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

JOHANNES KEPLER UNIVERSITY LINZ

ORACLE

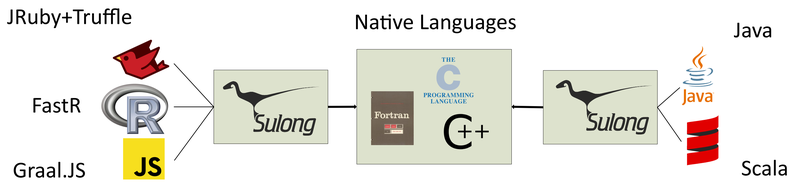
**Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck**

## Motivation

- To execute low-level languages (C, Fortran, C++, …) from JVM languages, one has to ***resort to native interfaces*** such as the Java Native Interface (JNI)
- JNI ***incurs overhead***, makes ***native interoperability complicated***, and ***contradicts Java's memory safety guarantees***
- **Language boundaries** are an obstacle for optimizations; compilers cannot optimize across language boundaries

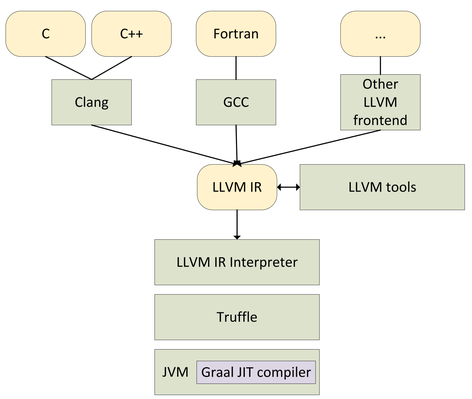
## Our Solution: Execute Native Languages on the JVM

- ***Interpreter for native languages*** on top of the JVM
  - to implement ***dynamic language's native interfaces***
  - to run ***native languages on the JVM***
- Having several languages on the same platform allows ***optimizations across language boundaries***



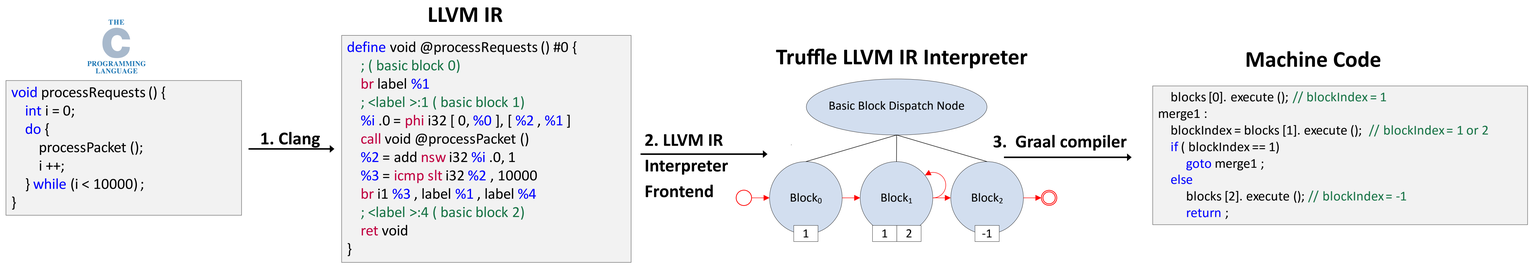

## System Overview: Bringing LLVM and the JVM together

- ***Clang and GCC*** compile ***native languages to LLVM IR***
- ***LLVM IR*** is a machine-specific, low-level intermediate format
- ***LLVM IR interpreter*** bases on the Truffle language implementation framework
- Truffle interpreters are ***Abstract Syntax Tree (AST) interpreters***
- Truffle uses ***the Graal JIT compiler*** for efficient compilation




## Interpretation and Dynamic Compilation of LLVM IR

1. ***Compile C/C++/Fortran Program*** to LLVM IR
2. Execute ***LLVM IR*** with the ***Truffle LLVM IR Interpreter***
3. Compile frequently executed functions to machine code by ***unrolling the interpreter loop***



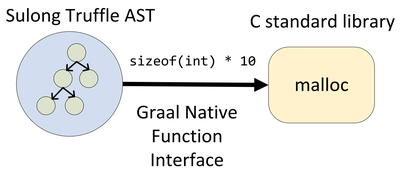

```
void processRequests () {
   int i = 0;
   do {
      processPacket ();
      i ++;
   } while (i < 10000) ;
}
```

```
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
}
```

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

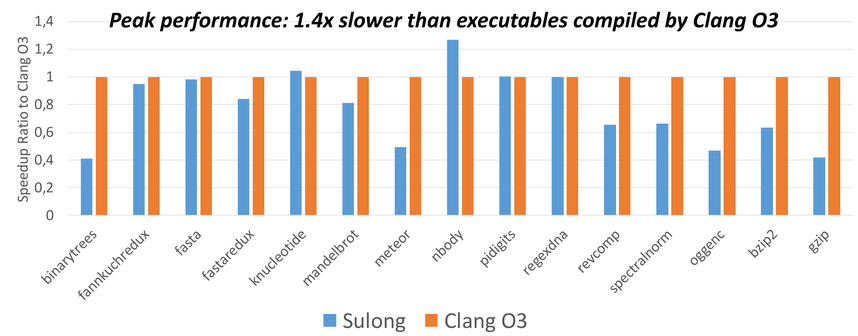
## Native Calls

- ***Graal Native Function Interface*** (NFI) to call native functions
  ***Direct calls*** from compiled Java code to native functions!
- Sulong allocates and accesses ***unmanaged memory to avoid conversion*** and marshalling of data




Grimmer, M., et al. "An efficient native function interface for Java." *Proceedings of PPPJ'13.*

## Performance Evaluation: C Benchmarks

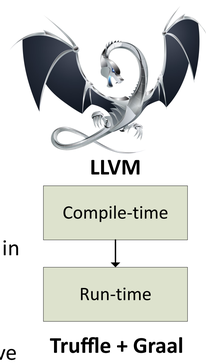



## Static Optimizations

- **Problem**: Java compilers do not optimize accesses to unmanaged memory
- **Solution**: Use LLVM's static optimization tool to promote memory accesses to local variables

## Dynamic Optimizations

- **Problem**: Static compilers cannot react to changes in program behavior
- **Solution**: Profile in the interpreter and exploit profiling data during compilation through speculative optimizations




## Contact

manuel.rigger@jku.at
@RiggerManuel
http://github.com/graalvm/sulong

Sulong

## Acknowledgement

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 contributions. The authors from Johannes Kepler University are funded in part by a research grant from Oracle.