Virtual Machines for Dynamic Languages

Ausgewählte Kapitel der Systemsoftwaretechnik, WS2021

Mark Deutel
Friedrich-Alexander-University Erlangen-Nürnberg (FAU)
Erlangen, Germany
mark.deutel@fau.de

ABSTRACT

A common approach to execute dynamic languages is inside a virtual machine (VM), usually implemented in a low level system language like C. Such VMs have to support the often complex syntax and semantics of the dynamic languages they are hosting. Additionally, some properties of dynamic language code can only be inferred at runtime making the optimization of code execution another challenging task. Therefore, it is hardly possible to provide general purpose implementations that support a large range of different dynamic languages. Furthermore, many commonly used dynamic languages such as Python or Javascript are still evolving which means that their VMs have to adapt constantly. This implies that it is necessary to write and maintain a sufficiently elaborated code base for every single dynamic language. As a result, there is a significant development overhead when creating efficient, well optimized, and feature complete dynamic language VMs which often stands in contrast to time and resource limitations. Tackling this problem, research has focused on techniques to lower the development effort required to create VMs while increasing their maintainability. Simultaneously, emphasis was placed on optimization strategies that enable efficient code execution despite dynamically inferred runtime properties. The goal of this work is to give an overview over techniques for writing dynamic language VMs proposed during the last 20 years and to evaluate them.

1 INTRODUCTION

Dynamic programming languages have become extremely popular among software developers over the last years. Looking at the TIOBE index which ranks programming languages by popularity based on search engine queries, one can find several dynamic languages which have entered the top ten of the ranking during the last years [21]. This includes JavaScript at currently sixth and Python at third place only beaten by Java and C. Both of these two dynamic languages have gained massive popularity among computer scientists. Python has become a staple in the data science and artificial intelligence community driving two of their most popular machine learning frameworks PyTorch and Tensorflow while JavaScript has become the de-facto standard for dynamic website programming.

Therefore, reliable and fast runtime environments to execute dynamic languages are more important than ever. The traditional approach to provide such an environment is by writing an interpreter in a low level system language like C. Usually, the interpreter implementation is then extended into a full-fledged virtual machine (VM). The interpreter part implements the syntax and semantics of the dynamic language generally by using some form of intermediate representation while the VM part provides infrastructure and runtime services like automatic memory management or a threading model. It is common that once a dynamic language has grown popular enough the interpreter is extended by a just-in-time (JIT) compiler to more efficiently execute interpreted code.

However this approach does not come without problems. The effort required to write a complete virtual machine from scratch is huge. It also forces the language implementer to deal with platform or target architecture specific details. This makes implementing a VM time consuming, error prone, and complicated to port to other systems. Additionally, modern dynamic languages like Python, Ruby or JavaScript evolve rapidly and have complex syntax and semantics. This not only makes it hard to represent the language properly in low level code, but makes maintenance of existing VM implementations a complicated task as well.

Looking at the development of dynamic language VMs during the last 20 years, one can identify two main issues authors tried to address with their work. The first issue is the high initial development effort necessary to implement an efficient low level VM. This is usually followed by tedious maintenance due to the often rapid advance of dynamic languages. Section 2 of this work presents two ideas that can help to reduce the development overhead and to speed up the creation and upkeeping of dynamic language VMs. The second issue is related to the execution of dynamic language programs. Compared to static languages where all properties of a source code fragment are inferred offline by a compiler, this is done online at runtime in dynamic languages. This makes them more difficult to interpret and execute efficiently in a virtualized environment. Section 3 has a look at two optimizations to improve execution speed of dynamic language programs.

2 WORKLOAD EFFICIENT VM IMPLEMENTATION

The following section of this work presents two main techniques that can be used to reduce the amount of work necessary to write a dynamic language VM. A straight forward idea is to hierarchically layer several VMs inside one another. Based on the principle of "divide and conquerer" the problem of providing a dynamic language runtime environment is split into several parts which can be handled by different VMs. The topic is discussed in detail in section 2.1. Another option is the usage of metaprogramming techniques. The idea is based on having a transformation toolchain which can automatically translate a high level implementation of a dynamic language interpreter into compilable low level VM code. This means that most of the target architecture specific code can be reused for several dynamic languages and only the interpreter has to be reimplemented for each language. A closer look at this approach is taken in section 2.2.

2.1 Hierarchical Layering of VMs

Hierarchical layering of VMs inside one another is based on the idea of decreasing implementation efforts by introducing modularization. All VMs generally have to provide implementations for similar problems like an efficient JIT compiler or memory management. However, for most dynamic language VMs a standard solution for these problems is sufficient. This allows for the creation of general purpose implementations which are deployed as modules and can be structured in layers. This layer abstraction not only allows to reuse own systems in several dynamic language implementations but also the integration of standard third party general purpose VMs like the Java Virtual Machine (JVM).

An implementation of this idea is provided by Yermolovich et al. in combination with a trace based JIT compiler [26]. A reiteration can be found in the work of Würthinger et al. who deployed the layering approach in combination with type specialization in the compiler infrastructure of their language implementation framework "Truffle" [23, 24]. Traced based JIT compilation as well as type specialization are described in greater detail in section 3. Both Yermolovich and Würthinger use a similar layered system architecture based on a well known general purpose VM implementation. Namely, Würthinger et al. use the Java Virtual Machine (JVM) implementation provided by the OpenJDK project [18] while Yermolovich et al. use Adobe's (now Mozilla's discontinued) Tamarin-Tracing VM [17]. A visualization of their system architecture can be seen in figure 1. Inside of the general purpose host VM another guest VM is executed which interprets and runs dynamic language programs inside the scope of the hosting VM. The main benefit of this idea is that the implementation of the guest VM does not have to be done on a low system level. Instead, the higher level managed language and runtime environment provided by the host VM can be used. For Yermolovich and Würthinger this mainly includes access to advanced services like automatic memory management and an optimized JIT compiler, both services which are tedious and error prone to implement manually in a low level system setting. Another aspect of the technique is, that a dynamic language VM implemented into another VM automatically becomes available on all architectures the host VM has been ported to. This implies that with one consistent code base it is possible to cover a potentially wide range of different target platforms.

However, the technique comes not without a cost: a major issue with layering VMs inside one another is that it can lead to subpar performance. This is noted by Yermolovich et al. who state that compared to their Lua interpreter on top of the Tamarin-Tracing VM "the native version of Lua is orders of magnitude faster for some benchmarks" [26]. In general, the reason for this observation is that general purpose VMs are almost always heavily optimized to run code written in their own managed language very well. This starts with aspects of the simulated system architecture like memory access, MMU behavior, or handling of interrupts and exceptions and continues with implementation specific optimizations applied by the VM's JIT compiler. To make programs executed by a guest VM inside of a hosting VM run fast, the byte code generated by the guest needs to match all "good case" criterions of the managed host language as close as possible. This requires a deep understanding

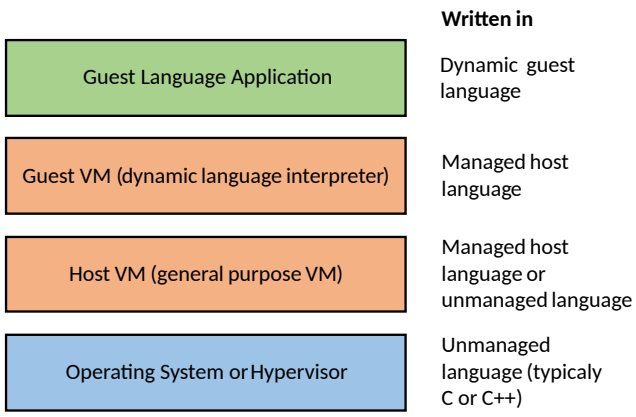

Figure 1: Generalized layered system architecture used both in Yermolovich's [26] and Würthinger's [24] work

of the host VM's architecture as well as a dynamic language with semantic patterns related to the ones of the managed host language.

Gaikwad et al. demonstrated a significant variation in performance when executing different dynamic guest language implementations on the same JVM instance using the "Truffle" framework [9]. Other insight on the performance of dynamic language programs running on the JVM is provided by Li et al. [13] and Sarimbekov et al. [20]. Both studies use a similar setup and heavily focus on runtime behavior using static code analysis as well as dynamic analysis of benchmark programs executed on different guest VMs. Metrics collected in both studies include method size, stack depth, method and basic block hotness, object lifetime and size, call-site polymorphism, immutability, and unnecessary zeroing. Their results show that there are noticeable differences in runtime behavior of regular Java byte code and byte code generated by dynamic language guest VMs. These differences can impact the guest program's performance when executed. Dynamic languages tend to use a significant amount of immutable objects. Objects are allocated more rapidly, are generally smaller and have a shorter lifetime compared to objects generated from Java code. The high number of short lived immutable objects allocated by dynamic language programs also leads to a high number of unnecessary zeroing. Finally, despite a high number of possibly polymorphic call-sites in dynamic language programs most actual method invocations happen at sites only targeting a single method. Worth mentioning is that regardless of all of these aberrations Li et al. [13] state that they did not find any major differences in method or block hotness in their evaluation.

Nevertheless, due to the general simplicity of the approach there is still a wide range of guest VM implementations available hosting dynamic languages like Python, Ruby or JavaScript.

2.2 Metaprogramming Techniques

Another approach which has been well discussed by the scientific community is the usage of metaprogramming techniques to significantly cut down the workload required to create a dynamic language VM. In a literal sense a metaprogram can be understood

as a computer program about another computer program. A more practical definition is given by Lilis et al. in their survey about metaprogramming languages [14]. They state that the main characteristic of metaprograms is that they treat other programs as data, enabling them to analyze or transform them. By doing so they can either modify existing programs or generate new programs from existing ones. Metaprogramming is a key concept in computer science and is deeply embedded in existing software development workflows. Any existing language compiler falls into the definition of a metaprogram as it manipulates higher level code to generate low level assembly language or machine code from it.

The way metaprogramming is used to reduce the development effort of writing a dynamic language VM is in form of a parser generator. The idea is to implement an interpreter for a dynamic language in a high level language and then use a translation toolchain to automatically generate low level VM code from it. The metaprogramming part of this technique can be found in the translation toolchain.

Bolz et al. go into more detail about the benefits of this approach in their paper called "How to not write VMs for dynamic languages [7]". Their main argument is that VMs can be generated automatically using a high level interpreter implementation which acts as a language specification. They argue that this technique not only has the same advantages as described in section 2.1 which was about using general purpose VMs as a base platform, but also brings additional benefits in terms of flexibility. Specifically, they state three major points: First, using a metaprogramming translation toolchain helps to keep one single code base for a dynamic language. From this code base, standalone VMs as well as guest modules for hosting VMs can be generated automatically. Second, the translation toolchain helps to introduce modularization and separation of concerns. This is especially useful later on because it generally makes maintenance easier than having to deal with monolithic software constructs. Third, since platform specific low level implementations are the concern of the translation toolchain, writing the interpreter module becomes easier. Together with the fact that the interpreter can be written in a dynamic language this leads to an increase in flexibility. As a result experimental features and techniques can be implemented and tested more easily without extensive code changes outside the interpreter module. However, Bolz et al. state some downsides as well [7]: First, moving low level functionality into the translation toolchain does not remove the necessity to deal with it at all but only defers it. The real benefit comes from having to do it only once and the ability to reuse it, rather than from not having to do it at all. For example, Bolz et al. state that they still had trouble to implement efficient garbage collection for their metaprogramming translation toolchain "PyPy" [7]. Second, generating a VM using the toolchain can take some time. For "PyPy" Rigo et al. note compile times up to ten minutes for the generation of a Python VM [19]. This reduces flexibility when testing new features. Third, VMs created with a translation toolchain generally perform worse then handwritten reference implementations. In the case of "PyPy", generated Python VMs ran up to 10 times slower than the reference CPython implementation when executing the Pystone benchmark program [19].

Still, there are many implementations available applying the idea of metaprogramming translation toolchains in practice: Some of the earliest examples date back to 1997, where Ingalls et al. [12] presented "Squeak", their Smalltalk implementation, whose virtual machine is entirely written in Smalltalk and to 2000, where Alpern et al. [1] published an article about their virtual machine for Java servers "Jalapeño" which is completely written in the Java language. A more recent example is the already mentioned "PyPy" toolchain by Rigo et al. [19].

3 PERFORMANCE EFFICIENT VM IMPLEMENTATION

The following section shifts its focus towards implementing runtime environments that can execute dynamic language programs efficiently, rather than having low implementation efforts. In general executing a dynamic language programs is done in two steps:

First, the program's source code is interpreted. A common way to interpret a dynamic language program is by transforming the program's logic written in lines of code into an intermediate tree or graph like representation. This representation can then easily be evaluated by recursively traversing its nodes. Second, the interpreted code is executed on the host hardware. A widely applied strategy to efficiently execute code in a VM setting is by implementing a just-in-time compiler (JIT). The idea is based around dynamic code generation at runtime. Interpreted guest code is cached in blocks which can later be reused if required. This way the same code does not have to be interpreted several times.

Section 3.1 of this chapter takes a closer look at a technique generally referred to as type specialization which can help increase performance during language interpretation. After that, section 3.2 focusses on an optimization of the basic JIT idea called trace based JIT compilation.

3.1 Type Specialization

Tree or graph based representations are a common way to make interpreting or compiling language code easier. An according intermediate data structure is usually generated by analyzing a given program's source code using a set of syntax rules defined by a language. The main advantage of the technique is that arbitrarily complex syntactic constructs can be described in a simple recursive manner. Furthermore, semantic evaluation of a created intermediate tree or graph representation can often be achieved with standard node visiting traversal strategies.

In the context of dynamic language execution using an intermediate data structure is considered a viable solution for code interpretation. However, due to the recursive nature of tree and graph traversal algorithms, using such a representation can cause a relatively high overhead. Since code interpretation of dynamic languages is performed at runtime, using a tree or graph based data structure can impact the execution performance of a dynamic language VM significantly. Therefore, there has been some work made to improve the performance of intermediate data structure based interpreters over the last years.

Williams et al. introduced a dynamic intermediate representation (DIR) based on a flow graph [22]. The representation allows them to perform type specialized interpretation of Lua programs. Each node in their DIR encodes the opcode of a specialized instruction while edges between them encode control-flow or type-flow. Type-flow

edges allow to keep track of a variable's data type during program execution. Edges encoding the correct type are selected by so called "type-directed" nodes which include a table of possible targets for every of the nine data types provided by the Lua language. The decision which one to choose is based on the resulting data type of the "type-directed" node's operation. This allows the interpreter to dynamically pick the correct path to continue its traversal through the DIR. Once visited, the chosen specialized node does not have to care about type checking of operands and can instead immediately perform its operation. The DIR also propagates types through method calls while keeping track of the most recently used parameter types to implement a caching mechanism comparable to other inline caches used to speed up runtime method binding for polymorphic call sites. Even though the dispatch overhead of the DIR interpreter is larger than the bytecode dispatch of the standard Lua interpreter, it can still achieve an average speedup of 1.3x according to Williams et al. [22].

Another technique to incorporate type specialization has been proposed by Würthinger et al. [25]. Their idea is based around a tree based intermediate representation called abstract syntax tree (AST). Every node of an AST represents an operation which uses its children as input operands. The node's children can again be operations with their own set of child operand nodes. This means that every inner node of an AST is an operand and an operation at the same time while leaf nodes are only operands. Würthinger et al. propose rewriting of AST nodes into more specialized versions during execution to dynamically incorporate available type feedback and profiling information from previous and current node operands. At the beginning of an AST's execution every node starts in an uninitialized state. This initial state does not provide any functional implementation which means that for each node type at least one generic implementation has to be defined. The generic implementation can be applied in any case and is used as a fallback. However, depending on the type of operation represented by a node additional specialized implementations can be provided. These specialized implementations make assumptions about their input operands, for example assuming a certain type. This usually allows them to execute faster but they lose their abstract nature and cannot handle all cases anymore. Based on the runtime properties of input operands the interpreter can rewrite a node dynamically to any matching specialized version. This decision is done optimistically, assuming that types stay stable in possible future executions. Still, if necessary, the interpreter can change a node's specialization at any time during execution. A simple example for this to happen is a node performing an arithmetical operation on two integers. The specialization of the node to use integer arithmetic changes in case one or both of the operands switches its type to floating point during program execution. A visualization of a transition model between type specialized versions of an AST node performing an addition can be seen in figure 2.

Würthinger et al. implemented an interpreter for JavaScript in Java which uses their tree rewriting technique. Furthermore, they added some optimization which utilizes the static typing and primitive types of Java to avoid the cost of having to use boxed data types. To evaluate their implementation they compared themselves to the JavaScript Rhino VM [16]. With tree rewriting enabled they were able to achieve a 4x speedup over the Rhino interpreter only.

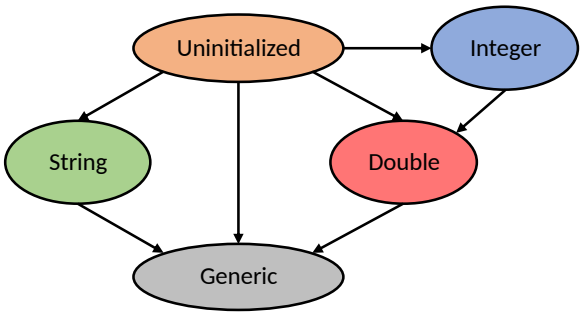

Figure 2: Transition model between different type specialized versions of an addition node as described by Würthinger et al. [25].

However, with its bytecode generation feature enabled Rhino still performed about 40% faster [25].

3.2 Tracing JIT Compilers

An efficient way of executing interpreted dynamic language code is by using a just-in-time (JIT) compiler. However, other than in static languages not all properties of a dynamic language program have to stay constant at runtime. This means that the JIT has to perform additional checks on cached code when reusing it to account for dynamic properties that may have changed. While it is possible to infer dynamic properties by, for example, using static analysis or profiling on intermediate code representations this hurts the JIT's performance since in any case control has to be given back to the interpreter. A way of keeping performance high and not having to fall back to the interpreter all the time has been explored by adapting trace based optimization strategies for dynamic languages.

Trace based optimization has a long history, although mostly applied to native or typed language code. The technique, for exampled was applied by Bal et al. as early as 1999 in their dynamic optimization system Dynamo [2, 3]. The goal of the system is to transparently improve the performance of native instruction streams like they are used by regular JIT compilers. The units of optimization at runtime are traces which are defined as a sequences of consecutively executed instructions. Traces have a fixed start and end address and may extend across several control structures like branches or methods. Dynamo picks a hot trace by using a speculative scheme based on the assumption that a program spends most of its time in loops. Therefore, all addresses that are reached via a backwards taken branch are considered a viable trace head. Counters are associated with each of the heads which are incremented every time their respective code address is executed. Once the counter has met a certain threshold the respective trace head is considered as "hot". Since only trace heads are monitored, another statistical assumption has to be used to generate a complete trace from a hot trace head: Blocks immediately executed after a hot trace head are very likely hot as well. This means that all basic blocks executed after a hot trace head has been identified are collected into a history buffer. This is continued until an end-of-trace condition is met. These can be conditions like the next block to be executed

is identical to the first block in the history buffer or the next instruction is a backwards taken branch. After a hot trace has been identified the associated blocks are converted into a low-level intermediate representation and optimized using strategies like copy and constant propagation, strength reduction, loop invariant code motion and loop unrolling [2]. Finally the optimized representation is converted to machine code and emitted into a managed cache.

The idea was extended by Gal et al. with the introduction of trace trees in their Hotpath Java VM implementation [11]. While using a similar algorithm for trace selection, the main difference is the subsequent tree based trace recording mechanism. A trace tree is defined to contain a set of basic blocks which are also called nodes and a set of direct edges between these nodes describing control-flow. After an appropriate trace head is found it is added to the trace tree as the anchor node. A trace tree can only contain one anchor node. Basic blocks are recorded until a cycle is found. The resulting trace is then added to the tree. Afterwards, the tree structure looks like a linked list where the anchor is the root of the tree and the last block of the traced instruction cycle is the only leaf node. The tree can subsequently be extend by side exits. Side exits are paths which originate from a node in the trace tree but are not already covered by it. To detect them during execution special guard instructions are compiled into the trace's bytecode as a replacement for the original conditional instructions. Once such an exit is detected by a guard during execution the new path is interpreted and recorded as a new trace. Finally, the new trace is added to the trace tree. Gal et al. integrated their tracing JIT compiler into JamVM [15]. Noteworthy is the low number of lines of code by the JIT which is only 1800 and the small memory footprint of the implementation which is around 128 KBytes. This makes their solution especially interesting for embedded environments with strict resource limitations. For highly regular programs they were able to achieve speedups up to 11x compared to pure interpretation. However, they note that for irregular code their implementation may not be able to pick up any traces at all and will yield no improvements in execution time [11].

In a later publication Gal et al. [10] use their trace based JIT compilation technique in the context of a dynamic language VM. Their implementation of a JavaScipt VM called TraceMonkey is based on their work with the Hotpath Java VM earlier described in this section. The main issue TraceMonkey has to solve when applying trace based optimization to dynamic language code is that type stability inside a found and compiled trace cannot be guaranteed but can only be assumed. Therefore additional guards checking the actual types of variables used in a compiled trace have to be added. In case any of the guards fail during execution, the corresponding trace has to exit and give control back to the interpreter. However, similar to how side exits due to changing control flow where handled by the Hotpath Java VM, TraceMonkey immediately starts recording the new type invariant path as a new trace. The new trace can subsequently be integrated into the original trace tree. As a result TraceMonkey's trace trees can contain several type specialized traces for one loop path. Gal et al. provide some examples on how this can look in practice which can be seen in figure 3. Furthermore, they report that their implementation can compete with Apple's JavaScript interpreter SquirrelFish Extreme (SFX) as well as Google's V8 JavaScript VM. Their implementation was the fastest of the three in nine of the 26 benchmarks in the

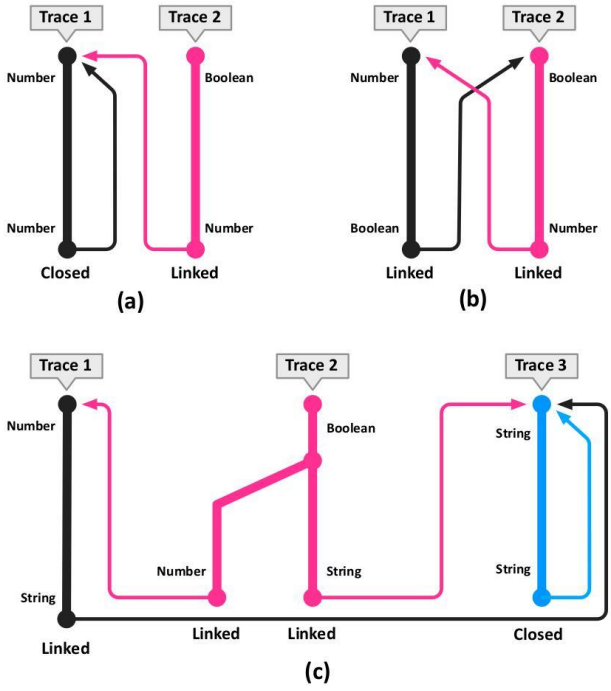

Figure 3: Gal et al. [10] link several traces found for a type unstable loops to form a group of trace trees that can execute without having to side-exit to the interpreter.

SunSpider benchmark suite. The execution of TraceMonkey was up to 25x faster compared to their base line interpreter called Spider Monkey [10].

Over the last years several dynamic language projects integrated a tracing JIT compiler into their runtime environment. Bolz et al. presented a tracing JIT compiler for their PyPy project in 2009 [5, 6]. The main difference compared to other implementations is that the JIT compilation is not applied to the user program but instead to the interpreter running it. Other examples are SPUR, a tracing JIT compiler for Microsoft's CIL [4] or Adobe's (now Mozilla's discontinued) Tamarin-Tracing for trace based execution of ActionScript [8].

4 CONCLUSION

In this work a closer look at techniques for creating dynamic language VMs was taken. The first part focused on implementing workload and resource efficient VMs. One presented idea showed that several VMs can be stacked inside one another which creates a hierarchical order and leads to modularization and layer abstraction. Another techniques focused on using metaprogramming concepts as they are usually seen in language compilers to automatically transform high level language interpreter implementations into low level VM code. Both described techniques help to significantly reduce the amount of complexity induced by writing and maintaining a single low level code base for every dynamic language.

The second part of this work presented two optimizations which help to improve the execution speed of dynamic language programs

inside VMs. Ideas were elaborated on how to use type specialization and tree like intermediate representations like ASTs to efficiently infer dynamic language features during interpretation at runtime. Furthermore, trace based optimization strategies were presented which help JIT compilers to better detect frequently used areas throughout applications and handle type instability more robustly.

While most of the proposed techniques shown in this work have seen successful implementation, writing a dynamic language VM with low implementation overhead that achieves good performance is still a big engineering challenge. All of the proposed techniques do not come without a cost. Being it either a loss of flexibility or a loss of runtime performance. Additionally, while some techniques aim to drastically reduce the required amount of low level system code which has to be written to create a VM, these solutions still often fail to abstract all low level properties completely. Therefore, in many cases it is just about the dynamic language creator's personal preferences and the required properties of a language's runtime environment to decide which of the shown techniques are worthwhile to implement.

REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238. https://doi.org/10.1147/sj.391.0211 Conference Name: IBM Systems Journal.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 1999. *Transparent dynamic optimization: the design and implementation of dynamo.* Technical Report. Hewlett Packard. 102 pages.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/349299.349303

[4] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. 2010. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 708–725. https://doi.org/10.1145/1869459.1869517

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '11)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2069172.2069181

[7] Carl Friedrich Bolz and Armin Rigo. 2007. How to not write Virtual Machines for Dynamic Languages. 3rd Workshop on Dynamic Languages and Applications (2007), 11.

[8] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. 2009. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '09)*. Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/1508293.1508304

[9] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. 2018. Performance analysis for languages hosted on the truffle framework. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3237009.3237019

[10] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. *ACM SIGPLAN Notices* 44, 6 (June 2009), 465–478. https://doi.org/10.1145/1543135.1542528

[11] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE '06)*. Association for Computing Machinery, New York, NY, USA, 144–153. https://doi.org/10.1145/1134760.1134780

[12] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. 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)*. Association for Computing Machinery, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[13] Wing Hang Li, David R. White, and Jeremy Singer. 2013. 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)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/2500828.2500838

[14] Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *Comput. Surveys* 52, 6 (Oct. 2019), 113:1–113:39. https://doi.org/10.1145/3354584

[15] Robert Lougher. 2014. *JamVM – A compact Java Virtual Machine.* Retrieved 2020-12-22 from http://jamvm.sourceforge.net/

[16] Mozilla. 2005. *Rhino documentation.* Retrieved 2020-12-21 from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/Documentation

[17] Mozilla. 2009. *Tamarin:Tracing - MozillaWiki.* Retrieved 2020-12-28 from https://wiki.mozilla.org/Tamarin:Tracing

[18] Oracle. 2020. *OpenJDK.* Retrieved 2020-12-28 from https://openjdk.java.net/

[19] Armin Rigo and Samuele Pedroni. 2006. 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)*. Association for Computing Machinery, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753

[20] Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. 2013. Characteristics of dynamic JVM languages. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages (VMIL '13)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/2542142.2542144

[21] The Software Quality Company. 2021. *TIOBE.* Retrieved 2021-01-25 from https://www.tiobe.com/tiobe-index/

[22] Kevin Williams, Jason McCandless, and David Gregg. 2010. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO '10)*. Association for Computing Machinery, New York, NY, USA, 278–287. https://doi.org/10.1145/1772954.1772993

[23] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[24] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[25] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages (DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

[26] Alexander Yermolovich, Christian Wimmer, and Michael Franz. 2009. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th symposium on Dynamic languages (DLS '09)*. Association for Computing Machinery, New York, NY, USA, 79–88. https://doi.org/10.1145/1640134.1640147