x33 expected: 0 got: 1 !!
```

Listing 9: Remote debugger: Output and example commands of the debugger. Exection 2750 outputs erros because of different values in the video RAM, resulting form a bug in the video chip implementation of the Python version.

In order to handle the errors after a certain amount of execution cycles, one can specify the number of initially skipping steps. If the compared values differ in a step, each difference is outputted and one can either launch the debugger to inspect the system, or continue with the execution. To inspect the Java version as well, we normally launched the debugger alone, and plugged in the Java client afterwards. This allowed us to use an IDE to track the changes on Java side.

5.3 High-Level Abstraction

Another way to debug the implementation which also agrees with our design gaols, is to introduce more abstraction. The reduction of bit operations is one step, the other is to introduce more high-level objects and thus clarify the intention of the program. Especially the video driver originally was filled with cryptic code, which filters the sprite at first and afterwards draws each Sprite by copying memory pieces directly. After refactoring the code, we could use the high-level objects to see what was actually happening. Using the Python Debugger we could manually check parts of the system by accessing the high-level VM objects. In order to achieve this at runtime in the Java version, each access of the Sprite attributes and data is coupled to a calculation of the memory location and a decoding of the information at that place, which is rather annoying. Since we assumed that the Java version is working correctly, we mostly depend on the automatic comparison of the memory contents and the objects.

6 Performance Evaluation

In this section we compare different implementations of the gaming device. We ran the emulation on three different flavours of the VM. The original Java emulation, the interpreted variant of our implementation and finally the translated version. The interpreted version was run on top of the Python 2.5 Interpreter[8], whereas the translated version was built from those sources.

The benchmarks are shown for the original Java implementation, the interpreted sources of PyGirl, and finally the translated binaries of PyGirl. Each test shows the average execution time over 1000 runs using Java `1.6.0_06`, `1.5.0_15` and Python `2.5.2` on a 64 Bit Ubuntu `8.04.1` server machine with an Intel Xeon CPU QuadCore 2.00 GHz processor. We use revision number `59328` of the PyPy project. PyPy uses the following GCC optimizations:

```
-O3 -fomit-frame-pointer -pthread -I/usr/include/python2.5
```

The complete GCC calls are visible during the end of the translation process. Instructions for running the benchmarks are given in Appendix D on page 26.

6.1 All Opcodes

This benchmark covered all possible opcodes which were executed equally distributed, giving a more theoretical overall-speed of the VM. The results of this benchmarks are shown in Figure 3.

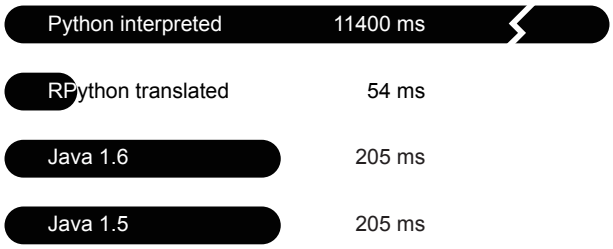


Figure 3: Benchmark: Comparing the execution time of PyGirl with the Java version weighting all opcodes equally. Each opcode was sampled 1000 times.

It is clearly visible that the interpreted version running on top of CPython is approximately 100 times slower than the Java version and 250 times slower than the translated version. This test compares the overall performance of the system, which is not necessarily directly related to the performance of the emulator running a game.

6.2 Typical Opcode Set

To create a more realistic test case, we profiled different games and created a list of the most frequently used opcodes. Those opcodes, listed in Table 4 on page 27, were then used to create the results in Figure 4 on the following page.

The results differ from the first benchmark, which has its origins in the different relative execution times of opcodes in the Java and in the translated version. Java also was able to optimize the code at runtime, using a JIT. While this will also be available for VMs using PyPy, it is currently still in development [5]. Comparing the execution time of each opcode, a steady performance optimization is visible in Java. The different distribution of the executed opcodes in this benchmark (Table 4 on page 27) allows Java

[8] `http://python.org/download/`

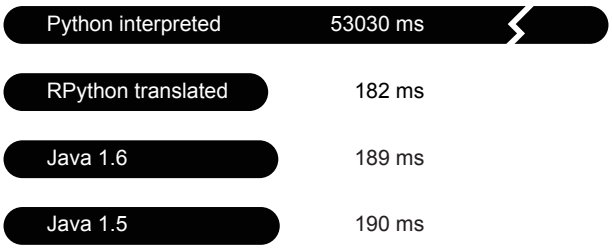


Figure 4: Benchmark: Comparing the execution time of PyGirl with the Java version weighting the opcodes according to a typical emulation run. The set was run only once.

to further improve the result. Java can optimize the most frequently executed opcodes, which weigh heavier on the total result in this benchmark than in the first one.

6.3 Opcode Comparison

In this test we compared the execution time of each opcode. This test reveals a cause for the different result from the preceding tests. Table 1 shows the results for the 30 most executed opcodes (see Appendix E on page 27).

Name	Opcode	Executions	Python	RPython	Java 6	Java 5
Reset Program Counter	`0xff`	45%	0.254	0.0014	0.0013	0.0013
Conditional Jump	`0x20`	13%	0.324	0.0013	0.0014	0.0017
Memory Read	`0xf0`	8%	0.281	0.0011	0.0020	0.0020
Compare	`0xfe`	8%	0.387	0.0013	0.0035	0.0035
Memory Write	`0x13`	4%	0.227	0.00072	0.0015	0.0015
Increment Register	`0x12`	4%	0.183	0.00064	0.0028	0.0029
Memory Read	`0x2a`	4%	0.267	0.00089	0.0029	0.0029
OR	`0xb1`	4%	0.267	0.00082	0.0020	0.0020
Decrement Register	`0x0b`	4%	0.419	0.0017	0.0080	0.0080
Register Write	`0x78`	4%	0.208	0.00079	0.0012	0.0012
Decrement Register	`0x05`	1%	0.271	0.00079	0.0047	0.0047
Memory Write	`0x32`	1%	0.261	0.00089	0.0028	0.0028

Table 1: The ten most executed opcodes from the typical set (Table 4 on page 27) evaluated on three different systems. All values are given in Seconds of execution time of 10'000 execution per opcode each averaged over 1000 runs.

In Table 1 one can see, that the RPython version and the Java version normally have results of the same magnitude. Both Java version run at the same speed, unlike on Mac OS X, where the immature state of the 1.6 implementation is clearly visible through some heavy differences in the execution times. Since most of the benchmarked operations are simple integer operations, it should behave more or less the same in both versions.

The translated Python version has about the same speed on the two most executed opcodes. With the third most executed one, the gap between the translated Python version and the Java version grows. In future versions, the differences between those versions will grow, since the JIT is not yet included in the sources used for this benchmark.

6.4 JIT Comparison

In order to show the impact of a JIT to the results of the benchmarks, we run in this section the equally distributed opcode set of Section 6.1 on page 18 with different execution counts. As the number of execution grows, the difference between the Java and the translated Python version shrinks, and eventually the Java version gets faster than the translated version.

Executions	RPython	Java 6	Ratio
1	0.00017	0.00058	0.30
10	0.00067	0.0036	0.19
100	0.0054	0.030	0.18
1000	0.054	0.21	0.26
10'000	0.54	0.82	0.65
100'000	5.4	3.1	1.74
1'000'000	54.0	25.4	2.13
10'000'000	540	248	2.18
100'000'000	5400	2462	2.19

Table 2: Influence of the JIT to the benchmark results. Average execution in Seconds over 10 runs per test. For this benchmark the equally distributed set has been used. The last two values for RPython are estimated by linearly extending the previous values

Table 2 shows the ratio of both execution times. It is clearly visible that the translated RPython version runs linear, whereas the Java version behaves differently. Other than expected, the Java version gets hot after a rather high amount of executions. We expected around 10'000 executions, but running this benchmark with finer grained steps resulted in 300'000 executions for warming up the JIT. Although the Java version features a JIT, it is only around two times faster than the our implementation. This difference is negligible with future versions of PyPy featuring a JIT.

7 Future Work

This section covers the future tasks necessary for completing PyGirl.

7.1 PyGirl

The current implementation is not fully usable yet. First of all, there is no implementation of the sound unit available in the emulation. The main goal is to get an emulation with working graphics and inputs. From all the hardware parts which already have a software counterpart, only the video driver is not working properly yet. It is currently able to display simple graphics and supports most operations. Still there are some hidden bugs, which occur only after a certain combination of instructions. Most of the bit-operations have been replaced by more readable boolean comparisons, as described in Section 4.2.3 on page 10. Moreover a set of objects has been added to represent high-level objects such as tiles, sprites, a masking window and a background. The graphic part, however, still relies mostly on pixel operations and raw memory access. In order to create a fully object-oriented implementation, a replacement of the painting mechanism to work directly on sprite objects rather than the memory, is desired.

7.2 PyPy

The current state of the implementation only allows the code of PyGirl to be translated using the C-backend. This is due to the fact that the I/O drivers for PyPy have only been completed for that backend. Since the Implementation relies only an a primitive set of functions, like bit blitting and raw sample output for the sound chip, this makes it straightforward to enable graphics support for the other backends as well.

The current video implementation of PyPy targets only the C backend. Currently the video driver is cluttered with specific code for lib SDL, such as initialization and event handling. In order to keep a clean implementation there should be an abstract video and sound driver which is then specialized for each backend.

Translating PyGirl with the JVM as target would make it possible to compare the performance of the original Java implementation directly with my approach. Another important factor is the availability and wide spread distribution of the Java Virtual Machine. It would be interesting to be able to run PyGirl on a mobile device [1]. Since the Java Mobile Edition is for the biggest part a subset of the Standard Edition, the toolchain needs only minor changes. Because our implementation does not rely on features which are not available on the Mobile Edition, PyGirl should be translatable in the same way to C, Java SE and Java ME.

8 Conclusion

In this Bachelor thesis we have shown that it is possible to create fast whole system emulators in a dynamic language using a high-level compiler PyPy. In this concluding section many of the problems and implementation aspects are reviewed.

8.1 Sources and Design

The initial project was built around 32 classes with a highly varying number of lines of code, reaching up to 4000 in the original CPU class, whereas the RPython implementation uses 16 Python modules with around 70 classes. The Python version has an additional set of 18 test files.

	Java	Python
number of files:	55	36
number of classes:	32	70
number of testfiles:	—	18
number of non-empty testlines:	—	4762
number of non-empty lines:	9464	5344
total lines of code	9464	10106

Table 3: Source code comparison of the Java and the Python implementation. The debugger implementation and the profiling extensions are counted in both implementations. The Java version includes the files for two different backends.

The number of classes in relation to the lines of code shows clearly the two different design approaches. On the one hand there is the procedural approach using the classes more or less for grouping functions, and on the other hand there is the high-level object oriented approach taking advantage of inheritance to clarify the code. Our implementation is much easier to understand and maintain. It shouldn't require an deep knowledge of the actual device implementation to understand what is going on.

8.2 PyPy Toolchain

The PyPy toolchain is a powerful compiler but also very complex, consisting of an nearly unmanageable number of classes. Although there is quite a lot of documentation on the PyPy website, very often the big picture is missing in these explanations. We made the biggest progress when getting help from the community either at sprints or in discussions on the IRC channel. After the first steps it was rather easy to manage the project with PyPy due to the fact that only a minimal amount of code relied on deeper PyPy knowledge. So the rest of the implementation could be completed independently.

8.3 Performance

The results of the performance evaluation were in the expected range. Still we would like to be faster than the original version. Since PyPy's JIT is in still in development we will have to wait for some time to get another performance boost, without cluttering the code with low-level optimizations. Still the Java version is in the end only around two times faster than our implementation. Comparing the initial values, where the JVM isn't hot yet, PyGirl could be around four times faster than the current version.

8.4 Personal Experience

In the beginning my personal image of the PyPy toolchain was rather unclear. Although I read quite some documentation on the website, only the raw general purpose of the toolchain was clear. The main problem about the documentation was that each subsystem of the toolchain is explained separately in depth. So it was rather hard to figure out the connections between each of these parts.

This view didn't change a lot due to my first porting approaches, since I didn't spend a lot of time using the toolchain. A big help was the visit in Berlin, joining the sprint there. I enjoyed this week in berlin very much and could improve my knowledge of this huge system by the direct help of the community. The sprint gave me a lot of new inputs, pushing my project further in the direction of a working version. The week in Berlin and the weeks after were rather successful: I made the code compile as a whole and introduced high-level object oriented code by refactorings. But as soon as the video driver was working and produced an output, I focused on getting the system fully working instead of optimizing and cleaning it. After the Berlin sprint I felt quite optimistic about the whole project, but analyzing the progress a few weeks later forced me to draw another picture. I mainly tried to solve the bugs concerning the video driver, which consumed most of the time. Most of these trial and error approaches didn't result in any improvement. From then on I tried to create and extend a debugger and optimize the code. It was interesting working on the debugger, since I could gain some experience in program communication between different architectures.

Acknowledgments

I would like to thank Laura Creighton for sponsoring my trip to Berlin and the visit to the S3 workshop and the PyPy sprint.

I gratefully thank Toon Verwaest for supporting my work and reviewing the thesis.

References

[1] Ludovic Aubry, David Douard, and Alexandre Fayolle. Case study on using pypy for embedded devices. Technical report, PyPy Consortium, 2007.

[2] Carlos Hasan. Jmario, 2007.

[3] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.

[4] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium, OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.

[5] Armin Rigo Samuele Pedroni. JIT compiler architecture. Technical report, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[6] James E. Smith and Ravi Nair. *Virtual Machines*. Morgan Kaufmann, 2005.

List of Figures

Listings

A How to run PyGirl

PyGirl requires Python version 2.5 or higher in order to run the RPython version a working version of PyGame is needed. The sources are available on the codespeak svn-repository. To run the project you have to checkout the full PyPy sources:

```
> svn co http://codespeak.net/svn/pypy/dist pypy-dist
> cd pypy-dist
```

To translate PyGirl you have to go to the translation-goal folder and run the translation script:

```
> cd pypy/translator/goal
> ./translate.py --gc=generation --batch targetgbimplementation.py
```

To see more translation options type `./translate.py --help`, also a more details description of the translation options can be found on the documentation website[9]. After translating you can run the binary file with one of the test ROMs which are located in `pypy/lang/gameboy/rom`

```
> ./targetgbimplementation-c ../../lang/gameboy/rom/rom8/rom8.gb
```

To stop the emulation hit the ESC-key.
The full project's sources are located under `pypy/lang/gameboy`

B How to run the Test-Cases

To run the test-cases make sure you checked out the latest version of the PyPy distribution, which includes the PyGirl project. Also assure that `pypy-dist/py` is in your Python include path. Go to `pypy/lang/gameboy` and run `py.test`

```
> cd pypy-dist/pypy/lang/gameboy
> py.test
```

C How to run jMario

To run the original Java version, you have to download the sources from the sourceforge project website

`http://sourceforge.net/projects/mario/`

Or you can download the sources directly from the CVS repository

```
> cvs -z3 \
      -d:pserver:anonymous@mario.cvs.sourceforge.net:\
      /cvsroot/mario co -P jmario
```

To run the script you have to change to the jMario folder and run the ant build file.

```
> cd jmario/
> ant
```

[9] `http://codespeak.net/pypy/dist/pypy/doc/config/`

Then the compiled jar is available in the build folder. Type the following commands to run the jar with a test-ROM. Simple test-ROMs are included in the PyPy distribution. Test-ROMs number 8 and 9 have graphical output, the other only server for testing the opcodes

```
> cd build/
> Java -jar jmario.jar path/to/your/testROM.gb
```

D How to run the Benchmarks

In this section we show how to use the benchmark tools to reproduce the results from Section 6 on page 18. Since PyPy is an evolving system, the results of the newest version might be different from the ones presented in the benchmark section.

In order to run the benchmarks properly you have to download the sources first. Since the profiler depends on a certain structure, the PyPy project and the jMario are packaged in this repository. Also the jMario implementation has an extended build file and some modified sources, targeting the benchmarks.

```
> svn co https://www.iam.unibe.ch/scg/svn_repos/Students/cami/pyGirl
> ls profiling/
    ...
    profileAll.sh
    profileEach.sh
profileJIT.sh
    profileTypical.sh
...
```

To run a benchmark, execute one of the listend bash scripts. Each script takes the number of test probes and the number of executions as arguments. The execution is then averaged over the number of test probes. To run 10 test probes which execute each test 100 times, type the following:

```
> profiling/profileAll.sh 10 100
-------------------------------------------------------
     java1.6
-------------------------------------------------------
BUILD SUCCESSFUL
Total time: 2 seconds
    Starting 10 test runs: ..........
    Concatenating results
```

The results are stored into files under `ARCHITECTURE/result/PROFILE_XYZ.txt`.

```
> cat profiling/java1.6/result/PROFILE_All.txt
    0.2054  0.0030
```

So this test took 0.2054 Seconds and the standard deviation over the 10 test runs is ± 0.0030 Seconds.

Each of the profiling scripts automatically creates the binaries necessary to run the benchmarks. To create the binaries manually, either run the ant script in the jMario project with the profiling target,

```
> cd jmario; ant dist-profile
```

or the profiling target in the PyPy project, which needs a running version of the developer version of Python installed.

```
> cd pypy-dist/pypy/translator/goal/
> ./translate.py --gc=generation targetgbprofiling
```

Further instruction on translating the sources can be found on the PyPy website under[10].

E Performance Evaluation

The list Table 4 shows the the 30 most executed opcodes summed over four different Games. Each game was allowed to execute 500'000 operations. Table 4 shows the results of this analysis, which we used to compare the different VM implementations (Section 6 on page 18).

Name	Opcode	Count
Reset Program Counter	0xff	911896
Conditional Jump	0x20	260814
Memory Read	0xf0	166327
Compare	0xfe	166263
Memory Write	0x13	74595
Increment Register	0x12	74582
Memory Read	0x2a	72546
OR	0xb1	70495
Decrement Register	0x0b	70487
Register Write	0x78	70487
Decrement Register	0x05	24998
Memory Write	0x32	24962
Relative Conditional Jump	0x38	4129
Decrement Register	0x0d	3170
Memory Write	0x22	1034
Call Subroutine	0xcd	308
Fetch	0x21	294
Return Subroutine	0xc9	292
Push Register	0xf5	284
Pop Register	0xf1	282
Jump	0xc3	277
Memory Write	0x77	275
Memory Read	0x7e	261
Increment Register	0x3c	260
Write Memory	0xe0	88
Register Write	0x3e	55
Write Memory	0xea	47
XOR	0xaf	45
Memory Write	0x70	40
Register Write	0x7d	40

Table 4: Typical set of the most executed opcodes, summed over four different emulations.

To run the performance evaluations for the Python version you first have to translate the profiling target.

[10] http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

```
> cd pypy/translator/goal
> ./translate.py --gc=generation --batch targetgbprofiling.py
```

After you can run the different profiling targets by passing in a number and the number of times each test is repeated. If you like to compare the test with the uncompiled version, try to run the compile target itself directly.

```
> cd pypy/translator/goal
> Python targetgbprofiling.py 1 10
```

There are 3 different profiling version.

runs the test on all opcodes equally distributed

runs the test on a typical set of opcodes (Appendix E on the previous page)

runs the test on each opcode separately

the second argument is the number of times each test is sampled before the results are printed out. Use Appendix D on page 26 for extended benchmarking.

F 2s Complemented Numbers

The 2s complemented 8 bit numbers have a range from -128 up to 127 which corresponds to `1000 0000` and `0111 1111` in binary notation. So the most left bit is counted negative. Taking the example from Listing 2 on page 7

```
def process_2s_complement(x):
        return (x ^ 0x80) - 128
```

The expression in parenthesis (`x ^ 0x80`) inverts the 8th bit of the incoming signed 8-bit integer. That means, that -128 (`1000 0000`) is mapped to 0 (`0000 0000`), and 127 (`0111 1111`) is mapped to 255 (`1111 1111`). It should be clearly visible, that one needs to subtract 128 from these values to correspond to the correct signed values. Speaking in terms of 32-bit integers, this corresponds to an addition of (`1111 1111 1111 1111 1111 1111 1000 0000`). Thus if the initial value has been negative, the 8th bit is unset and the first part of the 32-bit integer stays the same. On the other hand if the initial value was positive, the first bits are all flipped, since we assure with ()`x ^ 0x80`) that in this case, the 8th bit is set.