VM Wrapping

Fake it till you make it

Johannes Henning
Hasso-Plattner-Institute, University of Potsdam, Germany
johannes.henning@hpi.de

Tim Felgentreff
Hasso-Plattner-Institute, University of Potsdam, Germany
tim.felgentreff@hpi.de

Robert Hirschfeld
Hasso-Plattner-Institute, University of Potsdam, Germany
robert.hirschfeld@hpi.de

Abstract

Building or extending *Virtual Machine*s (VMs) to investigate new language features or optimization techniques is challenging in several ways. The overhead for developing a new research VM for an existing practical language is immense, and meaningful evaluation often requires implementing much more than just the parts that are interesting for the research question.

In this paper, we propose a different approach for implementing VMs based on wrapping an existing, feature complete VM. Our technique aims for lower implementation overhead by reducing the number of features that have to be implemented to produce a working prototype and thus producing results quicker. While already proving useful for research, our approach also suggests a way to extend legacy virtual machines with new features and optimizations.

Keywords Virtual Machines, RPython

ACM Reference format:
Johannes Henning, Tim Felgentreff, and Robert Hirschfeld. 2017. VM Wrapping. In *Proceedings of ICOOOLPS'17, Barcelona , Spain, June 19, 2017,* 4 pages.
https://doi.org/10.1145/3098572.3098576

1 Introduction

Many modern programming languages are executed inside an interpreter or VM. For popular programming languages such as Python or Java a multitude of VMs is available, offering different benefits or targeting different usage scenarios. VM-design is an integral part of programming language research. Optimization techniques implemented inside the VM such as *Just-In-Time* (JIT)-compilation continue to be active research topics. VM implementation frameworks such as RPython [1] offer optimization techniques which can be implemented language independently. The RPython JIT compiler has been applied to several languages, yielding significant performance benefits.

However, conducting such experiments is very time-intensive as altering an existing VM or implementing a new one requires significant implementation effort, as well as knowledge about minute language-internals. In particular, researchers might have to invest a lot of effort implementing parts of the language that are not directly relevant for answering the research question, but necessary to support the targeted language. If this process of VM development could be streamlined further, it would be beneficial to VM development, particularly in research. In this paper, we propose an implementation approach for quickly generating meaningful apples-to-apples measurements of the effectiveness of new optimization techniques or language features. This proposal centers on the idea of wrapping an existing VM and only overriding the parts relevant for research to quickly gain insights into the research question, while maintaining complete compatibility with the existing language implementation. This approach also seems to be applicable for language implementations in general, in particular maintaining and enhancing legacy languages.

In particular, we aim to lower the implementation effort of:

1. Implementing new bytecodes or language features
2. Modifying internal VM objects and storage strategies
3. Implementing and testing new optimization techniques
4. Implementing a new VM while continuously providing a compatible and executable VM throughout development

We will outline background and motivation in Section 2 before presenting our proposed implementation and possible usage scenarios in Sections 3 and 4, after which we position ourselves to related work in Section 5 and outline future work in Section 6, before concluding in Section 7.

2 Background and Motivation

Our approach builds upon the RPython framework [1], and optimization techniques such as just-in-time compilation and meta-tracing JIT compilers [2]. We will introduce the SQPyte prototype which builds upon these technologies and expand it as a motivating example for our proposed solution.

ICOOOLPS'17, June 19, 2017, Barcelona , Spain

ACM ISBN 978-1-4503-5088-4/17/06…$15.00
https://doi.org/10.1145/3098572.3098576

2.1 SQPyte

The SQPyte [3] prototype aimed at speeding up database access in PyPy by including SQLite in the runtime environment. SQLite is an in-process library implementing a SQL database engine with a c-based interpreter. By virtue of having no dependencies, being serverless, and requiring little configuration, it is one of the most widely deployed databases[1]. By overriding selected bytecodes in the *Main-Interpreter Loop* (MIL) of the SQL engine it allowed the JIT compiler to optimize application and database operations at the same time. They built their prototype in 8 person-months and used an implementation technique similar to the one presented here. Overall, the prototype was able to produce insights into the optimization potential of an in-process database while maintaining complete compatibility with unmodified SQLite invocations and only modifying relevant parts of the code base. The SQPyte prototype did not modify any data storage and did not need to be aware of specific database internals, while still allowing for JIT optimized SQL statement execution.

2.1.1 SQPyte as a Motivating Example

As demonstrated by the SQPyte prototype, an approach like VM wrapping can allow for experiments on established software projects, without incurring the massive implementation overhead of rebuilding the existing implementation in the research framework. More concretely: The established approach for this type of experiment would have required a reimplementation of a SQLite equivalent database inside the RPython framework or Python itself. Unless the newly implemented database would have been able to support all SQLite features, the researchers still would have needed to argue the applicability of the results to SQLite. SQLite has been developed for over a decade[2] and rebuilding it or exchanging significant parts of the implementation would require significant implementation effort. Without the approach followed, it would not have been possible to conduct the experiments in the relatively short period of time it took to build the prototype.

We believe the SQPyte prototype as a case study motivates VM wrapping as a general approach for implementing research prototypes for programming language experiments.

3 Proposed Implementation Strategy

VM wrapping relies on an existing implementation as much as possible in order to reduce implementation overhead and thereby streamlining the development of new VMs. We propose overriding and extending an existing implementation

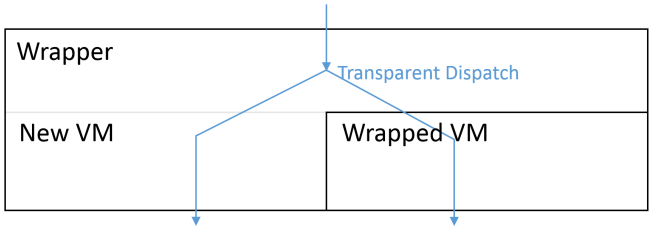

Figure 1. Conceptual architecture of VM wrapping with transparent dispatch between new and wrapped VM

and delegating everything that is not required by or interesting for our implementation goals. This is done by *wrapping* the existing VM in our new VM and delegating where appropriate, see Figure 1. Thereby, we support the entire language, even if we have not implemented all of it ourselves. This is particularly helpful, because we can rely on existing benchmark suites as well as an abundance of existing example applications.

There are several ways this delegation could be implemented, varying by implementation strategy and carrying different advantages and disadvantages, which we will discuss in the following Sections.

3.1 Granularity of Delegation

The first important decision depends on the parts of the implementation that the researchers are interested in. For example, are we interested in implementing a new bytecode, changing the implementation of an existing one, using a different layout for internal VM objects, or testing optimization techniques like unrolling?

This decision is important for identifying the parts of the existing implementation that we need to extend or replace. A new bytecode could be added relatively easily to the MIL of the existing VM, while changing storage strategies for internal VM objects would require modifications spanning several bytecodes and rewriting the underlying data structures. The following Sections present different delegation techniques appropriate for different granularities of delegation.

3.1.1 Override Existing Bytecodes

Overriding existing bytecodes is the most straightforward approach other than extending the existing VM altogether. The SQPyte prototype discussed in Section 2.1 used this approach. The idea is to compile the existing VM into our framework and run it as usual, only overriding bytecodes that we are interested in. The ideal place for dispatching between our new functionality and the wrapped VM is the MIL, which is responsible for dispatching each bytecode to its implementation. We retain all existing features of the VM, while the limitations of this approach lie in the existing implementation. As we can only redefine the execution of selected bytecodes, we cannot change the underlying data structures or make other major architectural changes. We

[1] According to https://www.sqlite.org/mostdeployed.html (accessed on 2017-06-01)

[2] Development started 2000-05-09 https://www.sqlite.org/about.html (accessed on 2017-06-01)

also might require somewhat extensive knowledge of the existing implementation.

3.1.2 Transfer State, Dispatch Per Bytecode

By mapping our internal VM objects onto the wrapped VM's internal objects it would be possible to switch bytecode execution between both VMs dynamically and transfer the internal VM state before each switch. This would allow for complete freedom in the new VM architecture, but would require significant knowledge about the wrapped VM in addition to run-time overhead for each switch.

An alternative way to ensure the same state in both VMs would be to execute both in lock-step, e.g. synchronizing after each bytecode. This would not be useful for performance benchmarks, but could help debugging and testing computational correctness.

3.1.3 Dispatch Per Function

Wrapped VM implementations could also be called on a per-function basis. Depending on the level of function 'purity' the passing of relevant internal objects could be relatively simple. While this would require the VM to implement all basic language features, it would allow for complete implementation freedom, while allowing the developer to delegate heavily to the wrapped VM similarly to invoking a library.

3.1.4 Pre-Scan, Dispatch Per Program

A different approach would be to keep both VMs separate from each other and select the appropriate VM through static analysis. Say our research VM only implements a subset of the specified language, we could determine whether the program uses language features outside of our implementation and transparently execute it in the wrapped VM. This approach would only make sense if the subset of the language we are implementing is large enough to write meaningful programs in. The advantages would be complete architectural freedom and requiring very little knowledge about the wrapped VM.

3.2 Combining Abstraction Levels

The implementation approaches presented can also work in combination with each other. For example, we might select the appropriate VM by analyzing byte- or program-code, but discover at run-time that the program uses an unimplemented language feature through a function like *eval*. At this point we could fall back on delegating to the wrapped VM via our *per-function* approach.

4 Usage Scenarios

We believe that the presented approach could be useful for multiple use-cases, presented in the following Sections.

4.1 Research Prototypes

As discussed in Section 2.1, the overhead of implementing equivalent technologies to create meaningful and applicable research results is a general problem of language implementation research. While RPython already makes it possible to experiment with JIT technologies with minimal overhead, it still requires the complete language to be implemented for the VM to be of practical use. While there have been some successful implementations of language VMs in RPython, i.e. PyPy and RSqueak[3], there have been more research VMs that are unlikely to ever support the entire language they were targeted to implement, simply because it is not connected closely enough to the research question the partial VM was built for.
We believe that VM wrapping can be an applicable solution for many similar problems in research. It is a way to get answers faster, much in the same way that RPython is. While the JIT optimizations RPython provides are generally not as efficient as what is available in specifically tailored VMs like Java hotspot or LuaJIT, the cost-benefit of RPython is far better. Similarly, the result of VM wrapping will not be an optimal VM for the targeted language, but a low-cost way of answering research questions about the language, while generating more meaningful results than alternative low-cost approaches.

4.2 Legacy Language Maintenance

There is another use-case for transparent dispatch between different VM implementations. We have observed problems for vendors of proprietary languages when deprecating language features. Since the vendor is not aware of all existing customer applications and features used within them, the vendor is often hesitant to deprecate at all, fearing suboptimal usability or lack of backwards compatibility. We believe that an approach like pre-scan (section 3.1.4) could help to update such languages while maintaining backwards compatibility for overhauled language features. Such an approach could also prove useful for transition periods and continuous integration development, as parts of the language could be redesigned one at a time, while continuously providing a compatible and complete VM.

4.3 General Language VM Development

For the design of a new production VM for non-research purposes, VM wrapping could also be useful as a way of getting early feedback during development by allowing to execute programs even if important parts of the new VM have not been implemented yet.

5 Related Work

As the idea we presented here is in essence about easing language implementation, there are numerous publications

[3] https://github.com/HPI-SWA-Lab/RSqueak (accessed on 2017-06-01)

directly related to this goal. However, we are not aware of previous work describing the particular implementation strategies presented here. We will present three projects which aimed to solve similar problems, namely a technique used in MacLisp, the aforementioned SQPyte project, and the Truffle framework.

5.1 Unimplemented User Operations

Unimplemented User Operations (UUO) were used in MacLisp to dispatch op-codes to different implementations [4]. These unimplemented op-codes would trap the execution and allow calling user-defined functions. In practice this was enabled transparent switches between compiled and interpreted versions of the VM and thus was not intended to dispatch between different VMs. However, this approach demonstrates the possibility of replacing bytecode implementations transparently at a lower level of abstraction.

5.2 SQPyte

While our approach is similar to the work done for the SQPyte prototype (Section 2.1), it was focused simply on getting results in the fastest and easiest way available and was not concerned with the general applicability of the used implementation approach, but presents a valuable case study for our proposed implementation approach. In Section, 3 we presented several considerations and implementation addendums that were outside the scope of the SQPyte prototype and thus not discussed previously.

5.3 Truffle

The Truffle framework [5] optimizes *Abstract Syntax Tree* (AST) interpreters building on the Graal JIT compiler. While the project goals are similar to the RPython project, the implementation is different, as it is based on the Java HotSpot VM and works with AST interpreters. While we focus on bytecode interpreters in this work, VM wrapping could work just as well with selectively overriding node interpretation in an AST interpreter. The JRuby+Truffle project [4] went a similar route when optimizing JRuby by overriding and thereby optimizing parts of the existing implementation step-by-step. As with the SQPyte project this approach was the simplest way to proceed and the project was not concerned with the general applicability of the used implementation approach.

6 Future Work & Open Questions

We are confident that VM wrapping can be applied to most bytecode interpreter by selectively overriding bytecodes as shown by the SQPyte project. However, we aim to show further uses of VM wrapping in the future, in particular concerning the other presented usage scenarios.
Our future experiments will continue to focus on the RPython tool-chain, which not only allows us to experiment with all the languages already implemented in it, but also enables us to wrap any C-based language implementation. In particular, we would be interested in the feasibility of extending an existing incomplete RPython-based VM (e.g. Topaz) by wrapping a C-based VM (e.g. MRI Ruby) and dispatch using the *per function* approach. The goal of such an experiment would be to show whether the effort invested would be less than porting the missing functionality.

[4] http://chrisseaton.com/rubytruffle/announcement/ (accessed on 2017-06-01)

6.1 Open Questions

While SQLite contains a classic C interpreter it is mainly a database and it is unclear how well the implementation approach of SQPyte fits to a programming language environment.
Also, the underlying assumption of VM wrapping is, that with any language implementation there are interfaces in the implementation that might ease reuse. It is not yet clear how this approach might work with parts of the implementation that are orthogonal to such interfaces, e.g. garbage collection.

7 Conclusion

We have presented an alternative approach to research VM development, which allows for streamlining parts of the process, by focusing on the research-relevant parts of the VM implementation and leaving the rest to an existing implementation while maintaining compatibility with existing benchmark suites and example applications. While VM wrapping seems feasible in theory, we have yet to test our proposed implementation approaches in practice, which is the logical next step and future work.

References

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D Matsakis. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the Dynamic Languages Symposium (DLS) 2007*, pages 53–64. ACM, 2007.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS) 2009*, pages 18–25. ACM, 2009.

[3] Carl Friedrich Bolz, Darya Kurilova, and Laurence Tratt. Making an Embedded DBMS JIT-friendly. *CoRR*, abs/1512.03207, 2015.

[4] Richard P Gabriel. *Performance and Evaluation of LISP Systems*, volume 263. MIT press Cambridge, Mass., 1985.

[5] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 133–144. ACM, 2014.