Virtual Machines for Dynamic Languages

Ausgewählte Kapitel der Systemsoftwaretechnik, WS2021

February 9, 2021

Mark Deutel

Friedrich-Alexander-University Erlangen-Nürnberg (FAU)

Lehrstuhl für Verteilte Systeme und Betriebssysteme

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Motivation

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)

Motivation

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?

Table of Contents

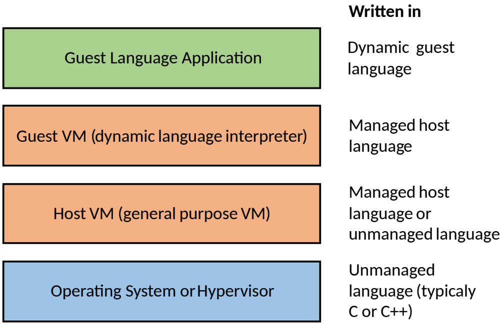
Workload Efficient VM Implementation

Hierarchical Layering of Virtual Machines

Compare [12, 10]

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



Hierarchical Layering of Virtual Machines

Compare [12, 10]

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

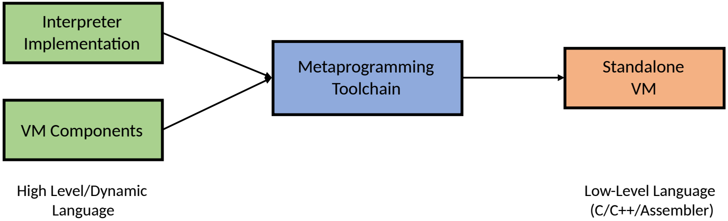
- additional overhead due to layered approach
- performance is usually significantly worse compared to manually written VMs [4]
- host code generated by the dynamic guest language may perform worse compared to host code generated by the host language's own compiler/interpreter [7, 9]

Applying Metaprogramming Techniques

Compare [3]

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

Compare [3]

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

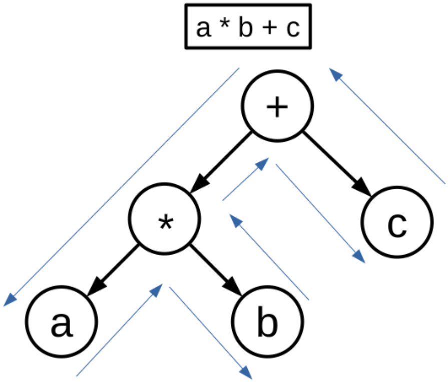
- **interpret** dynamic language code
- **execute** interpreted code on guest hardware

Some features of dynamic languages can only be inferred at runtime

- dynamic data types
- dynamic call sites

Abstract Syntax Trees

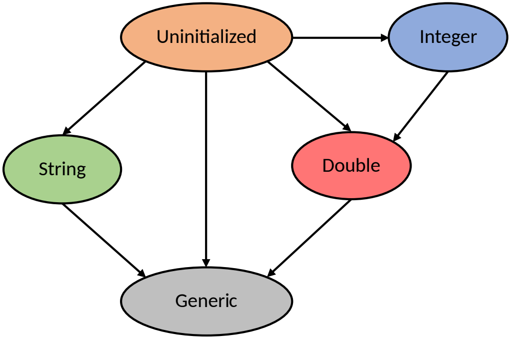
- each node represents an operation
- the operands of the operation are the node’s children
  - every inner node is an operation and an operand at the same time
  - leave nodes are only operands (e.g. constants, variables, ...)



Provide several type specialized ”versions” of a node

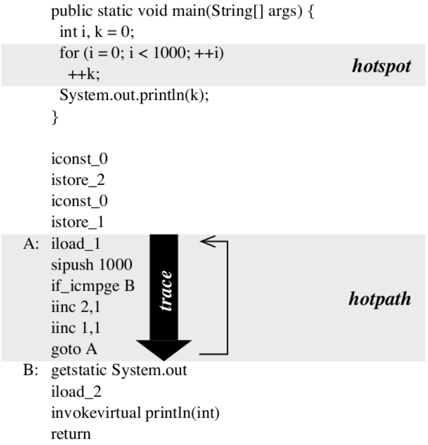
- interpreter rewrites nodes based on inferred operand types
  - type checking of operands can be omitted
  - type specific optimizations can be used
- nodes may have to change their specialization in case of type instability

Transition Model: Add-Node



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



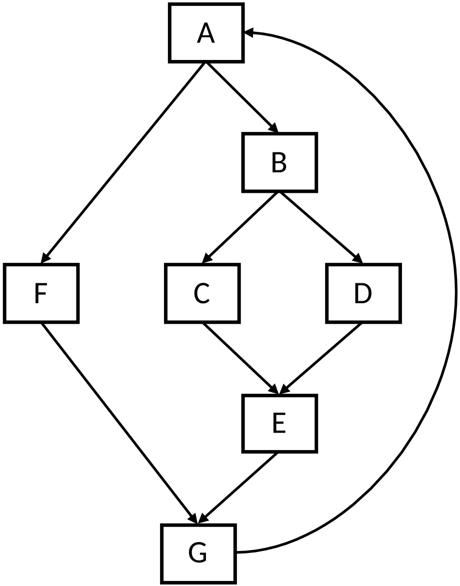
Execute Interpreted Code

Example:

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

Two worse case scenarios:

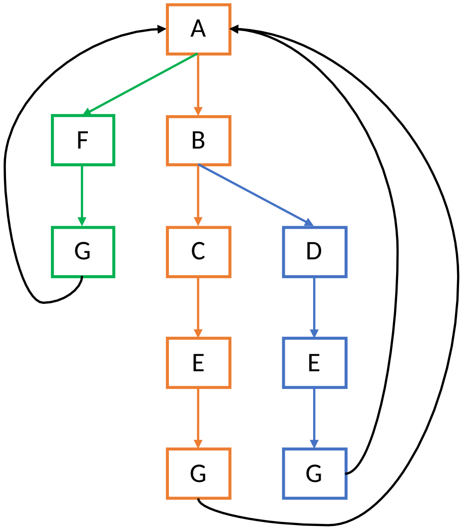
- The **JIT traces nothing** because the threshold is never reached
- The **JIT constantly creates** traces and throws them away immediately



Execute Interpreted Code

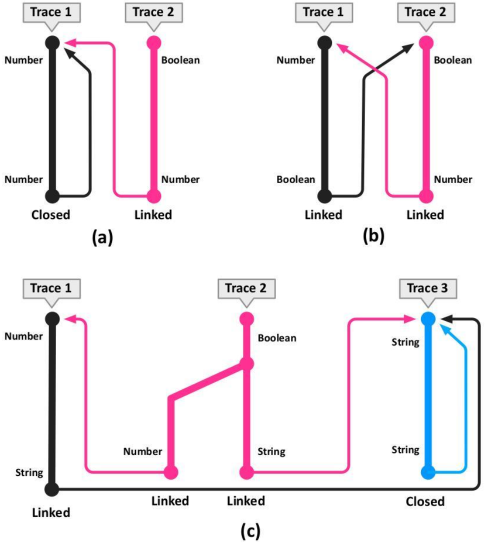
A more elaborate way of recording traces

- 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

Conclusion

Writing good VMs for dynamic languages is a difficult to solve problem

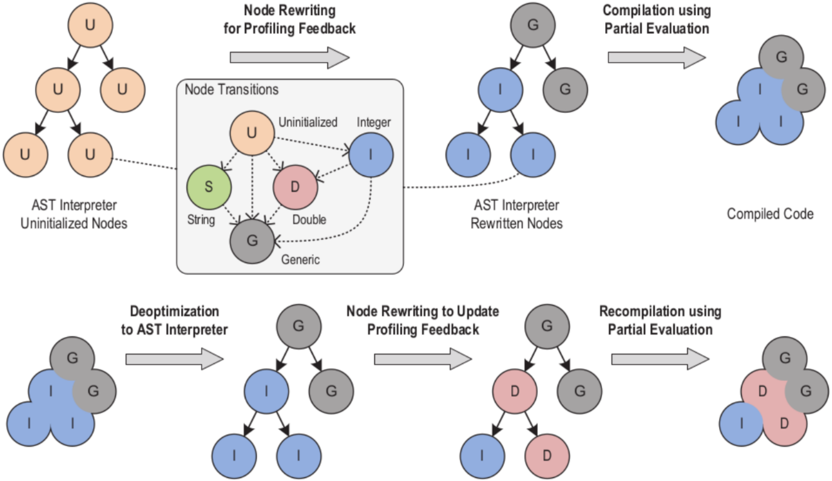
- techniques to reduce the workload help but usually at the cost of performance
- optimizations can be applied to intermediate representations and to the JIT compiler but due to their speculative nature do not always work

There is no ”goto technique”

- it very often depends on the required properties of a language’s runtime environment which techniques are useful

Type Specialization

Example from [11]



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

References (1)

V. Bala, E. Duesterwald, and S. Banerjia.
Transparent dynamic optimization: the design and implementation of dynamo.
Technical report, Hewlett Packard, 1999.

V. Bala, E. Duesterwald, and S. Banerjia.
Dynamo: a transparent dynamic optimization system.
In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, May 2000. Association for Computing Machinery.

References (2)

C. F. Bolz and A. Rigo.
How to not write Virtual Machines for Dynamic Languages.
(3rd Workshop on Dynamic Languages and Applications):11, 2007.

S. Gaikwad, A. Nisbet, and M. Luján.
Performance analysis for languages hosted on the truffle framework.
In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, pages 1–12, New York, NY, USA, Sept. 2018. Association for Computing Machinery.

References (3)

- A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz.
  Trace-based just-in-time type specialization for dynamic languages.
  ACM SIGPLAN Notices, 44(6):465–478, June 2009.
- A. Gal, C. W. Probst, and M. Franz.
  HotpathVM: an effective JIT compiler for resource-constrained devices.
  In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, June 2006. Association for Computing Machinery.

References (4)

- W. H. Li, D. R. White, and J. Singer.
  JVM-hosted languages: they talk the talk, but do they walk the walk?
  In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 101–112, New York, NY, USA, Sept. 2013. Association for Computing Machinery.
- A. Rigo and S. Pedroni.
  PyPy's approach to virtual machine construction.
  In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 944–953, New York, NY, USA, Oct. 2006. Association for Computing Machinery.

References (5)

- A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder.
  Characteristics of dynamic JVM languages.
  In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, VMIL '13, pages 11–20, New York, NY, USA, Oct. 2013. Association for Computing Machinery.
- T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko.
  One VM to rule them all.
  In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, Onward! 2013, pages 187–204, New York, NY, USA, Oct. 2013. Association for Computing Machinery.

References (6)

T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer.
Self-optimizing AST interpreters.
In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 73–82, New York, NY, USA, Oct. 2012. Association for Computing Machinery.

A. Yermolovich, C. Wimmer, and M. Franz.
Optimization of dynamic languages using hierarchical layering of virtual machines.
In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 79–88, New York, NY, USA, Oct. 2009. Association for Computing Machinery.