PyGirl: Generating Whole-System VMs from High-Level Prototypes using PyPy*

Camillo Bruni   Toon Verwaest

Software Composition Group
University of Bern – Switzerland

Abstract. Virtual machines (VMs) emulating hardware devices are generally implemented in low-level languages for performance reasons. This results in unmaintainable systems that are difficult to understand. In this paper we report on our experience using the PyPy toolchain to improve the portability and reduce the complexity of whole-system VM implementations. As a case study we implement a VM prototype for a Nintendo Game Boy, called PyGirl, in which the high-level model is separated from low-level VM implementation issues. We shed light on the process of refactoring from a low-level VM implementation in Java to a high-level model in RPython. We show that our whole-system VM written with PyPy is significantly less complex than standard implementations, without substantial loss in performance.

Keywords: PyPy, Python, RPython, whole-system virtual machine, translation toolchain, high-level language, compile-time meta-programming Game Boy, 8-bit CPU

1 Introduction

The research field revolving around virtual machines (VMs) is mainly split up in two large subfields. On the one hand we have the whole-system VM (WSVM) domain focusing on new ways to build and optimize emulators for hardware devices. These VMs mimic closely the actual hardware which they are emulating. On the other hand we have the language domain focusing on building high-level language VMs (HLLVM). These VMs *only* exist virtually. There are no hardware counterparts which natively understand the code running on those VMs. Although both domains share conceptual and implementation similarities, only recently has awareness been growing about the overlap of ideas and acknowledgement that the two fields can enforce each other. As a clear example of this fact we see that modern VM books discuss both fields [12].

Historically, the two fields developed independently of each other. Therefore the tools and techniques that are used in each of them are very different.

* In Proceedings of TOOLS Europe 2009, LNBIP 33 p. 328-347, © Springer-Verlag, 2009.

Although concepts for performance enhancement, like just-in-time compilation, are used in both fields, especially the infrastructure and tools used to realize the systems are very different.

An important progress that recently emerged in the field of language virtual machines is the use of higher-level models for describing virtual machines. Final VMs are then generated from these prototypes and enhanced by specific, low-level optimization techniques. This approach has been realized in the PyPy project. The PyPy project aims at building a complete high-level Python VM in Python rather than in a low-level language like C. All VM implementation details such as garbage collection and JIT compilation are excluded from this prototype. Performance and VM specific details are then reintroduced by applying several model transformation steps from the Python sources to a highly optimized target VM.

In this paper we report on our experience using the PyPy translation toolchain[2] to prototype WSVMs in a high-level language without sacrificing too much performance in the resulting VM, resulting in only about 40% slow-down compared to a similar WSVM in Java. By separating the high-level VM prototype from low-level implementation details we reduce code complexity. The high-level transformations provided by the PyPy toolchain ensure that the performance of the resulting VM is preserved.

Our case study is a custom high-level VM prototype similar to the Squeak VM *SPy* [3]. SPy is written in RPython as a clean high-level implementation and uses the PyPy toolchain to reintroduce all VM implementation details. Rather than implementing a HLLVM we concentrate on emulating a Game Boy. We port an existing Java implementation of the virtual machine, Mario [5], to RPython. We focus on building a high-level and abstract but executable prototype rather than an inflexible and early optimized system. We then show that by using the PyPy toolchain we are able to generate performant low-level virtual machines from those prototypes.

The main contributions of this paper are:

- We show how the execution and implementation details of WSVMs are separated in the same way as those of HLLVMs.
- We show how the use of preprocessing-time meta-programming minimizes the code and decreases the complexity.
- We provide a sample implementation of a WSVM prototype for PyPy which exhibits a simplified implementation without substantial loss of performance (about 40% compared to a similar WSVM in Java).

The remainder of this paper is structured as follows. In Section 2 we give an introduction to the PyPy project. Section 3 covers the technical details of the Game Boy, followed by the actual implementation details of PyGirl in Section 4. In Section 5 we compare the performance of the different WSVMs implementations. In Section 6 the future work is discussed. Finally in Section 7 we provide a brief overview of our achievements.

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

2 PyPy in a Nutshell

In this section we describe the PyPy project, which produced the toolchain that we use to translate our VM model. The PyPy toolchain transforms high-level prototypes into highly optimized executable binaries that incorporate all needed general VM features.

The initial goal of PyPy was to write a full-featured, customizable and fast interpreter for Python written in Python itself, in order to have the language described in itself, *i.e.* a meta-circular interpreter. Running an interpreter on top of another interpreter results in execution so slow as to be almost useless. PyPy addresses this by providing a "domain specific compiler", a toolchain that translates high-level VM prototypes in Python down to executables for different back ends, such as C/Posix [10]. Just like the interpreter, the toolchain itself is written in Python.

The effort of generating a VM from a model in the language itself is similar to other self-sustaining systems such as Squeak where the VM is written in Slang. Slang is a subset of Smalltalk [4] which can be directly translated to C. 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 [6]. For instance, in the Squeak VM exception handling is manually added to check some bit flags each time after returning from a function call. Not only is this a tedious task, but can also easily result in bugs by omitting a manual check. Using PyPy simplifies this task, since it is possible to use high-level exception handling in the VM prototype which eventually gets translated down automatically to something similar in the executable.

There are many examples of high-level language virtual machines (HLLVM) that are realized using higher-level languages. Jikes [7] and earlier Jalapeño [1] realized a complete, modern optimizing Java virtual machine in Java. Inspired by the idea of Squeak, the Squawk[3] Java VM [11] realizes a Java virtual machine in Java. The virtual machine is implemented in Java and then translated ahead-of-time to an executable. Klein [14] was a research project to explore how to realize and especially bootstrap a virtual machine for the dynamic prototype based language Self [13].

In all these examples the virtual machines realized were language virtual machines rather than virtual machines simulating real hardware. To our best knowledge none of the used frameworks and tools have been evaluated in the context of hardware VMs.

2.1 The Interpreter

The starting point for PyPy is to create a minimal but full interpreter for Python written in Python itself. By minimal we mean that all interpreter implementation

[3] `http://research.sun.com/projects/squawk/`

details such as garbage collection and optimizations are not implemented but provided by the environment running the interpreter. This results in a very clean and concise implementation of the interpreter, modelling how the language works without obscuring it with implementation details.

2.2 The Translation Toolchain

In this subsection we describe PyPy's translation toolchain, a "domain specific compiler" that translates VM prototypes to executables binaries. The translation of high-level VM prototypes to low-level back ends is necessary since the prototypes do not run fast by themselves. The Python interpreter written in Python running on top of standard CPython runs code about 2000 times slower than CPython.[4]

The PyPy translation toolchain is designed as a flexible toolchain where front and back ends can be replaced so that it can generate VMs for different languages running on different platforms. Not only the front and back end can be changed, but also the set of transformations applied during the translation process. This results in fast and portable VMs. The following figure shows how prototype VM models can be translated to different back ends:

This overall architecture of PyPy is shown in Figure 1. On top we have the front end or VM prototype which is the input to the translation toolchain. As output we get a self-containing VM which contains all the required implementation details. This VM is compatible with one of the many back ends which PyPy targets. From here on we will discuss all the steps which the toolchain undergoes to go from prototype to back end specific VM.

In short, the PyPy toolchain builds a dynamically modifiable flow graph from the target program's sources. Then this prototype is transformed in several steps until a final binary results. Using the same intermediate representation for a large part of the translation and applying transformations in small steps allows us to customize every translation aspect.

Translation Steps In the first step the translator loads the source-code it translates. Unlike standard compilers that start by parsing the source code, the PyPy toolchain never accesses the Python source code. The toolchain can use its hosting Python interpreter to load its input files. This is because 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. While loading, the Python interpreter evaluates all top-level statements and adds the loaded method and class definitions to the global Python memory. Afterwards the toolchain uses the the globally loaded `main` function as entry point for its input graph.

This setup allows the input prototype to apply meta-programming in plain Python code at preprocessing time. Only the object graph *resulting* from meta-programming has to be RPython compatible (Listing 2 shows an example of this feature).

[4] `http://codespeak.net/pypy/dist/pypy/doc/faq.html`

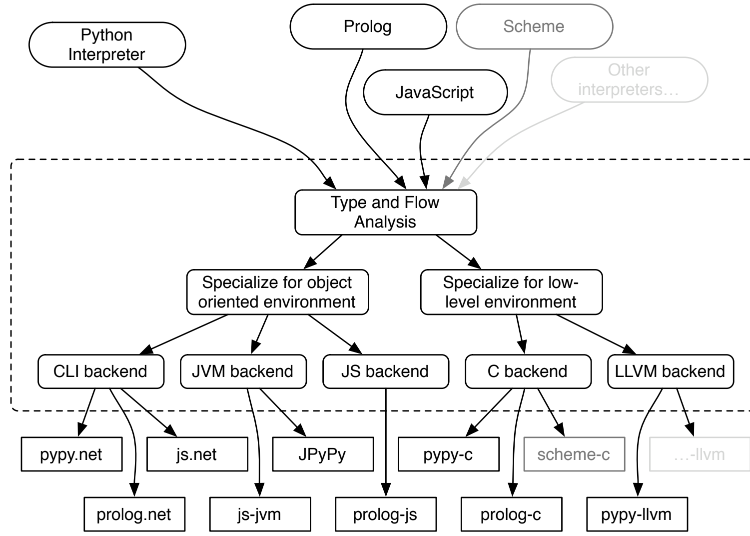


Fig. 1. PyPy translation toolchain architecture

Because PyPy mostly targets statically typed back ends the graph is annotated with inferred types. Starting with the specified entry point, the type inference engine works its way through the object flow graph and tries to infer most specific types. If multiple types are possible for a certain node, the type inference engine tries to select the least common superclass as type. If the least upper bound degenerates to `Object`, an error is thrown to show that no specialization is possible. The same happens when two types have no common superclass, like booleans and objects. If such type-errors arise, they have to be fixed by the programmer. These problems are often solved by introducing a new common superclass or moving a method higher up the hierarchy. In other cases they really are semantic errors and require restructuring. Section 4.3 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 limits the expressiveness of the input language. For this reason we call the restricted input language understood by the PyPy toolchain RPython instead of Python. RPython is described in more detail in Section 2.3.

The annotation step is followed by the conversion from a high-level flow-graph into a low-level one. Up to now there are two converters, one which specializes towards low-level back ends like C and one which specializes towards object-oriented back ends like CLI.

To the low-level flow-graph optional back end optimizations are applied. 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 back end. The preparation for code generation covers the following steps:

– Insertion of explicit exception handling.
– Adding memory management details. Different garbage-collection strategies are available[5]. 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.

Eventually the language-specific flow-graph is transformed into source files. These source files are then again processed by the back end, which can perform further domain-specific optimizations. For example generated C source files are compiled with GCC using the `-O3` flag.

2.3 RPython

The PyPy translation toolchain is designed to boost the performance of the Python interpreter written in Python. More than just that, it translates general VM prototypes to fast executable binaries. The translation from a dynamically-typed language like Python to a statically-typed language like C is not straightforward however. As mentioned before, in order to be able to preserve the semantics we are forced to limit the expressiveness of the input language. For this reason, when we talk about the language accepted by the translation toolchain, we do not refer to Python but rather to RPython or restricted Python. The language is defined implicitly by the translation toolchain[6]. The main differences between the full Python language and RPython are summarized in the following list:

– Variables need to be type consistent,
– Runtime reflection is not supported,
– All globals are assumed to be constants,
– Types of all variables in the code must be inferable.

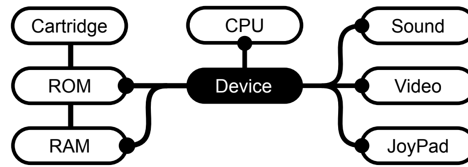
Although these restrictions seem to be substantial for a dynamic language such as Python, it is still possible to use high-level features like single inheritance, mixins and exception handling. More importantly, since RPython is a proper subset of Python, it is possible to test and debug the input programs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

3 Game Boy Technical Details

As a case study we implement PyGirl, an executable VM prototype of a Game Boy. We will then translate this prototype using the PyPy translation toolchain

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

presented in the previous section. In this section we list the technical details of the gaming hardware. The official documentation is available on Nintendo's website[7].



3.1 Hardware Pieces

The Game Boy system is composed of six essential pieces which are accessible through shared memory. External events are supported through an 8 bit maskable interrupt channel. There are two kind of 8 bit 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 Nintendo's gaming device.

- The 8 bit CPU is a slightly modified version of the Zilog 80 with a speed of 4.19 MHz. The CPU supports two power-saving mechanisms both working 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 ROM with the embedded game and possibly additional RAM and/or other devices. The size of the RAM depends on the type of cartridge. Some types of cartridges 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 8×8 or 8×16 pixels at the same time. The video chip has two tile-map memory regions, one for the background and one for the foreground. The actual background and foreground drawn on the screen are cropped versions of the data available in the tile maps. This allows the Game Boy to easily refresh the tile maps.

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

- A serial connection can be used to communicate with another device.
- 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 samples at the cost of its calculation, or it can be produced via a noise-channel or via two different wave-pattern generators.

4 PyGirl Implementation

In this section we highlight the most relevant implementation details of the PyGirl VM. We present a list of refactorings that we applied while going from a source implementation to our own model. We especially stress the details regarding the application of preprocessing-time meta-programming.

Instead of starting to implement from a formal specification for the Game Boy we adapt an existing stereotypical VM implementation to PyPy. This allows us to show the differences of both implementation styles more easily. We then compare the complexity of our resulting prototype to various other Game Boy VM implementations: Mario which we refactor to our own implementation, JavaBoy[8] and AEPgb[9]. Table 1 shows how our final prototype differs from the other Game Boy VM implementations in terms of McCabe cyclomatic code complexity (MCC) [8]. It assigns a number to a piece of code corresponding to the number of possible traces through the code.

Cartridge	JavaBoy	AEPgb	Mario	PyGirl `r54984`	`r63242`
KLoC	1.0	0.5	1.2	0.8	0.8
Number of methods	28	24	84	89	90
MCC Sum over all methods	220	103	268	217	211
Methods over MCC > 10	3	1	5	0	0
Max MCC	70	17	23	9	8
average	7.86	4.29	3.19	2.44	2.34

Table 1. McCabe cyclomatic complexity of the Cartridge related classes.

4.1 Source Implementation

In this section we present the important details of the Game Boy VM written in Java from which we started. The Game Boy VM Mario [5] is developed in a portable manner by abstracting out platform-specific details from certain components. Hence it provides variants of the emulator for the different versions of Java architectures like the Java Standard Edition and Applets for the web.

[8] `http://www.millstone.demon.co.uk/download/javaboy/`
[9] `http://sourceforge.net/projects/aepgb/`

The application is structured by providing one class for each physical piece of hardware. Platform-specific parts are factored out by providing a set of abstract driver interfaces handling input and output. These drivers are then implemented for each architecture separately, adapting to the platform-specific requirements. Even though this is a fairly abstract and portable design, this already is an indicator that without a toolchain VM implementations are bound to be cluttered with back end specific details.

While at first glance the code appears to be written in an object-oriented manner, many parts of Mario 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 does not provide many possibilities for abstraction. However, even the video chip is implemented in a non-abstract procedural way. This is so even though there are more conceptual components ready for abstract representation, such as *sprites*, *background* and *foreground*. PyGirl uses high-level abstractions for these components resulting in less complex code. Table 2 shows that our Video classes are less complex than the ones from Mario.

Video	JavaBoy	AEPgb	Mario	PyGirl `r54984`	`r63242`
KLoC	1.0	1.1	1.2	0.6	1.1
Number of methods	53	108	68	76	182
MCC Sum over all methods	212	288	223	201	293
Methods over MCC > 10	5	5	5	3	0
Max MCC	36	24	20	15	8
average	4.00	2.67	3.28	2.64	1.61

Table 2. McCabe cyclomatic complexity of the Video related classes.

4.2 From Java to Python

Now we show how we migrated the source VM from to Java to Python. In a first step we ported the code one-to-one, in order to easily track the upcoming refactoring progress. During the whole process we keep the overall structure of the existing system because it directly corresponds to the hardware.

The following sections cover different refactorings we apply and abstractions introduced to go from a low-level detailed implementation to a high-level prototype of the VM.

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 integers are used for all sorts of arithmetic operations. Instead of using integers whenever possible, the Java version focuses on reducing the memory footprint of the running emulator and focuses on the implementation details of the original hardware. Only at very few places in the code is the use of bytes justified by the resulting two's-complement interpretation of the numbers.

In our prototype type-casts are removed wherever possible to improve readability and maintainability. Firstly we consider the memory footprint of the device we emulate too small to justify optimization of memory usage. Even if we use four to eight times as much memory as the original device would have used, corresponding to an expansion from 8 to 32 or 64 Bit, this would mean that we end up with about 20Mb memory usage. Running PyGirl on a 64 Bit machine results in a total memory usage of 24Mb. This is a negligible amount for modern computers. Secondly and more importantly it is, hypothetically speaking, possible to plug an additional transformation into the toolchain converting all integers to bytes. By doing so the memory footprint of the final VM would again be equal to the one of the original device.

Metaprogramming As described in Section 2.2 we can use the full Python language at preprocessing-time for meta-programming. The most prominent candidate for refactoring is the CPU class. It is packed with duplicated and inlined code and has a typical opcode dispatch switch. Table 3 shows the MCC for the CPU class in Mario, the two other Game Boy VM implementations AEPgb and JavaBoy and two snapshots of PyGirl.

CPU	JavaBoy	AEPgb	Mario	PyGirl `r54984`	`r63242`
KLoC	2.7	2.9	4.2	1.2	1.0
Number of methods	22	111	372	173	180
MCC Sum over all methods	801	704	996	272	252
Methods over MCC > 10	5	1	2	0	0
Max MCC	415	536	513	10	9
average	36.41	6.34	2.28	1.57	1.4

Table 3. McCabe cyclomatic complexity of the CPU related classes.

PyGirl's CPU class has half of the number of methods that Mario has. The sum of the code complexity over all methods is drastically reduced. One reason for the high complexity sum of Mario's CPU class is its size. The original class is around 4000 lines long, featuring an unpleasant 1700 line switch delegating the incoming opcodes. On top of that there is a nested switch of 800 LoC handling the second-order opcodes. An excerpt of this dispatch switch is given in the following listing:

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

    ...

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

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

In both switches we identify patterns which can be used as basis for abstractions. In the following excerpt from the original Java code we 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 1. Java: Grouped opcode mappings

The Java code covers all these switch cases by manually specifying them and by encoding the logic in one function per opcode. Since all these operations are symmetrical in terms of semantics and use of cycles the code can be compacted by using meta-programming. Even while there are only few lines of code per operation there is quite some redundancy.

Instead of separate functions PyGirl uses a single `load` function for all register combinations. We reuse the function for multiple opcodes by applying the load function to each time two register objects. Every combination of the load function with two registers is related to a single opcode encoding its meaning.

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

To refactor Listing 1, we create such reusable functions for all the different types of operations. Then we replace the switch with a compact lookup in an opcode table generated from the abstract functionality descriptions.

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

Instead of hard-coding the mapping to the respective functions, we use meta-programming to compute the definition at translation time. The mapping of opcodes to functions in the example Listing 1 can be replaced with a more compact definition. We specify the connected opcodes in a set of entries, each consisting of a starting opcode, an offset, a function and a set of registers. At runtime a sequence of opcodes is mapped to such a function. A corresponding register out of the register set is passed as an argument to this function. An example of this compact opcode definition is given in the following listing:

```
REGS = [CPU.get_bc, CPU.get_de, CPU.get_hl, 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, registers)
    ...
]

OP_CODE_TABLE += create_op_codes(SET)
```

Listing 2. RPython: Compacted definitions of opcodes

In the first line we see that we do not directly specify the register. Instead we use getter methods of the CPU class returning these registers at runtime. The first line of the `SET` specifies that the opcode `0x01` is mapped to `fetch_double_register` passing in the register returned by `get_bc()`. The opcode is `0x11` using the result of `get_de()` as argument to the function. Since there are 4 registers in the `SET` the last opcode in this sequence is `0x31`.

Next we have to create the concrete methods from this definition. Helper function take the opcode definition set at preprocessing time and create corresponding closures. The `create_op_codes` method in the following listing creates such closures:

```
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
```

Since Python handles the scope of variables at the function rather than the block level we introduced a further helper method `register_lambda` for creating the closures. In the following excerpt you can see how this method creates a specific closure depending on the incoming `registerOrGetter` argument:

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

We create opcode table entries at preprocessing time not only for most of the register operations such as loading and storing, but also for nearly all other register operations. In total we apply meta-programming to generate about 450 out of all 512 opcodes.

As a note on performance, when translating this code to C, 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 substantial loss in performance. Even better, future versions of PyPy are expected to automatically optimize running bytecode interpreters dynamically towards the bytecode they evaluate, *i.e.* JIT compiling the bytecodes. PyPy can also inline small methods. Although there are no inlined methods in Mario, it would be the next logical step for manually optimizing the code. By letting PyPy handle the optimizations, we maintain a clean implementation.

4.3 Translation

Directly trying to translate our VM prototype, which is initially full Python code, to a low-level back end raises conflicts. This is due to the fact that the PyPy translation toolchain does not take full Python code as input, but rather a restricted subset of Python called RPython. The restrictions imposed by RPython only become apparent when you try to translate prototypes with PyPy. Most bugs come from the fact that while Python is fully polymorphic, RPython enforces the static types of all variables to be correct.

Every variable needs to be inferable to a specific type. The most generic type in the system, *i.e.* `Object`, is not allowed as type for any variable. All messages sent to instances of a class must be declared in that class, or in any of its superclasses.

The easiest translation bugs are straightforward syntactic bugs, like typos. Such bugs often go unnoticed in dynamic programming languages since they are not statically enforced and are only a problem at runtime. After fixing these, you typically encounter type conversion errors. Some of these problems are related to assignments of objects of different types to the same variable. One possible solution for static type inconsistencies is to introduce common abstract superclasses. This ensures that objects assigned to a single variable have this superclass as a common type.

Call Wrappers We handle operations on registers and other CPU functions elegantly by passing function closures around. This allows us to reuse methods for different actions. A very common example is the following `load` function:

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

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

The `load` function is called with different arguments. In the first example the function is used to copy the values from the flag-register into the register `a`. The second example loads the next instruction into register `a`. When we first translated this code fragment the toolchain was unable to transform the code into a typed counterpart. PyPy was unable to create a strict typed function due to the different origins of the passed function closures. We resolve this by creating call-wrappers with a common superclass. Introducing call-wrappers for the passed function closures adds some overhead, but it is 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)
```

`RegisterCallWrapper` takes a register and calls `get` or `set` on it. The `CPUFetchCaller` is an abstraction for the `fetch` method of the CPU that allows it to be used as an argument for the `load` function. These are two out of the five total call-wrappers we use to handle closure passing. The common superclass `CallWrapper` makes it possible for PyPy to infer a common type for the argument of every method. Note that return-types of methods are also required to be type-correct. For our wrappers this implies that all `get` methods are required to return a value of the same type. Thus all `get` methods return integers. To support the call-wrappers we replaced the closure calls. Instead of directly invoking the argument as in Listing 4.3 we call `get` or `set` on the call wrappers. Applied to the previously presented `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

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

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

While it is possible to do the same in Java, this imposes a huge runtime overhead. If we create the CallWrappers in Java, this form of meta-programming is not handled at preprocessing time. Instead the objects will be around at runtime, which implies a performance as well as a memory overhead.

5 Performance evaluation

In this section we show that using the PyPy toolchain to reduce complexity of WSVMs does not result in substantial performance loss. To do so we compare the performance of our Game Boy VM PyGirl to the performance of our source implementation, Mario. We run benchmarks on three different versions of the Game Boy VM. The original Java emulation Mario, the interpreted variant of PyGirl and finally the translated binary version of PyGirl. We benchmark the interpreted PyGirl by running it on top of CPython whereas the translated version is built from those sources.

Each test shows the average execution time over 100 runs using Java `1.6.0_10`, Cacao `0.97` and CPython `2.5.2` (additionally using Psyco `1.6-1`) on a 64 Bit Ubuntu `8.04.1` server machine with an Intel Xeon CPU QuadCore 2.00 GHz processor. We use revision number `r63242` of the PyPy project.

5.1 Benchmark Details

In this section we discuss the benchmark parameters. The binaries for the RPython benchmark are created using the default arguments for PyPy resulting in C code which is then transformed to an executable binary via GCC. PyPy uses the following GCC optimizations when creating the binary executable:

```
-c -O3 -pthread -fomit-frame-pointer
```

In order to show the difference and impact of a JIT and dynamic optimization for the interpreted PyGirl we use Psyco [9], a just-in-time specializer for Python. Since the current version of Psyco (`1.6-1`) only emits machine code for 32 bit intel-based systems, we benchmark our code using Psyco on a similar Ubuntu installation with the appropriate 32 bit processor. Since those results are less important than the comparison with Java, we simply scaled them to fit the 64 Bit machine. Those results should only display the possible performance gain by using a JIT. Still, it highlights that running the emulator on top of Psyco equally results in unacceptable performance.

For those who are missing the `-server` switch in the Java 1.6 benchmark, we can say that this option did not result in any significant performance gain for the tests. The speedup is less than a percent and about the order of the standard deviation, thus the results are merely distinguishable from the standard configuration.

5.2 Runtime Optimization Comparison

In this section we compare the performance of Mario and PyGirl. In order to test performance we let both systems run a ROM which exercises the video output[10] but produced no sound[11]. The ROM simply prints *hello world!* on the background while a *smiley* moves over the screen as a moving sprite, a simple but complex enough benchmark. In the Table 4 you can see the actual CPU time used by the emulators in relation to the actual game time. By game time we mean the actual time that a user spends playing a game at normal speed. By CPU time we mean the time spent by the processor of the host actually running the emulator. Notice that as game time increases, the difference in CPU time between Mario and the translated PyGirl emulator shrinks. Eventually Mario gets faster than PyGirl due to the runtime optimizations of the JVM. Figure 2 shows the performance for both emulators running on different platforms. It is clearly visible that the translated PyGirl runs at linear speed, whereas the Mario behaves differently. The JVM's JIT is only warmed up after about 30 to 40 seconds of actual game time. The Cacao VM performs somewhat differently from the standard JVM implementations. Cacao's JIT already runs Mario optimized after very little game time, but ends up being even a bit

[10] The pixels are drawn in internal buffers but are not actually written to a screen.

[11] In both systems we fully disabled the sound drivers so that they cannot have any impact on performance.

Game Time	RPython CPU time	Java 1.6 CPU time	Ratio
1	0.03	0.19	0.15
4	0.12	0.25	0.46
10	0.29	0.37	0.78
15	0.43	0.46	0.93
30	0.86	0.74	1.17
60	1.73	1.28	1.36
300	8.60	5.47	1.57
600	17.23	10.73	1.61
1800	51.81	31.65	1.64
3600	103.62	63.08	1.64

Table 4. Influence of the JIT on the benchmark results. Average execution in seconds over 100 runs per test.

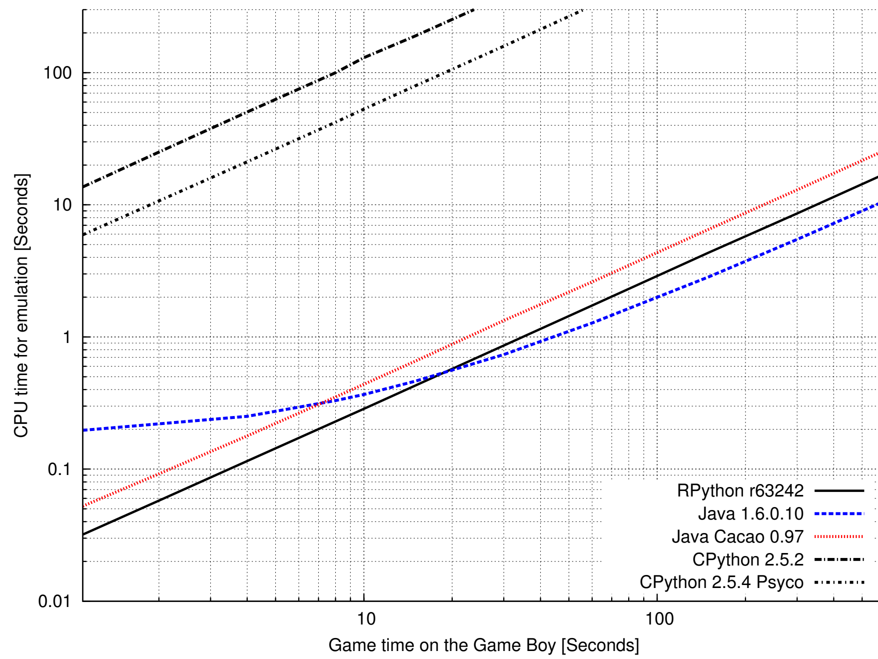


Fig. 2. This graph shows the CPU time of the host needed to emulate an example ROM, in relation to the game time on the Game Boy. It compares the performance of PyGirl and Mario emulating all device parts excluding sound. The results are discussed in Section 5.2

slower than PyGirl. While Mario on the standard JVM has the advantage of the availability of a JIT compiler and dynamic optimizations, it is only about 60% faster than PyGirl.

Simple benchmarks running on simple language interpreters which are compiled with PyPy to its CLI-back end[12] have shown that a JIT compiler can also be used here to get significant performance gains. We strongly believe that future versions of PyPy will directly improve the performance of the PyGirl VM, thanks to a generated JIT compiler, by the same order of magnitude. Future versions of PyPy featuring dynamic optimizations for the C-back end might even help us outplay the performance of Mario running on the standard JVM.

6 Future Work

In this section we discuss the future work needed for PyGirl and related tasks for PyPy.

6.1 Future work for the Game Boy VM PyGirl

While most hardware parts already have a fully functioning software counterpart, this is not the case yet for the sound unit. This is mostly the case since it is the least important piece of hardware for the Game Boy emulator to be immediately usable. For it to work it still needs to be fully ported and refactored.

6.2 Future work for PyPy

PyGirl's current state of implementation only allows the code to be translated using the C-back end. The current implementation of the I/O drivers are based on libSDL[13], thus it is not yet compatible with other back ends. The fact most VMs only need very basic graphics support, like simple bitblitting, makes enabling graphics support for the other back ends a straightforward task.

Translating PyGirl with the JVM as target makes it possible to directly compare the performance of the original Java implementation with our approach. Since the Java Virtual Machine is widely available and compatible with many different platforms, this would eventually even allow us to run PyGirl on mobile devices [2].

7 Conclusion

In this paper we have shown that the use of high-level prototypes for the definition of whole-system virtual machines and the application of meta-programming

[12] http://morepypy.blogspot.com/2008/11/porting-jit-to-cli-part-1.html
[13] http://www.libsdl.org/

reduces code complexity significantly without substantial loss of performance. By using the translation toolchain PyPy we keep our high-level WSVM model free from low-level implementation details, such as garbage collection and exception handling. When we build a final WSVM from out prototype, those details are reintroduced by the translation toolchain.

We have supported our claims by comparing the performance and complexity of two VM implementations for the Game Boy, PyGirl and Mario. We found that for specific classes the average McCabe cyclomatic complexity is reduced to less than the half compared to other Game Boy VMs. In other cases, like the CPU class, we even reduced the maximum MCC from over 500 down to 9. In order to strengthen our case about the reduced complexity, we compared the complexity of our prototype with yet another two Game Boy emulators, exhibiting the same complexity issues as Mario. In Section 5 we have shown that using PyPy does not result in a significant performance loss. PyGirl only runs about 40% slower than the Game Boy VM Mario on the standard JVM.

Acknowledgments. We would like to thank Marcus Denker, Adrian Kuhn, Jorge Ressia, Oscar Nierstrasz and the anonymous reviewers for kindly reviewing this paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010). The authors acknowledge the support of CHOOSE, the Swiss Group for Object-Oriented Systems and Environments.

References

1. Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'99)*, pages 314–324, New York, NY, USA, 1999. ACM.
2. Ludovic Aubry, David Douard, and Alexandre Fayolle. Case study on using PyPy for embedded devices. Technical report, PyPy Consortium, 2007.
3. Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. Back to the future in one week — implementing a Smalltalk VM in PyPy. In *Workshop on Self-sustaining Systems (S3) 2008*, page TBA. TBA, 2008. To appear.
4. Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Reading, Mass., May 1983.
5. Carlos Hasan. Mario, 2007.
6. 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.
7. The Jikes research virtual machine. http://jikesrvm.sourceforge.net/.

8. T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
9. Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for Python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM.
10. 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.
11. Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM Press.
12. James E. Smith and Ravi Nair. *Virtual Machines*. Morgan Kaufmann, 2005.
13. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.
14. David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.