Web Assembly Support for Clang REPL

Anubhab Ghosh

Mentors: Vassil Vassilev, Alexander Penev

Recap: WebAssembly

- WebAssembly is a web standard that defines a portable virtual CPU architecture and corresponding binary code format.
  - A browser may either emulate it or opportunistically JIT compile it to native code for maximum performance. This is transparent to the WASM program.
  - By default, it runs in a sandboxed environment with no access to the outside.
    - A WASM code module defines functions that it wants to import/export.
    - The Javascript runtime that loads the WASM module can provide implementation of the imports and call the exported methods.
  - WASM code does not live in the same address space as data (Harvard arch.).
    - In fact, WASM functions are defined with high level control flow preserved and have an opaque runtime representation. We cannot read/write or even check their size.

Main issues and challenges

- All WASM code within a module is read-only!
  - We cannot just write code into memory and jump to it.
  - However, we can create multiple modules!
    - They can share the same data memory passed from JS runtime!
    - Calls across module is possible by linking exports from one module with imports from another. Of course, Such calls have runtime overhead!
  - We can take a snapshot of the current state of the module, make a new module with the new code included and replace the previous one.
    - Linking takes too long.
- Size of the compiled modules can be massive. (currently ~72 MiB ☹)
  - We must ship at least LLVM, Clang and all libraries we want to use.

Approach

- I used Emscripten as the WASM compiler toolchain as it is mature.
  - It has many libraries ported (SDL -> HTMLCanvas, OpenGL -> WebGL).
- We build LLVM, Clang and LLD for WASM and link libclangInterpreter with our own glue interpreter code.
- A little bit of Javascript code passes some basic functions (such as console.log for stdout/stderr)
- We also pass a default virtual file system.
  - It includes headers files provided by both Clang and Emscripten.
- At runtime, on each REPL iteration we call a function with the input code on the WASM module.

Approach

- At runtime, at each iteration we call clang::Interpreter::Parse() to pass our input to clang but don't call clang::Interpreter::Execute().
  - Instead, we take the llvm::Module and generate an object file.
  - We then pass this file to wasm-ld to generate a "Shared library".
  - We can then use Emscripten's dlopen() implementation to load it.
    - It will create a new module, attach the same data memory and resolve the symbols from the previous modules.
- Output (if any) is directly shown in page body with the functions already provided to the module.
- Other side effects can be executed similarly.

Remaining tasks and issues

- Integrate this with xeus-lite.
  - This is a framework that would allow Clang REPL to be used from Jupyter Lite.
  - It's a version of Jupyter that runs entirely in browser with WASM kernels.
- Reduce the size if possible.
  - All the libraries need to be preserved in entirety in the Emscripten main module because we don't know which symbols will be needed at runtime.
    - Our initial compiler WASM module is built as a main module (includes the libraries).
      - Main Emscripten module cannot be stripped. Unused functions cannot be removed.
      - But it includes a large chunk of possibly unused LLVM/Clang code and all the symbols.
      - A possible workaround is to move clang into a separate side module.

DEMO

Thank You

https://wasmclang.argentite.me