Example code:

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

Documentation:

- **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.
  - 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**
  - **LLVM IR**
  - `define void @processRequests() #0 {
      ; ( basic block 0)
      br label %1
      ; <label >:1 ( basic block 1)
      %i .0 = phi i32 [ ... i32 %i .0, 1
      %3 = icmp slt i32 %2 , 10000
      br i1 %3 , label %1 , label %4
      ; <label >:4 ( basic block 2)
      ret void
    }
  - **C standard library**
  - `malloc`

- **Native Calls**
  - Interoperability with native libraries.
  - Sulong allocates and accesses unmanaged memory to avoid conversion and marshalling of data.

- **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.

- **Performance Evaluation: C Benchmarks**
  - **Peak performance**: 1.4x slower than executables compiled by Clang O3.

- **Contact**
  - [manuel.rigger@jku.at](mailto:manuel.rigger@jku.at)
  - @RiggerManuel

- **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.