Virtual Machines for Dynamic Languages
Ausgewählte Kapitel der Systemsoftwaretechnik, WS2021

February 9, 2021

Mark Deutel
Friedrich-Alexander-University Erlangen-Nürnberg (FAU)
Motivation
Dynamic languages are becoming more and more important

- Python: Machine learning, simulations and statistics
- JavaScript: De-factor standard for modern dynamic website programming
Motivation

VMs for dynamic languages are still mostly written by hand in a low level system language like C - this has some disadvantages

- complex, monolithic software constructs
- require large efforts to create and maintain
- not easily portable to other target platforms
- some language properties can only be inferred at runtime (instead of compiletime)
Is it possible to ...

...reduce the workload required to write a dynamic language VM?

...increase the performance of dynamic language program execution despite having to evaluate properties at runtime?
1. Motivation

2. Workload Efficient VM Implementation
   - Hierarchical Layering of Virtual Machines
   - Applying Metaprogramming Techniques

3. Performance Efficient VM Implementation
   - Interpret Dynamic Language Code
   - Execute Interpreted Code

4. Conclusion
Workload Efficient VM Implementation
Layer VMs on top of each other

- introduces modularization and layer abstraction
- general purpose VMs can be used as hosting VMs
- the dynamic language VM runs inside the host VM as a guest

- Written in
  - Dynamic guest language
  - Managed host language
  - Managed host language or unmanaged language
  - Unmanaged language (typically C or C++)

- Guest Language Application
- Guest VM (dynamic language interpreter)
- Host VM (general purpose VM)
- Operating System or Hypervisor
Hierarchical Layering of Virtual Machines

Advantages

+ abstraction of target architecture specific details

+ guest VM can be written in the managed language of the host VM

+ advanced services of the host VM can be used by the guest (memory management, JIT, ...)

Disadvantages

- additional overhead due to layered approach

- performance is usually significantly worse compared to manually written VMs [4]

- host code generated by the dynamic guest language may perform worse compared to host code generated by the host language’s own compiler/interpreter [7, 9]
Create a low level VM from an interpreter description using a translation toolchain

- interpreter can be written with a high level (even dynamic) language
- translation toolchain transforms the description into a VM using metaprogramming techniques
Applying Metaprogramming Techniques

Advantages

+ approach is not dependent on any host VM
+ no actual bytecode has to be generated by the interpreter; all the heavy lifting is done by the translation toolchain
+ the translation toolchain allows for more target platform specific optimizations

Disadvantages

- translation toolchain has to be written at least once (for every target platform)
- resulting VMs usually perform worse than manually written ones [8]
Performance Efficient VM
Implementation
Dynamic Language Program Execution

Two main tasks
- **interpret** dynamic language code
- **execute** interpreted code on guest hardware

Some features of dynamic languages can only be inferred at runtime
- dynamic data types
- dynamic call sites
Abstract Syntax Trees

- each node represents an operation
- the operands of the operation are the node’s children
  - every inner node is an operation and an operand at the same time
  - leave nodes are only operands (e.g. constants, variables, ...)
Provide several type specialized "versions" of a node

- interpreter rewrites nodes based on inferred operand types
  - type checking of operands can be omitted
  - type specific optimizations can be used
- nodes may have to change their specialization in case of type instability
Execute Interpreted Code

Find traces through loops during JIT execution and optimize them

- assumption: programs spend most of their time in loops
- blocks belonging to a trace are linked together and then optimized
- the trace becomes invalid in case any execution "side exits" the trace

```java
public static void main(String[] args) {
    int i, k = 0;
    for (i = 0; i < 1000; ++i)
        ++k;
    System.out.println(k);
}
```

```
iconst_0
istore_2
iconst_0
istore_1
```

```
A: iload_1
    sipush 1000
    if_icmpge B
    iinc 2,1
    iinc 1,1
goto A
B: getstatic System.out
    iload_2
    invokevirtual println(int)
    return
```
Execute Interpreted Code

Trace Based JIT Compilers [2, 1]

Example:

- For loop with two nested conditional blocks
- Up to three different traces through the loop

Two worse case scenarios:

- The JIT traces nothing because the threshold is never reached
- The JIT constantly creates traces and throws them away immediately
A more elaborate way of recording traces

- Instead of discarding a trace in case of a side exit, the divergent execution is recorded.
- The new record is then added to the trace tree as a new branch.
Trace trees and dynamic languages

- Trace trees can be utilized for type divergence
- Allow **several trees** with different data types for a loop
- Allow **side exits** in case of a changing data type
- **Trace stitching**: Link together several trace trees
Conclusion
Writing good VMs for dynamic languages is a difficult to solve problem

- techniques to reduce the workload help but usually at the cost of performance
- optimizations can be applied to intermediate representations and to the JIT compiler but due to their speculative nature do not always work

There is no "goto technique"

- it very often depends on the required properties of a language’s runtime environment which techniques are useful
Just-In-Time (JIT) Compilers

- instead of interpreting code several times (e.g. loops, methods) just **interpret it once** and **cache the emitted code**
- since managing and finding single compiled instructions is expensive whole **blocks of instructions** are compiled together

+ interpreting and compiling code has to be done only once  
+ certain properties can be evaluated lazily  
+ instruction pointer only has to be updated at the end of a block

- compiled code may have to be invalidated  
- code cache has a limited size  
- finding compiled code must be a quick operation
V. Bala, E. Duesterwald, and S. Banerjia.
Transparent dynamic optimization: the design and implementation of dynamo.

V. Bala, E. Duesterwald, and S. Banerjia.
Dynamo: a transparent dynamic optimization system.
C. F. Bolz and A. Rigo.  
How to not write Virtual Machines for Dynamic Languages.  

S. Gaikwad, A. Nisbet, and M. Luján.  
Performance analysis for languages hosted on the truffle framework.  

Trace-based just-in-time type specialization for dynamic languages.


A. Gal, C. W. Probst, and M. Franz.

HotpathVM: an effective JIT compiler for resource-constrained devices.


T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. **Self-optimizing AST interpreters.**

A. Yermolovich, C. Wimmer, and M. Franz. **Optimization of dynamic languages using hierarchical layering of virtual machines.**