Adding BPF as new JITLink Backend

Google Summer of Code 2023 Proposal.

Takemaru Kadoi(diohabara@gmail.com)

Overview of this project

- **Project size: Large**
- **Difficulty: Medium**
- **Confirmed Mentor: Vassil Vassilev, Stefan Gränitz, Lang Hames**

LLVM is an open-source module reusable compiler and toolchain technologies[1]. Various modern languages born in the 2010s adopt LLVM as its backend, such as Julia, Swift, and Rust.LLVM has two ways of compiling code. AOT(**A**head **O**f **T**ime) and JIT(**J**ust **I**n **T**ime).AOT is a compilation technique that compiles the code before the runtime[2], while JIT is a form of dynamic compilation that compiles code during the runtime[3].

JITLink is LLVM's relatively new JIT linker API – the low-level API that transforms compiler output (relocatable object files) into ready-to-execute bytes in memory[4]. JiTLink is aiming for the following goals.

- Cross-process and cross-architecture linking of single relocatable objects into a target executor process.
- Support for all object format features.
- Open linker data structures (LinkGraph) and pass them to the system.

JITLink has missing targets such as PowerPC and cBPF/eBPF in ELF and MachO. Below is the current implementation status(not including loongarch)[5].

	ELF(Linux)	MachO(macOS)	COFF(Windows)
i386	o	x	x
x86-64	o	o	o
arm32	x	x	x
arm64	o	o	x
RISC-V	o	x	x

I aim to implement these formats/architectures in this project.

	ELF(Linux)
cBPF	o
eBPF	o

BPF originally stood for Berkeley Packet Filter[6]. eBPF(**e**xtended **BPF**) can do more than packet filtering, which differs from its original version, cBPF(**c**lassical **BPF**). As the LLVM document says[7], I will mention BPF as eBPF in this document too.

[1] https://llvm.org/
[2] https://en.wikipedia.org/wiki/Ahead-of-time_compilation
[3] https://en.wikipedia.org/wiki/Just-in-time_compilation
[4] https://llvm.org/docs/JITLink.html
[5] https://github.com/llvm/llvm-project/tree/main/llvm/include/llvm/ExecutionEngine/JITLink
[6] https://ebpf.io/what-is-ebpf/
[7] https://llvm.org/docs/CodeGenerator.html#the-extended-berkeley-packet-filter-ebpf-backend

BPF is an innovative RISC register machine with 11 64-bit registers[8]. It has its "origins in the Linux kernel that can run sandboxed programs in a privileged context such as the operating system kernel. It is used to safely and efficiently extend the kernel's capabilities without requiring to change kernel source code or load kernel modules." [9]

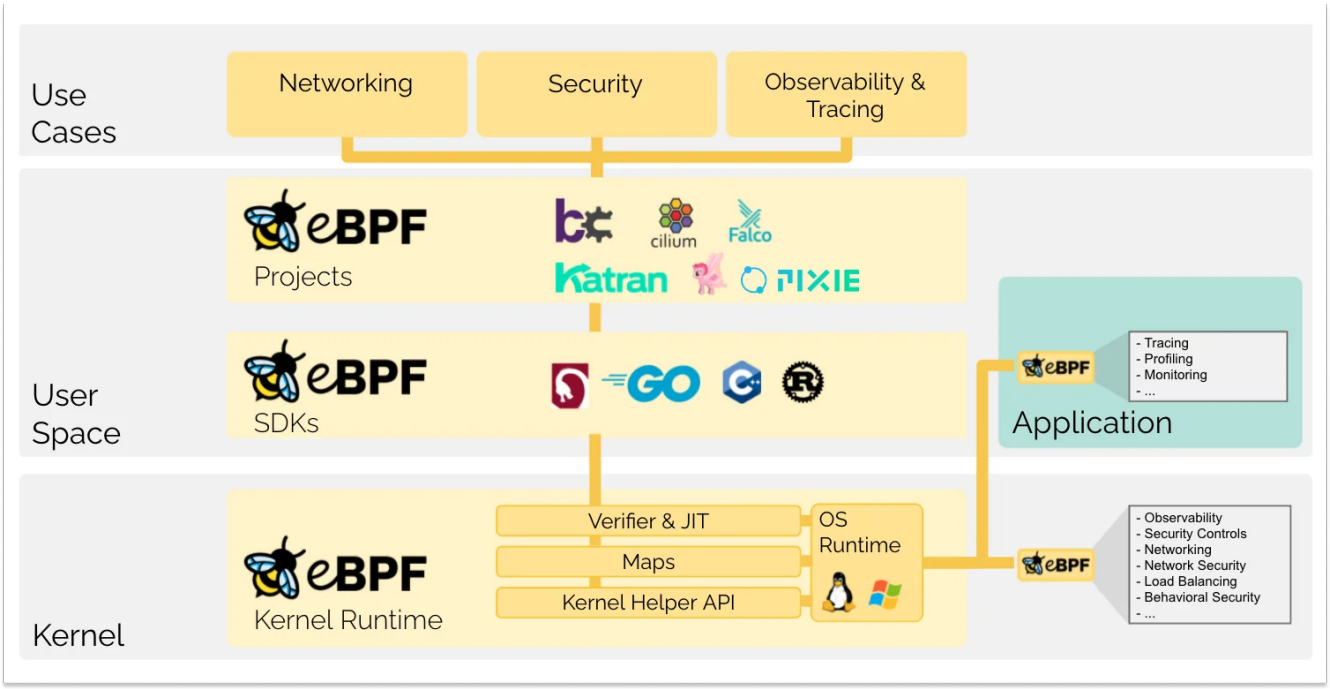

Figure 1: bpf

The implementation of JIT linking of BPF is significant as it achieves incremental compilation, a process that optimizes software performance and network processing[ˆbpf-google1][10][11]. This technology also allows dynamic updates and modifications to be made without interrupting services. Moreover, the popularity of BPF among big tech companies reflects its maturity as a trusted and proven technology that can provide substantial advantages to developers and organizations operating in complex systems[12].

We can compile it but cannot link it for now. If JITLink of BPF becomes available, incremental compilation will be possible and BPF will be available in interactive environments such as clang-repl.

```
$ cat bpf.c # sample C program
int func() { return 0; }
$ clang -target bpf -S -o bpf.s bpf.c # compile C program into BPF assembly
$ llvm-jitlink bpf.o --entry func # cannot link it
llvm-jitlink error: Unsupported target machine architecture in ELF object bpf.o
```

Optional background

clang-repl[13] is a command-line tool that provides an interactive environment for experimenting with C and C++ code. It allows users to enter code snippets and immediately see the results of running them, without needing to compile and link a full program. This can be very useful for quickly testing out small code changes or testing out new language features.

Incorporating LLVM JITLink into clang-repl is significant because it allows the REPL to link on the fly and execute code snippets as they are entered, rather than relying on static linking or separate compilation. This can provide

[8] https://github.com/torvalds/linux/blob/v4.20/include/uapi/linux/bpf.h#L45
[9] https://ebpf.io/what-is-ebpf/
[10] https://legacy.netdevconf.info/0x14/session.html?talk-replacing-HTB-with-EDT-and-BPF
[11] https://lpc.events/event/11/contributions/954/
[12] https://ebpf.io/
[13] https://clang.llvm.org/docs/ClangRepl.html

faster feedback and a more interactive development experience for users since they can see the results of their code changes immediately.

In addition, LLVM JITLink provides a high-performance and flexible linking infrastructure that can be used in a variety of contexts, not just within the clang-repl. This makes it a valuable addition to the LLVM ecosystem as a whole.

Objectives

I will discuss what should be done to achieve the goal. First, we need user space executors that include VM and verifier for BPF as weliveindetail kindly pointed out[^discourse-vm]. There should be several approaches. One approach is the BPF Verifier Harness. It runs in user space independently of any locally running kernel, which opens the door to testing BPF programs across different kernel versions[^ebpf-harness]. [^discourse-vm]: https://discourse.llvm.org/t/jitlink-new-backends/68223/14?u=diohabara [^ebpf-harness]: https://blog.trailofbits.com/2023/01/19/ebpf-verifier-harness/

Expected deliverables are below.

- Set up an environment for executing and verifying the BPF program
- Teach the VM to accept incremental updates for a program
- Implement basic linker implementation that loads an ELF object file and supports relocations[14]
- Implement exception registration and thread-local storage(TLS)
- Incorporate BPF backend into clang-repl
- Add tests[15]

Roadmap

The following roadmap use **MUST** and **SHOULD** as defined in RFC 2119[16].

- **MUST**: Set up user-space VM for BPF
  - BPF needs VM to interpret the instructions and executes them on our physical CPUs
  - We cannot use the VM that Linux-kernel provides for our purpose, so we need to find an alternative
- **MUST**: Set up basic linker implementation that loads elf object file and iterate over relocation sections
- **MUST**: Implement local relocations
  - Definitions for BPF relocations for LLVM 16 can be found here[17]
  - `R_BPF_NONE`, `R_BPF_64_64`, `R_BPF_64_ABS64`, `R_BPF_64_ABS32`, `R_BPF_64_NODYLD32`, `R_BPF_64_32`
- **MUST**: Implement global offset table relocation
- **MUST**: Implement thread local storage relocation edges
- **SHOULD**: Target BPF in clang-repl
- **SHOULD**: See BPF incrementally compiled in clang-repl

Timeline

Week	Deliverables	Description
Week0, -May/28	Development setup & Familiarizing myself with Linker and BPF	Determine what VM to use for this project & Familiarize myself with project concepts
Week1-2, May/29-June/11	Minimal test cases & project design	Design the project structure & Start implementing using initial commits as references[18] [19]
Week3-4, June/12-June/25	Testing environment for BPF & Loading an ELF object	Prepare the environment to work when BPF is implemented

[14] https://github.com/llvm/llvm-project/blob/release/16.x/llvm/include/llvm/BinaryFormat/ELFRelocs/BPF.def
[15] https://github.com/llvm/llvm-project/tree/main/llvm/test/ExecutionEngine/JITLink
[16] https://www.rfc-editor.org/rfc/rfc2119
[17] https://github.com/llvm/llvm-project/blob/release/16.x/llvm/include/llvm/BinaryFormat/ELFRelocs/BPF.def
[18] https://github.com/llvm/llvm-project/commit/29fe204b4e87dcd78bebd40df512e8017dffea8f
[19] https://github.com/llvm/llvm-project/commit/2ed91da0f1f312aedcfa01786fe991e33c6cf1fe

Week	Deliverables	Description
Week5-6, June/26-July/03	Implement `R_BPF_NONE`, `R_BPF_64_64`, `R_BPF_64_ABS64`, `R_BPF_64_ABS32`, `R_BPF_64_NODYLD32`, `R_BPF_64_32` & Linking simple C programs	Follow this document[20] & Implement relocations
Week7-8, July/10-July/23	More test cases & Fix bugs & Linking complex C programs	Identify bugs in the implementation and make the implementation more robust & Midterm evaluation deadline (July/14)
Week9-10, July/24-August/06	Linking multi-thread C programs	Aim to realize multi-threaded BPF programs
Week11-12, August/07-August/20	Incorporate BPF into clang-repl & Incremental compilation in clang-repl	Start to target BPF into clang-repl
Week13, August/21-August/27	Fix bugs & Prepare for final evaluation like documenting	Final week: GSoC contributors submit their final work product and their final mentor evaluation

About Me

Personal Information

- Name: Takemaru Kadoi
- Email: diohabara@gmail.com
- LinkedIn: https://www.linkedin.com/in/takemaru-kadoi/
- GitHub: diohabara
- Discourse on LLVM: diohabara
- Discord on LLVM: diohabara
- Residency: CST(UTC - 06:00)
- Availability over the project
  - Up to 40 hours per week during `2023/05/13-2023/08/20`
  - No courses, no internships
- Operating systems and hardware
  - Linux(amd64), macOS(AArch64)

Why Me?

There are mainly three reasons I am fit for this project, technical skills, soft skills, and personal motivations.

I have a strong background in compilers and low-level programming, and I am highly motivated to contribute to the LLVM JIT project. I have experience with a C compiler targeting x86-64 written in C, which taught me about testing, virtualization, and the whole pipeline of language[21]. I also have experience with a toy compiler targeting LLVM with JIT, which taught me about LLVM IR, optimization, JIT, and how to write a simple language using LLVM in C++[ˆllvm-kaleidoscope][22]. In addition, I have knowledge of other IR and am currently researching a decompiler of Python. This knowledge, analyzing IR and its behavior, can be applicable to LLVM IR. I also have experience with assembly through CTF and coursework. For example, I analyzed binary code in a competition presented by NSA, which demonstrates my ability to debug code with persistence[23]. I have taken several courses related to low-level programming, including Compiler, Hardware Architecture, and Operating Systems at the undergraduate level and Compiler and Binary Analysis at the graduate level.

In addition to my technical skills, I have strong soft skills. I have worked at several companies and learned the way to write good documents. I always care about leaving reports in written format. I am currently leading the

[20] https://www.kernel.org/doc/html/latest/bpf/llvm_reloc.html
[21] https://github.com/diohabara/ccc
[22] https://github.com/diohabara/Kaleidoscope
[23] https://github.com/diohabara/nsa-codebreaker-challenge2022

Computer Security Group at my university[24] as an officer, so I know how to communicate effectively.

Contributing to the LLVM JIT project is an excellent opportunity for me to apply my skills and knowledge to a widely used and important project. I believe that my experience and motivation will enable me to make a valuable contribution to the project. I love the technical stack related to programming languages and dedicate my summer to this project. Some commit to open source projects is a compelling example[ˆgccrs1][25].

Special Thanks

- Stefan's introduction to this year's GSoC
  - https://weliveindetail.github.io/blog/post/2023/03/20/llvm-gsoc-2023.html#jitlink-new-backends
- Proposals from past year participants. They were very informative.
  - https://compiler-research.org/assets/docs/Anubhab_Ghosh_Proposal_2022.pdf
  - https://compiler-research.org/assets/docs/Sunho_Kim_Proposal_2022.pdf
- Final reports. So were they.
  - https://gist.github.com/sunho/36d6400a00498507bc2cae0b9d5c15a3

[24] https://www.linkedin.com/company/utdcsg
[25] https://github.com/Rust-GCC/gccrs/commit/66832f312a7db436110a3b08ff51eac349d5fdbf