Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle

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

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

Christian Wimmer
Oracle Labs
christian.wimmer@oracle.com

Thomas Würthinger
Oracle Labs
thomas.wuerthinger@oracle.com

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

Abstract

Although the Java platform has been used as a multi-language platform, most of the low-level languages (such as C, Fortran, and C++) cannot be executed efficiently on the JVM. We propose Sulong, a system that can execute LLVM-based languages on the JVM. By targeting LLVM IR, Sulong is able to execute C, Fortran, and other languages that can be compiled to LLVM IR. Sulong combines LLVM's static optimizations with dynamic compilation to reach a peak performance that is near to the performance achievable with static compilers. For C benchmarks, Sulong's peak runtime performance is on average 1.39× slower (0.79× to 2.45×) compared to the performance of executables compiled by Clang O3. For Fortran benchmarks, Sulong is 2.63× slower (1.43× to 4.96×) than the performance of executables compiled by GCC O3. This low overhead makes Sulong an alternative to Java's native function interfaces. More importantly, it also allows other JVM language implementations to use Sulong for implementing their native interfaces.

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

Keywords LLVM, JVM, Sulong, dynamic compilation

1. Introduction

The Java Virtual Machine (JVM) has been recently used as a platform for not only Java and Scala, but also for dynamic languages including Ruby, Python, and JavaScript (Rose 2009). Having the JVM as a common platform enables cross-language interoperability so that Java code can call functions or methods written in other languages. Language implementation frameworks such as Truffle (Würthinger et al. 2013) feature a mechanism for cross-language interoperability, which allows writing efficient multi-language applications (Grimmer et al. 2015b). However, except from a C implementation (Grimmer et al. 2014, 2015a), there are no efficient Truffle implementations of lower-level languages, e.g., Fortran, C++, and others. To call functions written in such languages, developers have to resort to the Java Native Interface (JNI, Liang 1999) or other native function interfaces. These native function interfaces add runtime overhead since data structures have to be converted or (un)marshalled when transferring data between Java and the target language. Also, language boundaries are compilation boundaries, so a compiler cannot, for example, apply function inlining across languages.[1]

In this paper we present Sulong, a system that enriches the JVM with a variety of new languages by executing LLVM IR on the JVM. Sulong includes a new LLVM IR interpreter, which allows it to execute all languages that have an LLVM IR front end, including C/C++, Fortran, Ada, and Haskell. Developers can use the interpreter as a Java library to execute these languages on the JVM. We implemented the LLVM IR interpreter in Java on top of the Truffle framework (Würthinger et al. 2013), so that Sulong does not only interface with Java but also provides seamless interoperability with other Truffle language implementations (Grimmer et al. 2015b) such as R (Stadler et al. 2016), Ruby (Seaton 2015), and JavaScript (Würthinger et al. 2013). This in-

[1] Stepanian et al. (Stepanian et al. 2005) show that inlining native C code into Java is important and improves performance significantly. However, they convert the native code to the same intermediate language as the JIT compiler uses while we want to directly run low-level code on the JVM.

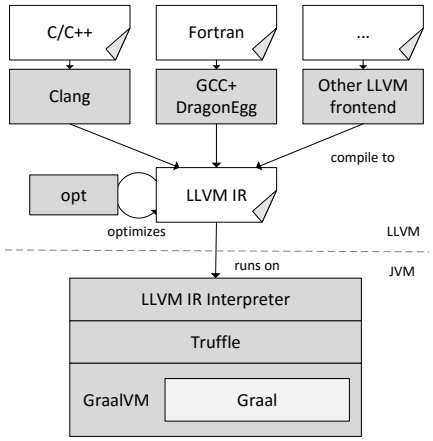

Figure 1. System overview.

teroperability mechanism allows optimizations across language boundaries such as cross-language function inlining. Furthermore, by having Truffle as a common base, other language implementations can use Sulong to implement their Native Function Interfaces (NFIs). For example, JRuby+Truffle (a Truffle implementation of Ruby, Seaton 2015) already uses Sulong to implement C extension support and FastR (a Truffle implementation of R, Stadler et al. 2016) experiments implementing support for native extensions with it.

Efficiency is very important to make Sulong an alternative to NFIs for both Java developers and Truffle language implementers. To achieve the necessary performance, Sulong combines LLVM's static optimizations at compile-time with a dynamic compiler at run-time. We use the Graal dynamic compiler (Duboscq et al. 2013; Stadler et al. 2014) to compile frequently executed LLVM IR functions to native code. This allows Sulong to reach peak performance that is near to the performance of code produced by industrial-strength compilers such as Clang.

In summary, this paper contributes the following:

- We describe how we bring a variety of languages to the JVM by using LLVM front ends and implementing a self-optimizing LLVM IR interpreter.
- We present a novel compilation approach to dynamically compile LLVM IR.
- We describe how we use static optimizations in combination with dynamic compilation to generate efficient machine code and demonstrate its peak performance on a range of C and Fortran benchmarks.

2. System Overview

Sulong is a modularized system that uses parts of LLVM and the JVM (see Figure 1). In this section we describe LLVM and Truffle + Graal, which are the basis of Sulong.

```
void processRequests() {
    int i = 0;
    do {
        processPacket();
        i++;
    } while (i < 10000);
}
```

Figure 2. A small C program containing a loop.

2.1 LLVM

LLVM (Lattner and Adve 2004) is a modular static compilation framework that consists of a standardized IR (called LLVM IR or bitcode) and a set of libraries. LLVM front ends translate a source program to an LLVM IR program. LLVM's official front-end is Clang which can compile C, C++, Objective C, and Objective C++. To enable GCC to compile its supported languages including Ada, Fortran, and Go to LLVM IR, one can use the DragonEgg plugin.[2] After compilation, a user can decide to further process the LLVM IR file, e.g., by using the LLVM static optimization tool *opt* to optimize the program. To get an executable from the LLVM IR file one can use the LLVM linker and assembler to link the LLVM IR files and to compile them to machine code. Sulong consists of a Truffle interpreter that we use to execute this IR on the JVM (see Section 3).

Figure 2 shows a C program, and Figure 3 the corresponding LLVM IR program in textual form. In LLVM IR, as in most IRs, a function comprises basic blocks that consist of sequential instructions and end with a terminator instruction that transfers control to the next basic block. For example, `br label %1` is an unconditional branch to the basic block labeled `%1`, `br i1 %3, label %1, label %4` is a conditional branch (depending on the boolean value `%3`) to the basic blocks labelled `%1` or `%4`, and `ret void` is a return from the function. These branches transfer control between basic blocks, similar as in non-structured programming languages that use `goto`. The biggest challenge for Sulong's LLVM IR interpreter is to efficiently interpret and dynamically compile the dispatch between basic blocks.

LLVM IR is in Static Single Assignment form (SSA, Cytron et al. 1991), i.e., each variable is only assigned once. In LLVM IR, these variables are called virtual registers and are prefixed with `%`. To merge assignments to the same variable after branches, LLVM IR uses phi functions. For example, in `%i.0 = phi i32 [ 0, %0 ], [ %2, %1 ]`, the value assigned to `%i.0` is `0` when the predecessor block is `%0` and `%2` if the predecessor block is `%1`. Sulong's LLVM IR interpreter needs to implement these virtual registers of LLVM IR as well as the native memory access that low-level languages use.

[2] `http://dragonegg.llvm.org/`

```
define void @processRequests() #0 {
; (basic block 0)
  br label %1

; <label>:1 (basic block 1)
  %i.0 = phi i32 [ 0, %0 ], [ %2, %1 ]
  call void @processPacket()
  %2 = add nsw i32 %i.0, 1
  %3 = icmp slt i32 %2, 10000
  br i1 %3, label %1, label %4

; <label>:4 (basic block 2)
  ret void
}
```

Figure 3. LLVM IR of the C program in Figure 2.

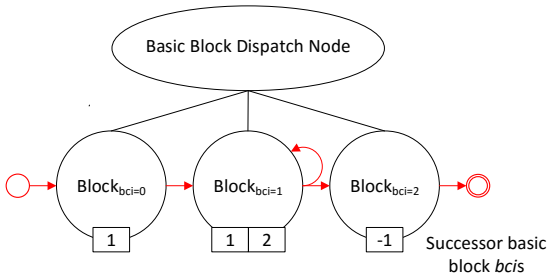

Figure 4. Basic block dispatch node for Figure 3

2.2 Truffle

Truffle (Würthinger et al. 2013) is a language implementation framework to build high-performance Abstract Syntax Tree (AST) interpreters on the JVM. Each node in a Truffle AST has an *execute* method in which it executes its children and returns its own result. Truffle AST interpreters are self-optimizing (Würthinger et al. 2012) in the sense that AST nodes can speculatively rewrite themselves with *specialized* variants at run time, e.g., based on profile information obtained during execution such as type information. For example, our LLVM IR interpreter can optimize indirect function calls by rewriting the indirect call node to a specialized node that speculates on a constant call target and can thus build polymorphic inline caches (Hölzle et al. 1991). In turn, this optimization enables speculative function inlining of indirect calls.

If these speculative assumptions turn out to be wrong, the specialized tree can be reverted to a more generic version that provides functionality for all possible cases. Truffle guest languages use self-optimization via tree rewriting as a general mechanism for dynamically optimizing code at run-time. For example, if an indirect function call is highly polymorphic, Truffle languages rewrite the polymorphic inline cache to a node that performs the lookup and calls the function.

2.3 Graal

When the execution count of a Truffle AST reaches a predefined threshold, Truffle uses the dynamic Graal compiler (Duboscq et al. 2013; Stadler et al. 2014) to compile the AST to machine code. The compiler assumes that the AST is stable and inlines node execution methods of a hot AST into a single method (known as partial evaluation, Futamura 1999) and performs aggressive optimizations over the whole tree. Graal inserts deoptimization points (Hölzle et al. 1992) in the machine code where the speculative assumptions are checked. If they turn out to be wrong, control is transferred back from compiled code to the interpreted AST, where specialized nodes can be reverted to a more generic version.

2.4 Sulong

Sulong uses LLVM front ends to compile source languages such as C/C++ or Fortran to LLVM IR and interprets it on the JVM. To simplify and optimize an LLVM IR program prior to interpretation, Sulong uses LLVM's static optimizers. Sulong then executes the LLVM IR on the JVM using a new Truffle interpreter. This Truffle language implementation brings all LLVM languages to the JVM, and makes them accessible to other Truffle language implementations.

Sulong's interpreter optimizes the AST based on the profile feedback that it observes at run time. Eventually, Truffle uses Graal as a dynamic compiler to compile the program to machine code, from which execution continues with native speeds. This architecture allows Sulong to profit from both static optimizations by LLVM, and dynamic optimizations by Truffle and Graal.

3. Execution of Unstructured Control Flow

The LLVM IR interpreter is different from previous language implementations on top of Truffle since it has to deal with unstructured control flow that cannot easily be handled in an AST interpreter. Support for unstructured control flow is the key for enabling the execution of LLVM IR, both in the interpreter and in the dynamically compiled code.

3.1 Interpreter

To support unstructured control flow in the interpreter we follow a mixed AST execution and bytecode interpretation approach. Basic blocks only contain sequential instructions, hence, we build ASTs for them. We do not build ASTs to implement transferring control between the basic blocks, since unstructured control flow cannot be directly modeled using ASTs. We could convert the unstructured LLVM IR programs to structured programs (Erosa and Hendren 1994), at the expense of making the implementation more complicated and removing the direct correspondence between LLVM IR instructions and Truffle nodes. Instead, we use a basic block dispatch node to transfer control between the basic blocks (and also add support for its compilation, see Section 3.2). Each function has such a basic block dispatch node. In the loop of the basic block dispatch node (see Fig-

```
int bci = 0;
while (bci != -1)
    bci = blocks[bci].execute();
```

Figure 5. Sulong's basic block dispatch node.

ure 5), we execute a basic block in each iteration, starting from a bitcode index of zero (bci = 0). Each node that represents a basic block contains an `int[]` array with the bcis of its successor blocks, which allows the compiler to see all possible successors of a block, i.e., the successor bcis are compile-time constants. The compiler needs this information to compile the basic block dispatch node (see Section 3.2). When executing a basic block, the basic block computes an index into this successor array, which it uses to return the next bci. Execution of basic blocks continues until bci = -1 which signals a return statement.

For the program in Figure 3, the basic block dispatch node transfers execution between three basic blocks that have consecutive indices from 0 to 2. Figure 4 shows the basic block dispatch node for this program and illustrates the control flow between the basic blocks with red arrows. Execution starts with the first basic block $\text{block}_{bci=0}$. $\text{Block}_{bci=0}$ has only one possible successor ($\text{block}_{bci=1}$), therefore its successor array contains only one element, namely bci = 1. The basic block dispatch node executes $\text{block}_{bci=0}$, and reads the next bci = 1 from its successor array. $\text{Block}_{bci=1}$ has two possible successors ($\text{block}_{bci=1}$, the loop body; and $\text{block}_{bci=2}$, the loop exit), therefore the successor array contains two elements, namely bci = 1 and bci = 2. Again, the basic block dispatch node executes $\text{block}_{bci=1}$, and returns either bci = 1 or bci = 2 from its successor array. The successor of $\text{block}_{bci=2}$ is bci = -1, which signals a return from the function.

3.2 Compilation

When compiling an AST, the Graal compiler has to recursively inline the execution methods of all AST nodes. While this is trivial for a regular AST, Graal has to treat the basic block dispatch node differently. For the basic block dispatch node, the compiler unrolls the loop (`while (bci != -1)`, see Figure 5) until all paths through the program are expanded. With respect to the program in Figure 3, the compiler starts with a bci = 0 and determines all successors of $\text{block}_{bci=0}$. The successor of $\text{block}_{bci=0}$ is $\text{block}_{bci=1}$. The compiler can peel the first iteration, and thus moves the execution of $\text{block}_{bci=0}$ out of the loop. Figure 6 illustrates this first step of the loop expansion in pseudo code; note that the first loop iteration (the execution of $\text{block}_{bci=0}$) is peeled. Next, the compiler determines the successors of $\text{block}_{bci=1}$, which are $\text{block}_{bci=1}$ (i.e., the loop body) and $\text{block}_{bci=2}$ (i.e., the loop exit). The compiler detects when a path has already been expanded and merges it with the existing path, which guarantees that the loop expansion terminates. In our

```
  blocks[0].execute();            // bci = 1
  bci = blocks[1].execute();      // to be expanded
```

Figure 6. Step 1: Unrolling the loop of the basic block dispatch node.

```
  blocks[0].execute();            // bci = 1
merge1:
  bci = blocks[1].execute();      // bci = 1 or 2
  if (bci == 1)
      goto merge1;
  else
      bci = blocks[2].execute(); // to be expanded
```

Figure 7. Step 2: Unrolling the loop of the basic block dispatch node.

```
  blocks[0].execute();            // bci = 1
merge1:
  bci = blocks[1].execute();      // bci = 1 or 2
  if (bci == 1)
      goto merge1;
  else
      blocks[2].execute();        // bci = -1
      return;
```

Figure 8. Final state: Unrolled loop of the basic block dispatch node.

example, the compiler sees that it has already expanded $\text{block}_{bci=1}$, and inserts a backjump ($\text{block}_{bci=1}$ has itself as a successor, so the compiler detected a loop). The second successor of $\text{block}_{bci=1}$ is $\text{block}_{bci=2}$, which the compiler expands. Figure 7 shows how the successors of $\text{block}_{bci=1}$ are expanded; note that the compiler inserts a jump (`goto merge1`) if it detects a path that has already been expanded ($\text{block}_{bci=1}$ has itself as a successor). Finally, the compiler expands the successors of $\text{block}_{bci=2}$, of which there are none (indicated by bci = -1). The bci = -1 terminates the loop and the compiler has finished loop unrolling. Figure 8 shows how the successors of $\text{block}_{bci=2}$ are expanded; note that the compiler inserts code to return from the function (`return`) if it detects a path that lets the basic block dispatch loop terminate. The Graal compiler then further optimizes the graph obtained by this partial evaluation.

4. Native Calls and Memory Management

One concern for Sulong is seamless and efficient interoperability with native shared libraries such as the C standard library. Reusing existing code in low-level languages such as C/C++ is commonly done by linking user programs against a shared native library that is present as a machine code binary but not available as source code (e.g. the C standard library). Sulong uses the Graal Native Function Interface (Graal NFI, Grimmer et al. 2013) to call native functions of such a library. When Graal compiles the AST to machine code, the

compiled Java code directly (i.e., without overhead) calls the native function.

To be interoperable with native functions, the Graal NFI expects its caller to either pass primitive values (by value) or unmanaged objects such as structs or arrays (by reference). Sulong aligns LLVM IR objects (structs, arrays, and vectors) using the same layout as in executables produced by static compilers. It reads this layout information from the bitcode file. When Sulong calls a native function, this native function can directly operate on allocations provided by Sulong, since they match the platform's Application Binary Interface. Thus, Sulong does not need to marshal or convert objects when calling shared library functions, and can call native functions with zero overhead when compared to native to native calls in executables. Following the object layout of static compilers also allows programmers to not only rely on standard C, but even to run programs that rely on undefined aspects of the memory layout when accessing native memory. This is useful in practice, since many programmers rely on what today's compilers do and not what ISO C specifies (Memarian et al. 2016). Sulong allocates, deallocates, and accesses unmanaged memory using the JDK internal *sun.misc.Unsafe* API.

To execute LLVM IR, Sulong has to support two types of unmanaged memory:

Stack: LLVM IR has an *alloca* instruction to allocate stack memory. To implement stack memory, Sulong allocates a block of memory at the start of the program and assigns its address to a stack pointer. The implementation of the *alloca* instruction then increments this stack pointer to allocate memory on the stack.

Heap: LLVM IR can allocate heap memory using external calls to a library function such as *malloc* from the C standard library. Heap memory allocation is transparent for Sulong and is handled like any other external call to a shared library.

5. Static and Dynamic Optimizations

By default, LLVM front ends such as Clang compile local variables in C/C++ to LLVM IR instructions that allocate the variables on the stack. Once a local variable is needed, it is loaded from memory and assigned to a virtual register. Thus, unoptimized LLVM IR programs have many stack allocations and memory accesses that could be avoided by keeping variables in virtual registers as long as their addresses are not needed and the variables have a primitive type. Storing local variables in memory is especially a problem for Sulong: The Graal compiler does not optimize allocations and accesses to unmanaged memory since Java programs mostly use managed memory. To overcome this shortfall, Sulong uses static LLVM optimizations to reduce the number of allocations and accesses to unmanaged memory. LLVM offers the *mem2reg* optimization which attempts to lift such stack allocations to virtual registers or constants. Sulong applies this optimization to reduce native memory accesses which enables the Graal compiler to produce more efficient machine code. Sulong's LLVM IR interpreter efficiently represents virtual registers (see Section 2.1) as Java objects that Graal can optimize well. In compiled code, virtual registers map to machine registers, or are allocated on the stack.

Besides *mem2reg*, LLVM provides other optimizations that reduce memory accesses such as dead store elimination, promote "by reference" arguments to scalars, and handle loop invariant code motion.

In addition to the static optimizations by LLVM Sulong performs several dynamic optimizations that cannot be performed by classic static compilers. On the Truffle level Sulong performs the following optimizations:

Runtime Inlining: Truffle performs profiling-based inlining during run-time. While we could use LLVM to perform static inlining we defer inlining to the run time since Truffle can exploit profiling feedback such as function call counts that can lead to better inlining decisions.

Dynamic Dead Code Elimination: We profile the probability of basic block successors in our basic block dispatch node. Graal will not compile a basic block that has never been executed and instead inserts a deoptimization point. This effectively results in a dynamic dead-instruction elimination (Butts and Sohi 2002), since Graal only considers those nodes for compilation that have been executed by the Sulong interpreter. Additionally, the successor probability profiling helps Graal during optimization and enables re-ordering of basic blocks based on the frequency of their execution.

Value profiling: We identify run-time-invariant memory values (Calder et al. 1997) by observing if a loaded memory value does not change, and replace such a load node by a node that checks if the value is still the same and returns the cached constant. When Graal compiles the node, it can propagate the profiled constant through constant folding and other optimizations. This optimization is especially beneficial for global variables that are set at the beginning of a program (e.g., configuration values) and do not change afterwards.

Polymorphic inline caches: We construct polymorphic inline caches (Hölzle et al. 1991) for function pointer calls. The first time we indirectly call a function the call site caches the target function up to a certain cache size. Subsequent calls then first check if the current function pointer is one of the cached target functions, and if so, perform a direct call to the function. Guarded direct calling enables Truffle to inline function pointer calls which eliminates the call overhead and enables optimizations on a larger range of code. If the number of cached functions exceeds a predefined threshold, we perform a nor-

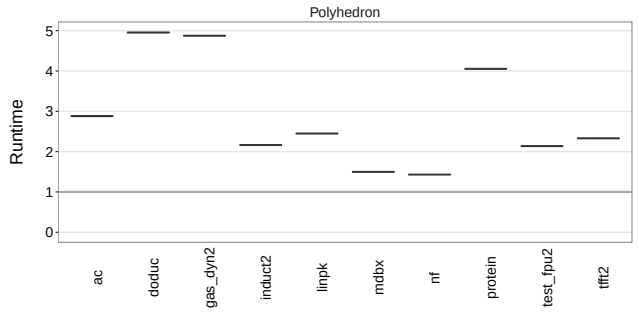

Figure 10. Polyhedron benchmark suite; peak performance (lower is better, relative to *GCC O3*)

mal indirect call since the inlining benefits are not likely to amortize the additional checks.

6. Evaluation

To evaluate Sulong, we choose C and Fortran as two LLVM languages. We do not evaluate C++ since we do not yet support LLVM IR exception handling. We use LLVM's official front end Clang to compile C to LLVM IR. Since Clang cannot compile Fortran, we use GCC with the DragonEgg plugin to compile Fortran to LLVM IR.

6.1 Benchmarks

To evaluate Sulong, we use all single-threaded C benchmarks from the *Computer Language Benchmark game* (shootouts)[3]. The shootouts are small benchmarks (66-453 LOC[4]) designed to compare the performance of different languages. They are useful as a base for the comparison of language implementations, since language implementers commonly use them as an optimization target (Barrett et al. 2016; Marr et al. 2016). We also include the whetstone[5], deltablue[6], and richards[7] benchmarks (239 to 839 LOC) since they are similarly popular small benchmarks for C.

Sulong is still a prototype and in an early stage. It cannot yet execute all SPEC CPU benchmarks. However, we want to also present performance numbers on real world applications. Sulong can already execute an application for compression using bzip2 (5k LOC) and gzip (5K LOC), and an application that converts an audio file using oggenc (48K LOC). These benchmarks are part of the *Large scale compilation-unit C programs* [8].

The same is true when executing Fortan on top of Sulong. Sulong can run 10 benchmarks from the *Polyhedron Benchmark Suite*[9], which in total consists of 17 mixed-size (161 LOC - 27K LOC) benchmarks to evaluate Fortran compiler implementations.

The benchmarks from SPEC CPU and the Polyhedron Benchmark Suite that are not part of our evaluation cannot be executed by Sulong. Sulong either fails parsing their LLVM IR, crashes because of implementation bugs, or reports an unimplemented feature. We are convinced that the implementation of missing features and resolving the known issues is possible with reasonable effort in the future.

6.2 Experimental Setup

To account for the adaptive compilation techniques of Truffle and Graal, we set up a harness that warms up the benchmarks. After the warm-up iterations, every benchmark reaches a steady state such that subsequent iterations are identically and independently distributed. We execute each C benchmark 100 times and use the last 50 iterations to compute the runtime. Since the Fortran benchmarks warm up faster and run longer, we execute them 20 times and use the last 10 iterations to compute the runtime.

We measure the peak performance of C and Fortran code on top of Sulong and then compare it with the performance of executables generated by the static compilers Clang (for C), and GCC (for Fortran). We focus this evaluation on peak performance of long-running applications where the startup performance plays a minor role. Hence, we neglect the startup time and present performance numbers after an initial warm-up.

We executed the benchmarks on a quad-core Intel Core i7-6700HQ CPU at 2.60GHz on Ubuntu version 14.04 (4.3.0-040300rc3-generic) with 16 GB of memory. We use *Sulong* revision *ad56c6f*, which is publicly available at `https://github.com/graalvm/sulong`, that uses LLVM 3.3 (we currently cannot use a newer version due to parser limitations), and the Graal version that will be contained in the GraalVM 0.17 release. When compiling Fortran files to LLVM IR, *Sulong* uses GCC 4.6, the version that is expected to work best with the DragonEgg plugin. When compiling C or Fortran benchmarks for *Sulong* we use the following static optimization parameters to *opt*: *-mem2reg -globalopt -simplifycfg -constprop -instcombine -dse -loop-simplify -reassociate -licm -gvn*. We consider a systematic evaluation of combinations of static and dynamic optimizations on Sulong as future work.

We use *Clang O3* (*-O3* LLVM optimizations) for C, and *GCC O3* (*-O3* GCC optimizations) for Fortran to get a static compilation upper performance boundary. For comparability, *Clang O3* and *GCC O3* use the same LLVM and GCC versions as Sulong. We visualize the peak performance runtime of the benchmarks using box plots. The y-axis shows Sulong's run-time (lower is better) relative to *Clang O3*'s and *GCC O3*'s runtime which is normalized to *1*.

[3] `http://benchmarksgame.alioth.debian.org/`

[4] We used *cloc* to get the lines of code (LOC) without blank lines and comments.

[5] `http://www.netlib.org/benchmark/whetstone.c`

[6] `https://github.com/xxgreg/deltablue/blob/master/deltablue.c`

[7] `http://www.cl.cam.ac.uk/~mr10/Bench.html`

[8] `http://people.csail.mit.edu/smcc/projects/single-file-programs/`

[9] `http://www.polyhedron.com`

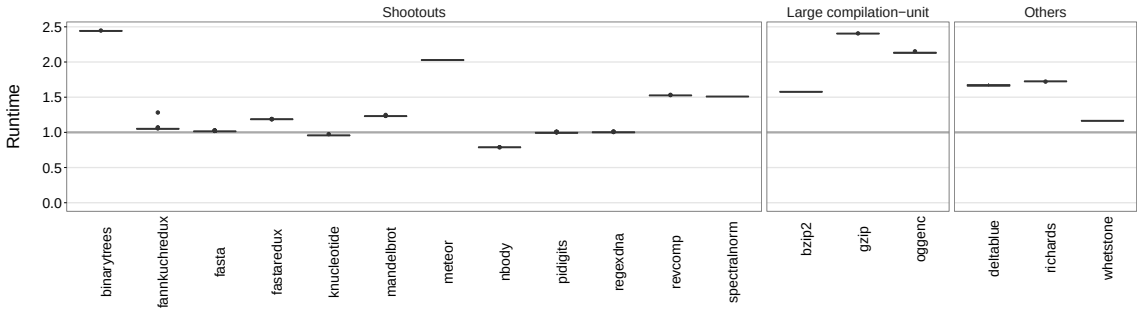

Figure 9. C benchmarks; peak performance (lower is better, relative to *Clang O3*).

6.3 Result

On the C benchmarks (see Figure 9), Sulong's peak performance ranges from being 0.79× faster than Clang (nbody), and being 2.45× slower (binarytrees). On average (geometric mean (Fleming and Wallace 1986)), Sulong is 1.39× slower than Clang. On nbody, Sulong is faster since it can use the SSE sqrt instruction instead of a call to the standard library, and since it can unroll a loop whose number of loop iterations depends on an input parameter to the function. On many benchmarks, Sulong achieves similar performance as Clang O3 (fannkuchredux, fasta, fastaredux, knucleotide, pidigits, regexdna, and whetstone). For most of these benchmarks, Sulong produces similarly efficient code as Clang. However, pidigits and regexdna spend most work in calls to (and in) third-party libraries. Having no overhead on these benchmarks demonstrates that Sulong can efficiently interface with native code. On the remaining C benchmarks (binarytrees, bzip2, deltablue, gzip, meteor, oggenc, revcomp, richards, and spectralnorm), Sulongs performance is between 1.5× and 2.45× slower than Clang O3.

On the Fortran benchmarks (see Figure 10), Sulong's peak performance is between 1.43× (nf) and 4.96× (doduc) slower than the performance of GCC O3 executables. On average, Sulong is 2.63× slower compared to GCC O3. So far, we mainly optimized Sulong for executing C programs, and have not yet looked into optimizing Fortran programs, which explains the larger gap between Sulong and GCC.

Besides missing various micro optimizations, there are three main reasons for the overheads on the C and Fortran benchmarks:

Needless interpreter-level object allocations: Graal implements a partial escape analysis with scalar replacement to optimize or remove object allocations where possible (Stadler et al. 2014). It is critical for performance, that all Java allocations that the LLVM IR interpreter uses in its runtime (i.e., interpreter-level allocations as opposed to user-level allocations) are optimized or removed in compiled code. Unfortunately, we still have situations where this is not the case, and where we either have to adapt data structures in the interpreter or fix problems in Graal's escape analysis.

Truffle's calling convention: Truffle passes function arguments in an Object array and returns the function return value as an Object, so parameters and return values have to be boxed and unboxed. Function inlining usually removes this overhead. However, in benchmarks that stress recursive calls (which can only be inlined up to a certain level) such as binarytrees and richards, the overhead is still significant.

Missing vectorization: Graal cannot produce vectorized code for Sulong, since it does not provide sufficient analyses for accesses to unmanaged memory.

7. Limitations

Sulong can currently execute most small and middle-sized single-threaded C and Fortran programs. We did not concentrate on other languages so far and thus did not implement, for example, LLVM IR exception handling, which is needed to execute C++ programs that use exceptions. Although we did not find any essential problems when executing LLVM IR on the JVM, our current implementations has several limitations:

Unsupported library functions: To achieve better performance and faster startup times, we still use the native (i.e., machine code) standard libraries instead of their bitcode versions. When Sulong is complete and fast enough, we will execute the LLVM IR of the standard libraries with Sulong for which we will only have to substitute system calls. Currently, Sulong does not support creating new processes with `fork`, since a call to `fork` would create a copy of the JVM. Similarly, we currently also do not support `setjmp`/`longjmp`, signal handling, and POSIX `pthreads` for multithreading.

Callbacks from native functions: In terms of native interoperability, our foreign function interface does not support native callbacks yet (Grimmer et al. 2013). For example, we cannot call a native function to which we pass a Truffle AST (e.g., `qsort`) that could be called from the native side. To prevent this case for the standard libraries, we substitute these functions with Java or bitcode equiv-

alents (see above). For third-party libraries we compile such functions to a shared library which we then link.

Manipulation of function return addresses: In Sulong, the memory layout matches that of executables produced by static compilers. One exception is the function return address that executables store in the same stack as data passed to other functions. The Sulong interpreter implicitly uses the Java execution stack when executing functions. This execution stack is different from our data stack that uses unmanaged allocated memory. Thus, we cannot provide support for reading and manipulating function return addresses. However, this also restricts return oriented programming (a security exploit technique, Shacham 2007) since buffer overflows cannot overwrite the return address.

80 bit floats: Most primitive data types in LLVM IR directly map to Java data types. An exception is LLVM IR's 80 bit float type that Clang uses for C's long double data type on the AMD64 architecture. We do not completely support this data type so far due to the implementation effort required to correctly and efficiently implement it using Java primitives.

Inline assembler: Sulong only partially supports inline assembler by constructing a Truffle AST from it and representing the machine registers as Java objects. Still, Sulong cannot execute generated code (such as produced by JITs), for which Sulong would need to interpret the generated machine instructions.

8. Related Work

8.1 Java's Foreign Function Interfaces

Java's standard NFI is JNI (Liang 1999). JNI is a platform independent interface that not only allows calling native functions, but also enables programmers to interact with Java objects and the JVM. However, JNI requires the declaration of *native* Java methods and the implementation of native functions that match a generated header file, which makes JNI complicated to use, especially when a programmer only wants to call native functions. Due to the abstraction overheads, JNI is also slow (Kurzyniec and Sunderam 2001). Previous work showed that the overheads can greatly be reduced by inlining native function calls and by using the same intermediate language for Java and the target low-level language (Stepanian et al. 2005).

An alternative to JNI is Java Native Access[10] (JNA) which is built on top of JNI and provides access to shared native libraries that it dynamically links. Dynamic linking frees the programmer from the burden of writing boilerplate code, but makes calls slower. Efforts to reduce this overhead by generating call stubs using LLVM as a JIT compiler (but still using JNI) can improve performance by 7.84% (Tsai et al. 2013). Besides JNA, also the Java Native Runtime (JNR) is built on top of JNI and provides a user-oriented API to call native functions[11]. Based on the experiences with JNR, a JDK Enhancement Proposal (JEP 191) was drafted that tackles JNI's drawbacks and aims at providing better usability and optimizing calls to native functions (Nutter and Rose 2014). Project Panama, an OpenJDK subproject, works on improving interoperability between the JVM and native functions based on this JEP with the eventual goal to include the changes in the JDK[12].

In our previous work, we introduced the Graal NFI (Grimmer et al. 2013) to call native functions that are dynamically linked. The Graal NFI is fast, since it compiles a call stub to the native function before invoking it the first time, and inlines the call stub when the surrounding Java code is compiled. However, in contrast to JNA and JNR the programmer is responsible for data alignment and handling of unsafe memory, which makes it error-prone and difficult to use (it was designed for native language implementations on top of Truffle). Also, it is only available in the Graal compiler. Jeannie (Hirzel and Grimm 2007) is a language design that allows nesting Java and C code in the same file, which is then compiled down to JNI. Through static checks on syntax and semantics of both languages, it is easy to use and also eliminates writing boilerplate code.

Sulong is an alternative to traditional native function interfaces since it can execute low-level languages directly on the JVM. Sulong does not require writing boilerplate code, and programmers can use Sulong as a Java library to execute native functions. Additionally, Sulong is fast and supports execution of all LLVM languages. However, Sulong requires that the source code of the native function to be called is available. Also, it requires the Graal compiler in order to reach peak performance that is near to the performance of statically compiled code, and to call native functions.

8.2 PyPy

PyPy (Rigo and Pedroni 2006) and its virtual machine construction approach is an alternative to Truffle/Graal's meta-compilation approach (Marr and Ducasse 2015). Both approaches strive to provide a reusable base for dynamic language implementations and also provide language interoperability mechanisms (Barrett et al. 2013, 2015; Grimmer et al. 2015b). In both cases, a language implementer can use high-level languages with automatic memory management for implementing a language. While PyPy uses RPython (a semantic subset of Python, Ancona et al. 2007) for the implementation of its interpreters, Truffle uses Java. PyPy language implementations can be any kind of interpreters, while Truffle implementations are implemented as self-optimizing AST interpreters. With Sulong, we showed how a hybrid bytecode/AST interpreter can be implemented in Truffle.

[10] `https://github.com/java-native-access/jna`

[11] `https://github.com/jnr/jnr-ffi`

[12] `http://openjdk.java.net/projects/panama/`

For an efficient implementation, PyPy uses a translation process to transform the RPython interpreter to low-level code for a target environment (Rigo and Pedroni 2006). This translation process first analyzes the interpreter, annotates it with types, and then consecutively transforms it to lower-level operations. For optimal performance, the translation target is a C interpreter that contains a tracing JIT compiler (Bolz et al. 2009). The tracing JIT is not applied to the user program, but to the interpreter running the user program. Similarly, Truffle compiles ASTs (and not traces) that represent the user program to machine code by using Graal as a dynamic compiler. With Sulong's approach, Graal also supports the compilation of bytecode interpreters and hybrid AST/bytecode interpreters.

8.3 Hybrid Compilation Approaches

Dynamo (Bala et al. 2000) is a dynamic optimization system that re-optimizes an already compiled native instruction stream to exploit dynamic optimizations. Like Sulong, Dynamo profits from static optimizations at compile time and profiling information at run time. In contrast to Sulong, Dynamo supports any kind of native instruction stream and not only those languages supported by LLVM. However, due to the low-level information on the machine code level, Dynamo's approach is limited in the optimizations that it can apply. Finally, Dynamo re-compiles traces while Sulong uses Truffle and Graal to compile function ASTs to machine code.

Previous work also includes a fat binary approach (Nuzman et al. 2013), where a program is distributed as an executable that comprises both the native code and the IR of that program. The program starts execution with the native code, which incurs only low start-up and warm-up costs. A runtime manager samples the execution count of the functions and when exceeding a certain threshold, it adds instrumentation to it. Finally, a repurposed Java compiler compiles the IR of that function to optimized machine code, for which it also uses the profiling feedback of the instrumented function. While Sulong has higher start-up and warm-up costs, it does not require a modified toolchain that is needed to produce fat binaries. Sulong can execute unmodified LLVM IR that is produced by language front ends for many languages.

8.4 Other Truffle Implementations

We previously worked on Truffle/C (Grimmer et al. 2014) and ManagedC (Grimmer et al. 2015a) which are Truffle interpreters for C. Similarly to Sulong, Truffle/C uses unmanaged memory for its allocations. ManagedC uses Java allocations instead of unmanaged memory. The C interpreters provide the same dynamic optimizations that Sulong does. In contrast to the C interpreters, Sulong also uses static optimizations by LLVM to optimize the program before executing it with its LLVM IR interpreter. Unlike the C interpreters, Sulong is not restricted to C but can execute a range of different languages by targeting LLVM IR. Also, the C interpreters do not have to efficiently support unstructured control flow since it is only used in exceptional situations, e.g., in exception handling using `goto`. To efficiently execute LLVM IR (which contains no high-level loop constructs), we use a hybrid bytecode/AST interpreter approach.

9. Conclusion and Future Work

In this paper we presented Sulong, a system to execute low-level languages such as C and Fortran on the JVM. By providing a Truffle LLVM IR interpreter, Sulong can execute all languages that can be translated to LLVM IR. By combining static optimizations with dynamic compilation Sulong can achieve peak performance that is near to the performance of code that is produced by industrial-strength compilers such as GCC and Clang. We demonstrated that Sulong currently runs C code with a peak performance that is in average 1.39× slower than code compiled by Clang O3 and Fortran code 2.63× slower compared to code compiled by GCC O3.

Other Truffle implementations can profit by using Sulong to implement their native function interfaces. JRuby+Truffle (a Truffle implementation of Ruby) already uses Sulong for its C extension support, and FastR (a Truffle implementation of R) provides an option to use Sulong instead of JNI for calling native routines. Due to Sulong's low overhead and Truffle's language interoperability mechanism that supports inlining across language boundaries, we expect that we can improve the performance of these languages when calling native code. In future work, we want to demonstrate this on case studies, and also provide a version of Sulong that only uses managed Java memory to guarantee memory safety for the programs it executes (Rigger et al. 2016).

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 for their support and contributions. We especially thank Roland Schatz for his assistance on performance improvements, Chris Seaton for the implementation of JRuby's C extensions using Sulong, and Mick Jordan for his work on implementing calls to native routines using Sulong. We also thank Edd Barrett, Benoit Daloze, Stefan Marr, and Chris Seaton for their comments which greatly improved the paper. 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

D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed oo languages. In *Proceedings of DLS 2007*, pages 53–64, 2007.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of PLDI '00*, pages 1–12, 2000.

E. Barrett, C. F. Bolz, and L. Tratt. Unipycation: A case study in cross-language tracing. In *Proceedings of VMIL 2013*, pages 31–40, 2013.

E. Barrett, C. F. Bolz, and L. Tratt. Approaches to interpreter composition. *Computer Languages, Systems & Structures*, 44: 199–217, 2015.

E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *ICOOOLPS*, 2016, 2016.

C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of ICOOOLPS 2009*, pages 18–25, 2009.

J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. *ACM SIGOPS OSR*, 36(5):199–210, 2002.

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

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

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 VMIL '13*, pages 1–10, 2013.

A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of Computer Languages*, pages 229–240, 1994.

P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, Mar. 1986.

Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An efficient native function interface for java. In *Proceedings of PPPJ '13*, pages 35–44, 2013.

M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. Trufflec: Dynamic execution of c on a java virtual machine. In *Proceedings of PPPJ '14*, pages 17–26, 2014.

M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe execution of c on a java vm. In *Proceedings of PLAS'15*, PLAS'15, pages 16–27, 2015a.

M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of DLS 2015*, pages 78–90, 2015b.

M. Hirzel and R. Grimm. Jeannie: Granting java native interface developers their wishes. In *Proceedings of OOPSLA '07*, pages 19–38, 2007.

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

U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *ACM Sigplan Notices*, volume 27, pages 32–43, 1992.

D. Kurzyniec and V. Sunderam. Efficient cooperation between java and native codes–jni performance benchmark. In *PDPTA'01*, 2001.

C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *CGO 2004*, pages 75–86, March 2004.

S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. 1999.

S. Marr and S. Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. *ACM SIGPLAN Notices*, 50(10):821–839, 2015.

S. Marr, B. Daloze, and H. Mössenböck. Cross-language compiler benchmarking. In *DLS 2016 (to appear)*, 2016.

K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell. Into the depths of c: elaborating the de facto standards. In *PLDI 2016*, pages 1–15, 2016.

C. O. Nutter and J. Rose. Jep 191: Foreign function interface, 2014. URL `openjdk.java.net/jeps/191`.

D. Nuzman, R. Eres, S. Dyshel, M. Zalmanovici, and J. Castanos. Jit technology with c/c++: feedback-directed dynamic recompilation for statically compiled languages. *TACO*, 10(4):59, 2013.

M. Rigger, M. Grimmer, and H. Mössenböck. Sulong - execution of llvm-based languages on the jvm. In *ICOOOLPS'16*, 2016.

A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *SPLASH 2006*, pages 944–953, 2006.

J. R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *Proceedings of VMIL '09*, pages 2:1–2:11, 2009.

C. Seaton. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. PhD thesis, University of Manchester, 2015.

H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, pages 552–561, 2007.

L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of CGO '14*, pages 165–174, 2014.

L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing r language execution via aggressive speculation. In *DLS 2016 (to appear)*, 2016.

L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining java native calls at runtime. In *Proceedings of VEE '05*, pages 121–131, 2005.

Y.-H. Tsai, I.-W. Wu, I.-C. Liu, and J. J.-J. Shann. Improving performance of jna by using llvm jit compiler. In *ICIS 2013*, pages 483–488, 2013.

T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In *Proceedings of DLS '12*, pages 73–82, 2012.

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 Onward! 2013*, pages 187–204, 2013.