WebAssembly Support for clang-repl

Name: Anubhab Ghosh

GitHub Username: argentite

Email: anubhabghosh.me@gmail.com

LLVM Discord Username: argentite#0791

LLVM Discourse Username: argentite

Description

Clang includes libInterpreter, a framework for incrementally JIT compiling and executing C++ code. It is exposed to users through the clang-repl CLI tool. There is also another project xeus-clang-repl that integrates it with the Jupyter xeus framework allowing clang-repl to run on the server and interact with the user using a Jupyter interface inside a browser. This allows an user to use clang-repl without any kind of environment setup.

However an obvious problem of this approach is that it shifts all the computational work to a server from the user's device. A potential solution to this problem is the use of Web Assembly (WASM). It allows programs compiled to an intermediate bytecode representation to be executed (inside a sandbox) by a Javascript runtime at closer to native speeds compared to regular Javascript. This sandbox by default has no access to outside environments (such as syscalls) and can only communicate using symbols imported to it or exported from it.

WASM code can be generated ahead-of-time from many general purpose programming languages including C++ using Clang and LLVM. However existing C++ code makes some assumptions about the runtime environment such as the presence of standard libraries and to some extent an Unix-like operating system with file system, sockets etc. WASI is a standard "modular system interface" that defines a set of APIs (similar to sys calls) that a WASM program can import to get access to such an Unix-like environment allowing standard libraries to be built on top of it. Emscripten, alongside providing a custom compiler pipeline based on Clang/LLVM for WASM also provides a standard library implementation that interacts with Javascript code generated by it for unrestricted access to the outside environment. However it is less flexible than WASI.

But our goal is to run libInterpreter/clang-repl as a WASM module and for it to generate WASM output so that it can be executed by the same Javascript runtime (such as inside a browser). From an AOT compiler point of view, WASM is just another CPU architecture that is already well supported by LLVM. However, for a JIT compiler, it is significantly more complex because, for security reasons and to allow the JS runtime to possibly compile WASM to native code, WASM uses an abstract virtual stack machine.

The abstract WASM CPU uses a Harvard architecture. Code is represented as a set of variably sized functions, identified uniquely by an integer with associated bytecode while data uses a linear address space. This means there is no intersection between code and data memory and generated JIT code residing in data memory cannot simply be marked as executable pages.

In fact, as a sandboxed architecture, there is possibly no practical way to add additional code to a running WASM module instance. Instead the interpreter must create a new module for the newly generated code. The new module has to be linked with existing code which can be done statically or dynamically. In both cases, a single unified memory can be used as shared data address space.

When “statically” linking, the existing code can be copied into the new module and the data memory can be shared among them. Now the new module can replace the existing module with help from Javascript code. This obviously has a high overhead for each iteration of the interpreter.

When “dynamically” linking, reference to functions can be exported by modules to the Javascript code and passed along to another module. This is faster for compilation but inter module calls have higher overhead.

Mentors

- Vassil Vassilev
- Alexander Penev

Size of Project

Large

Expected Deliverables

- A feasibility study of whether it is possible to load and execute the WASM emitted by clang inside the browser without server side support.
- Enable support for generating web assembly output in libInterpreter.
- Port clang-repl to web assembly to run inside a JS engine environment.
- Load and execute the compiled WASM modules.
- Integrate the work in xeus-clang-repl.

Project Timeline

Time	Task	Deliverable
Week 1	Define scope of work	A short technical report
Week 2	Produce WASM code from the interpreter (outside a Javascript environment)	Display the generated code to the user
Week 3-4	Implement the environment required for Clang/LLVM to run in WASM (possibly stub functions in the Support library) and run libInterpreter inside a JS engine	libInterpreter can parse user input and display the generated WASM inside of a JS/WASM engine
Week 5-6	Produce separate WASM modules for each interpreter input and execute them	User can run independent units of code in the REPL without interdependence inside a JS engine
Week 7-10	Link together the modules through Javascript	User can run code dependent on previous input code
Week 11	Integration with Jupyterlite through xeus-lite or otherwise	User can run interpreted C++ code in Jupyter
Week 12	Write instructions for using the kernel	Documentation in the form of a wiki page/post

Timeframe of Participation

29th May to 28th August 2023, according to the GSoC timeline.

Personal Details

- I am a final year Computer Science and Engineering student at Indian Institute of Information Technology, Kalyani. I am interested in compiler design and development. I have taken two courses in compiler design and automata theory.
- I have had an internship under INRIA, Paris where I worked on an OCaml lex-yacc based tool that parsed (incomplete) portions of C code extracted from git diffs (of the Linux kernel source) and analyzed them for multiple occurrences of similar changes. I also wrote another tool to approximately convert python code to a C-like syntax to be used by the former parser.
- I have had another internship under INRIA where I performed some experimental modifications on the Linux kernel scheduler introducing an atomic flag to avoid race conditions when multiple processes wake up at the same time causing them to end up at the run queue of the same core instead of other free ones.

Have you had any prior contributions to LLVM? If yes, please provide links to these contributions.

I have previously participated in Google Summer of Code 2022 working on Shared Memory Based JITLink Memory Manager.

Technical Skills

- C/C++
- GNU Debugger
- Git
- LLVM/Clang

Availability & Other Commitments

I will possibly have classes during the month of August.