Sulong - Execution of LLVM-Based Languages on the JVM

Position Paper

Manuel Rigger
Johannes Kepler University, Linz, Austria
manuel.rigger@jku.at

Matthias Grimmer
Johannes Kepler University, Linz, Austria
matthias.grimmer@jku.at

Hanspeter Mössenböck
Johannes Kepler University, Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

For the last decade, the Java Virtual Machine (JVM) has been a popular platform to host languages other than Java. Language implementation frameworks like Truffle allow the implementation of dynamic languages such as JavaScript or Ruby with competitive performance and completeness. However, statically typed languages are still rare under Truffle. We present Sulong, an LLVM IR interpreter that brings all LLVM-based languages including C, C++, and Fortran in one stroke to the JVM. Executing these languages on the JVM enables a wide area of future research, including high-performance interoperability between high-level and low-level languages, combination of static and dynamic optimizations, and a memory-safe execution of otherwise unsafe and unmanaged languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Run-time environments, Code generation, Interpreters, Compilers, Optimization

Keywords Sulong, LLVM, Truffle, dynamic compilation, static compilation

1. Introduction

In recent years the Java Virtual Machine (JVM) has become a popular platform for languages other than Java and Scala. Frameworks like Truffle (Würthinger et al. 2013) allow building high-performance language implementations on top of a JVM. There are already Truffle implementations for high-level languages such as JavaScript, Ruby, R, or Python with competitive performance and completeness. However, apart from C (Grimmer et al. 2014), Truffle still lacks implementations for statically typed languages such as C++, D, Fortran, Objective-C, and Objective-C++. We present Sulong, an ecosystem that can execute and dynamically compile LLVM IR on a JVM, and can thus handle any language that can be compiled to LLVM IR. In this position paper we outline three future research directions based on Sulong. First, we want to use Sulong to efficiently implement native interfaces of high-level languages. Second, we want to investigate optimization opportunities when combining sophisticated static program analysis with speculative dynamic optimizations. Finally, we want to extend Sulong so that it guarantees spatial and temporal memory safety when executing programs.

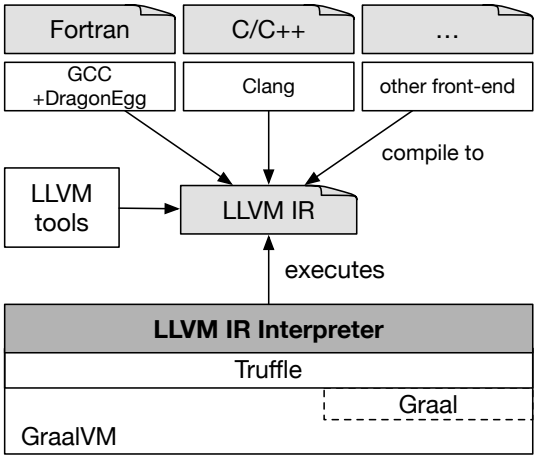

Figure 1. Sulong Interpreter

2. System Overview

Sulong is a modular ecosystem that reuses different LLVM front-ends such as Clang (C/C++) or GCC with DragonEgg (Fortran) that compile a program to LLVM IR. This LLVM IR is the input of our Truffle language implementation, which executes it on top of a JVM (see Figure 1).

2.1 LLVM

LLVM is a modular compiler framework engineered primarily for statically typed languages that envisions to perform

analysis and transformations during the whole life-time of a program (Lattner and Adve 2004). To achieve this goal, many parts of the framework work on a common IR, namely LLVM IR. LLVM's Clang front-end and other front-ends translate languages such as C, C++, and Fortran to LLVM IR. LLVM then performs compiler optimizations on the IR level before compiling the IR to machine code. However, we do not use the machine code but rather take the LLVM IR as the input for our Truffle interpreter. Executing LLVM IR minimizes implementation work, since existing LLVM front-ends already parse and compile the source languages to LLVM IR. Also, LLVM IR is platform independent and significantly easier to process than the source languages. For example, there are no OOP concepts in LLVM IR, and the interpreter does not have to deal with virtual function calls when executing C++ programs but only with plain function calls and function pointer calls.

2.2 Truffle Language Implementation

The core of Sulong is a Truffle language implementation for LLVM IR. Truffle is a framework for building high-performance language implementations in Java. Truffle language implementations are executable abstract syntax tree (AST) interpreters. These ASTs are self-optimizing, i.e, they use run-time feedback to replace certain nodes of the AST with more specialized versions during execution (Würthinger et al. 2012). A typical use case is type specialization for dynamically typed languages, for which nodes speculate on the data types of their operands. For example, the semantics of an add operation in JavaScript depends on the types of its operands. Truffle nodes specialize themselves according to type feedback, e.g., a general add operation node can replace itself with a faster integer add operation node if its operands have been observed to be integers.

When an AST execution count exceeds a predefined threshold, the Truffle framework assumes that the AST is stable. Then, Truffle uses the Graal compiler (Duboscq et al. 2013) to dynamically compile the AST to highly optimized machine code. When a speculative optimization (i.e., node rewriting) turns out to be wrong at run time, the VM discards the machine code and continues execution in the interpreter, until the compilation threshold is reached again.

In previous work we presented TruffleC (Grimmer et al. 2014), a C implementation on top of Truffle that implements speculative optimizations for C. For example, TruffleC monitors values of variables and replaces them with constants if they do not change over time. It also builds polymorphic inline caches for function pointer calls with varying call targets. Our dynamically compiled C code reaches a peak performance which is competitive with that of static optimizing compilers such as GCC or Clang. With Sulong we want to apply these optimizations to a broader scope of languages, e.g., we want to provide polymorphic inline caches also for C++ virtual function calls.

2.3 Sulong

Sulong is a modular and feature-rich ecosystem. It reuses different LLVM front-ends and can therefore execute languages such as C, C++, Objective-C, Swift, Fortran, and others. Compiling programs to LLVM IR prior to execution also allows us to reuse existing tools in the LLVM framework to transform the IR. For example, we can use the static optimization tool *opt* to apply static optimizations to the LLVM IR, and then use *llvm-link* to link several LLVM IR files to a single file. Eventually, we construct a Truffle AST from the LLVM IR and execute it using tree-rewriting specializations and dynamic compilation from the Truffle framework. This way, we can implement speculative optimizations for statically typed languages.

3. Research Directions

3.1 Language Interoperability and Native Interfaces

Modern software projects tend to be written in multiple different languages. Multi-language applications allow choosing the most suitable language for a given problem, gradually migrating code from one language to another, or reusing existing code. The Truffle framework features a mechanism for cross-language interoperability between different languages (Grimmer et al. 2015b,c), which allows executing multi-language applications. This mechanism allows calling and inlining of functions across any language boundary and exchanging data without marshalling or conversion. With Sulong we add low-level, statically typed languages such as C/C++ and Fortran to this ecosystem. We want to use Sulong to provide an efficient implementation of the different native interfaces of various high-level languages.

While most foreign function interfaces have to convert or marshall between language boundaries, a Truffle language implementation can instead use Sulong to efficiently call native code. For example, we want to run native modules of NodeJS on Sulong or use Sulong to execute native R packages.

To evaluate the feasability of using Sulong as a Truffle native interface, we intend to make case studies on executing Ruby programs that use C extensions and R programs that rely on R native packages. In these case studies we want to evaluate the performance compared to conventional native interfaces and demonstrate the flexibility of Truffle's interoperability mechanism.

3.2 Static vs. Dynamic Optimizations

Static compilers such as LLVM or GCC can apply time-intensive analyses and optimizations, since all this happens before distributing and running the program. In contrast, dynamic compilers can use profiling information and speculative optimizations to increase peak performance, but cannot apply time-intensive optimizations since they would have to be performed every time a program is executed.

Previous approaches showed that the upper boundary of static compilation approaches can be further pushed by applying profiling information (Bala et al. 2000; Nuzman et al. 2013). We want to further experiment with this approach, by combining both static and dynamic optimizations to improve the peak performance of C, C++, and Fortran programs.

We intend to use LLVM to apply optimizations based on pointer analysis such as promotion of memory to virtual registers or elimination of memory loads on the LLVM IR before execution. During execution, we use our interpreter to perform speculative optimizations, such as value profiling (Calder et al. 1997), polymorphic inline caches (Hölzle et al. 1991), and branch probability recording. Peak performance profits from the expensive static optimizations by LLVM, as well as from our dynamic optimizations at run time. Following this approach, we expect to top the peak performance of pure static or dynamic compilation approaches.

To evaluate peak performance we intend to use the Computer Language Game benchmarks but also the larger SPEC CPU benchmarks. For a static compilation baseline we will use Clang O3 to compile the benchmarks to binaries, execute them and measure the execution time. For a dynamic compilation baseline, we will use Clang to compile them to bitcode files without applying static optimizations and execute them with Sulong. For the combined approach where we additionaly apply static optimizations on the bitcode file, we will create a matrix of benchmark run-times with a combination of static optimizations on one axis, and dynamic optimizations on the other axis. Due to the phase ordering problem and thus the infinite number of possible optimization combinations (Touati and Barthou 2006) we will only be able to select a subset of combinations. We want to demonstrate that we can find a combination of static dynamic optimizations that will on average top both the static and dynamic baselines.

3.3 Memory Safety

Memory errors (e.g., accessing memory outside of objects) rank among the most serious security threats in languages such as C or C++ and still have not been addressed adequately (van der Veen et al. 2012). Although there are countless memory safety approaches, existing approaches either suffer from performance overhead, incompatibility with legacy software, incomplete protection, or a dependency on the compiler chain (Szekeres et al. 2013).

With Sulong, we want to re-use the JVM's type and bounds checking as well as automatic memory management to guarantee spatial and temporal memory safety. Spatial memory safety guarantees that no accesses occur outside of objects, while temporal memory safety ensures that no dangling pointers are accessed (Szekeres et al. 2013). By using Java allocations and relying on a garbage collector, Sulong can obtain complete memory safety.

In our previous work we implemented a memory safe version of TruffleC, which guaranteed spatial and temporal memory safety while only being 15% slower than GCC with optimizations turned on (Grimmer et al. 2015a). With Sulong, we want to apply this approach to the execution of LLVM IR to support memory safety for arbitrary LLVM languages. We also want to demonstrate that memory safety scales to large C/C++ programs in terms of execution speed.

To evaluate peak performance, we want to apply both static and dynamic optimizations and use the benchmarks mentioned in the previous section. To evaluate memory safety, we want to use NIST's Juliet test suite (Center for Assured Software 2012) which provides synthetical tests for various vulnerability classes including memory errors. We expect that Sulong can detect all memory errors that are exhibited during run-time. Also, we want to validate memory safety on real world programs to demonstrate that Sulong is a sound approach in practice. To find programs with attack vectors, we will use NIST's National Vulnerability Database (NIST 2016).

4. Limitations

While we expect that Sulong will provide excellent peak performance, warm-up will contribute to the total execution time, especially for short-running applications. Also, while traditional static compilation approaches have predictable performance; warm-up and invalidated speculations cause dynamic compilation approaches to have varying performance. This can especially be a problem in real-time systems and security-critical applications that are sensitive to timing-based attacks. Although we cannot offer a complete solution for this problem, one way to significantly improve interpreter speed is to use the SubstrateVM, where the Sulong interpreter is compiled to an executable instead of being run on a JVM.

Another issue is that real world programs often rely on undefined behavior for which most static compilers produce programs that match the programmer's intent. An example are handcrafted pointers obtained by converting pointers to integers, performing computations on them, and converting them back to pointers. Sulong will not attempt to support all such cases, but similarly to ManagedC provide a *relaxed* and *strict* mode. In relaxed mode certain operations (e.g., type punning) that result in undefined behavior are supported, and in strict mode undefined behavior results in an error. Thus, Sulong will not be able to support all real world programs. On the other hand, Sulong will improve the portability of programs similarly as GCC's and LLVM's undefined behavior sanitizers.

5. Summary

With Sulong, we will bring a variety of additional languages such as C, C++, and Fortran to the JVM. Sulong will allow us to experiment with the improvement of native function interfaces by basing on the multi-language environment given by the JVM. We also want to top the performance of

static compilers, by combining both static and dynamic optimization approaches. Finally, we want to guarantee memory safety for C/C++ by relying on the type and bounds checking as well as on the memory management of the underlying JVM.

Acknowledgments

We thank the Virtual Machine Research Group at Oracle Labs and the members of the Institute for System Software at the Johannes Kepler University Linz for their support and contributions. The authors from Johannes Kepler University are funded in part by a research grant from Oracle. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

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, 2000. ACM. doi: 10.1145/349299.349303. URL `http://doi.acm.org/10.1145/349299.349303`.

B. Calder, P. Feller, and A. Eustace. Value profiling. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 259–269. IEEE, 1997.

Center for Assured Software. Juliet test suite v1.2 for c/c++. Technical report, National Security Agency, 2012. URL `https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf`.

G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. doi: 10.1145/2542142.2542143. URL `http://doi.acm.org/10.1145/2542142.2542143`.

M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. Trufflec: Dynamic execution of c on a java virtual machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 17–26, New York, NY, USA, 2014. ACM. doi: 10.1145/2647508.2647528. URL `http://doi.acm.org/10.1145/2647508.2647528`.

M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe execution of c on a java vm. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, PLAS'15, pages 16–27, New York, NY, USA, 2015a. ACM. doi: 10.1145/2786558.2786565. URL `http://doi.acm.org/10.1145/2786558.2786565`.

M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 78–90, New York, NY, USA, 2015b. ACM. doi: 10.1145/2816707.2816714. URL `http://doi.acm.org/10.1145/2816707.2816714`.

M. Grimmer, C. Seaton, T. Würthinger, and H. Mössenböck. Dynamically composing languages in a modular way: Supporting c extensions for dynamic languages. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 1–13, New York, NY, USA, 2015c. ACM. doi: 10.1145/2724525.2728790. URL `http://doi.acm.org/10.1145/2724525.2728790`.

U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004. doi: 10.1109/CGO.2004.1281665.

NIST. National vulnerability database, 2016. URL `https://web.nvd.nist.gov/view/vuln/search`.

D. Nuzman, R. Eres, S. Dyshel, M. Zalmanovici, and J. Castanos. Jit technology with c/c++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Archit. Code Optim.*, 10(4):59:1–59:25, Dec. 2013. ISSN 1544-3566. doi: 10.1145/2541228.2555315. URL `http://doi.acm.org/10.1145/2541228.2555315`.

L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society. doi: 10.1109/SP.2013.13. URL `http://dx.doi.org/10.1109/SP.2013.13`.

S.-A.-A. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, pages 147–156. ACM, 2006.

V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 86–106, Berlin, Heidelberg, 2012. Springer-Verlag. doi: 10.1007/978-3-642-33338-5_5. URL `http://dx.doi.org/10.1007/978-3-642-33338-5_5`.

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, 2012. ACM. doi: 10.1145/2384577.2384587. URL `http://doi.acm.org/10.1145/2384577.2384587`.

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, 2013. ACM. doi: 10.1145/2509578.2509581. URL `http://doi.acm.org/10.1145/2509578.2509581`.