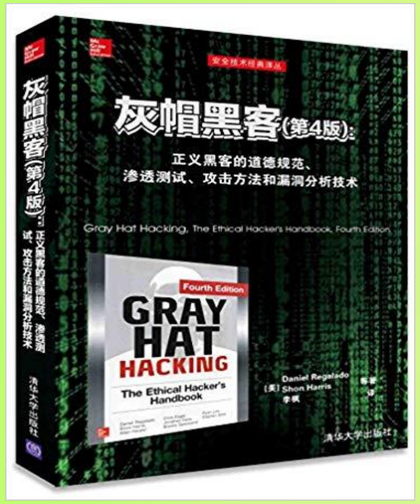
Programming eBPF with Java

Feng Li (李枫)

Indie developer

hkli2013@126.com

Who am I



An indie developer from China:

- **The main translator of the book «Gray Hat Hacking The Ethical Hacker's Handbook, Fourth Edition» (ISBN：9787302428671) & «Linux Hardening in Hostile Networks, First Edition» (ISBN: 9787115544384).**
- **Pure software development for ~15 years(~11 years on Mobile dev).**
- **Actively participating Open Source Communities:**
  https://github.com/XianBeiTuoBaFeng2015/MySlides/
  https://github.com/XianBeiTuoBaFeng2015/MySlides/tree/master/LTS/
- **Recently, focus on infrastructure of Cloud/Edge Computing, AI, IoT, Programming Languages & Runtimes, Network, Virtualization, RISC-V, EDA, 5G/6G…**

Agenda

I. Background

- An overview of xBPF
- Testbeds

II. Project hello-ebpf

- Overview
- Architecture and design
- Internals
- Write eBPF programs in pure Java

III. Discussion on GraalVM-based eBPF development

- uBPF on GraalVM
- Natively support eBPF in GraalVM?
- A customized GraalVM
- A new GraalVM-based DSL for eBPF?

IV. Wrap-up

I. Background

1) An overview of xBPF

1.1 eBPF

1.1.1 Overview

- **https://en.wikipedia.org/wiki/Berkeley_Packet_Filter**

  eBPF [edit]

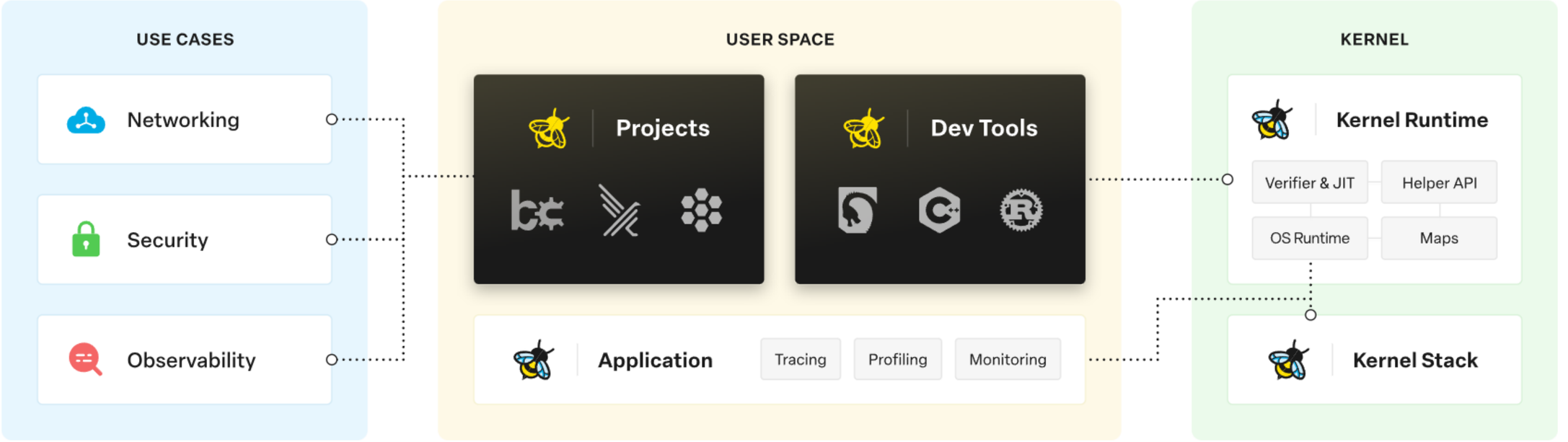
  Main article: eBPF

  Since version 3.18, the Linux kernel includes an extended BPF virtual machine with ten 64-bit registers, termed eBPF. It can be used for non-networking purposes, such as for attaching eBPF programs to various tracepoints.[14][15][16] Since kernel version 3.19, eBPF filters can be attached to sockets,[17][18] and, since kernel version 4.1, to traffic control classifiers for the ingress and egress networking data path.[19][20] The original and obsolete version has been retroactively renamed to *classic BPF* (*cBPF*). Nowadays, the Linux kernel runs eBPF only and loaded cBPF bytecode is transparently translated into an eBPF representation in the kernel before program execution.[21] All bytecode is verified before running to prevent denial-of-service attacks. Until Linux 5.3, the verifier prohibited the use of loops, to prevent potentially unbounded execution times; loops with bounded execution time are now permitted in more recent kernels.[22]

- **https://en.wikipedia.org/wiki/EBPF**
- **aims at being a universal in-kernel virtual machine**
- **dynamically programs the Kernel for efficient networking, observability, tracing, security, and more**
- **drives the User Space and Kernel Space repartition**
- **nearly every subsystem of Kernel is or will be effected by eBPF. especially for network and security**
- **for more details, you may refer to our previous talks and the LTS slides "eBPF: the Past, Present, and Future“ (https://github.com/XianBeiTuoBaFeng2015/MySlides/blob/master/LTS/eBPF--Past%2C%20Present%2C%20and%20Future__20220922p.pdf)**

I. Background

How it works



Source: https://ebpf.io/

I. Background

1.1.2 uBPF (userspace BPF)

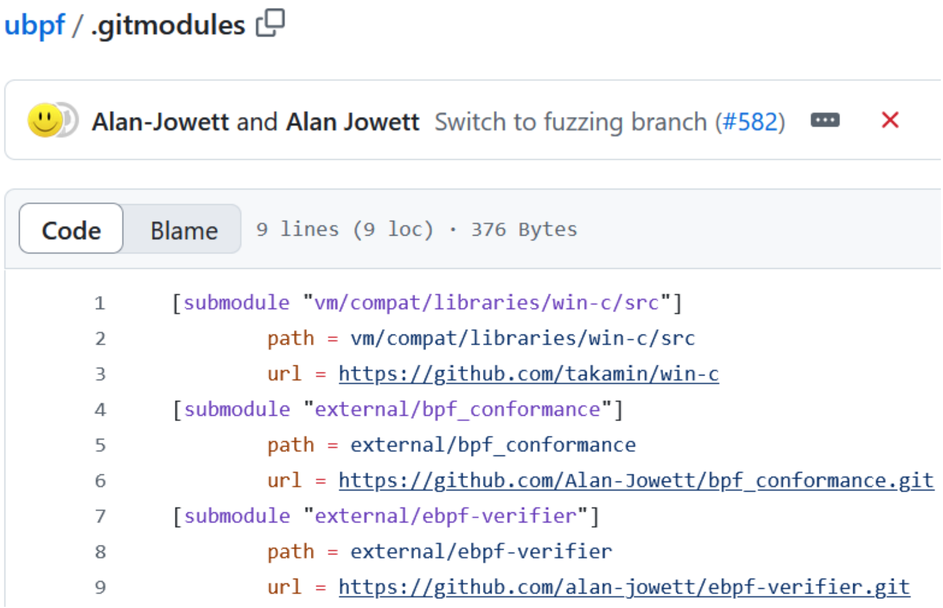
- Another user-mode interpreter is *uBPF*, which supports JIT and eBPF (without cBPF). Its code has been reused to provide eBPF support in non-Linux systems.[6] Microsoft's *eBPF on Windows* builds on uBPF and the PREVAIL formal verifier.[7] *rBPF*, a Rust rewrite of uBPF, is used by the Solana blockchain platform as the execution engine.[8]

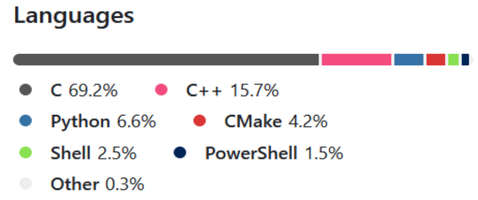
Source: https://en.wikipedia.org/wiki/Berkeley_Packet_Filter

- **https://stackoverflow.com/questions/65904948/why-is-having-an-userspace-version-of-ebpf-interesting: mainly applied to Networking, Blockchain and more.**

- **https://github.com/iovisor/ubpf**

This project includes an eBPF assembler, disassembler, interpreter (for all platforms), and JIT compiler (for x86-64 and Arm64 targets).





I. Background

1.1.3 xBPF

- From our point of view, **xBPF** stands for **eBPF/uBPF** and all of their **derivatives**, together with the technologies that base on them.
- For more details, you may refer to our previous talk **"Revisiting the eBPF-centric new approach for Hyper-Converged Infrastructure & Edge Computing"** at K+ Summit 2021(Shanghai) and the follow-ups.

I. Background

2) Testbeds

2.1 X86

Intel NUC X15

- 

LAPAC71H(32GB DDR5) with Fedora 41(Linux Kernel 6.12.6).

```
[mydev11@koonuc15x-1 /]$ gcc --version
gcc (GCC) 14.2.1 20240912 (Red Hat 14.2.1-3)
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

[mydev11@koonuc15x-1 /]$ clang --version
clang version 19.1.5 (Fedora 19.1.5-1.fc41)
Target: x86_64-redhat-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
Configuration file: /etc/clang/x86_64-redhat-linux-gnu-clang.cfg
[mydev11@koonuc15x-1 /]$
```

```
[mydev11@koonuc15x-1 boot]$ cat config-6.12.6-200.fc41.x86_64 |grep BPF
CONFIG_BPF=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_ARCH_WANT_DEFAULT_BPF_JIT=y
# BPF subsystem
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT=y
CONFIG_BPF_JIT_ALWAYS_ON=y
CONFIG_BPF_JIT_DEFAULT_ON=y
CONFIG_BPF_UNPRIV_DEFAULT_OFF=y
CONFIG_BPF_PRELOAD=y
CONFIG_BPF_PRELOAD_UMD=m
CONFIG_BPF_LSM=y
# end of BPF subsystem
CONFIG_CGROUP_BPF=y
CONFIG_IPV6_SEG6_BPF=y
CONFIG_NETFILTER_BPF_LINK=y
CONFIG_NETFILTER_XT_MATCH_BPF=m
CONFIG_NET_CLS_BPF=m
CONFIG_NET_ACT_BPF=m
CONFIG_BPF_STREAM_PARSER=y
CONFIG_LWTUNNEL_BPF=y
CONFIG_BPF_LIRC_MODE2=y
# HID-BPF support
CONFIG_HID_BPF=y
# end of HID-BPF support
CONFIG_BPF_EVENTS=y
CONFIG_TEST_BPF=m
[mydev11@koonuc15x-1 boot]$
```

```
[mydev11@koonuc15x-1 boot]$ cat config-6.12.6-200.fc41.x86_64 |grep XDP
CONFIG_XDP_SOCKETS=y
CONFIG_XDP_SOCKETS_DIAG=m
CONFIG_SENSORS_XDP710=m
CONFIG_SENSORS_XDPE152=m
# CONFIG_SENSORS_XDPE122 is not set
[mydev11@koonuc15x-1 boot]$
```

II. Project hello-ebpf

Johannes Bechberger
mostlynerdless.de
OpenJDK Developer, SAP
Creator of hello-ebpf

1) Overview

- https://github.com/parttimenerd/hello-ebpf/

Languages

Java 98.5% Other 1.5%

Hello eBPF world! Hello Java world! Let's discover eBPF together and write Java user-land library along the way.

There are user land libraries for eBPF that allow you to write eBPF applications in C++, Rust, Go, Python and even Lua. But there are none for Java, which is a pity. So... I decided to write my own, which allows you to write eBPF programs directly in Java.

This is still in the early stages, but you can already use it for developing small tools and more coming in the future.

Prerequisites

These might change in the future, but for now, you need the following:

Either a Linux machine with the following:

- Linux 64-bit (or a VM)
- Java 22 or later
- libbpf and bpf-tool
  - e.g. `apt install libbpf-dev linux-tools-common linux-tools-$(uname -r)` on Ubuntu
- root privileges (for executing the eBPF programs)

- On Mac OS, you can use the Lima VM (or use the `hello-ebpf.yaml` file as a guide to install the prerequisites):

```
limactl start hello-ebpf.yaml --mount-writable
limactl shell hello-ebpf sudo bin/install.sh
limactl shell hello-ebpf

# You'll need to be root for most of the examples
sudo -s PATH=$PATH
```

The scheduler examples require a patched 6.11 kernel with the scheduler extensions, you can get it from here. You might also be able to run CachyOS and install a patched kernel from there.

https://cachyos.org/

Blazingly Fast & Customizable Linux distribution

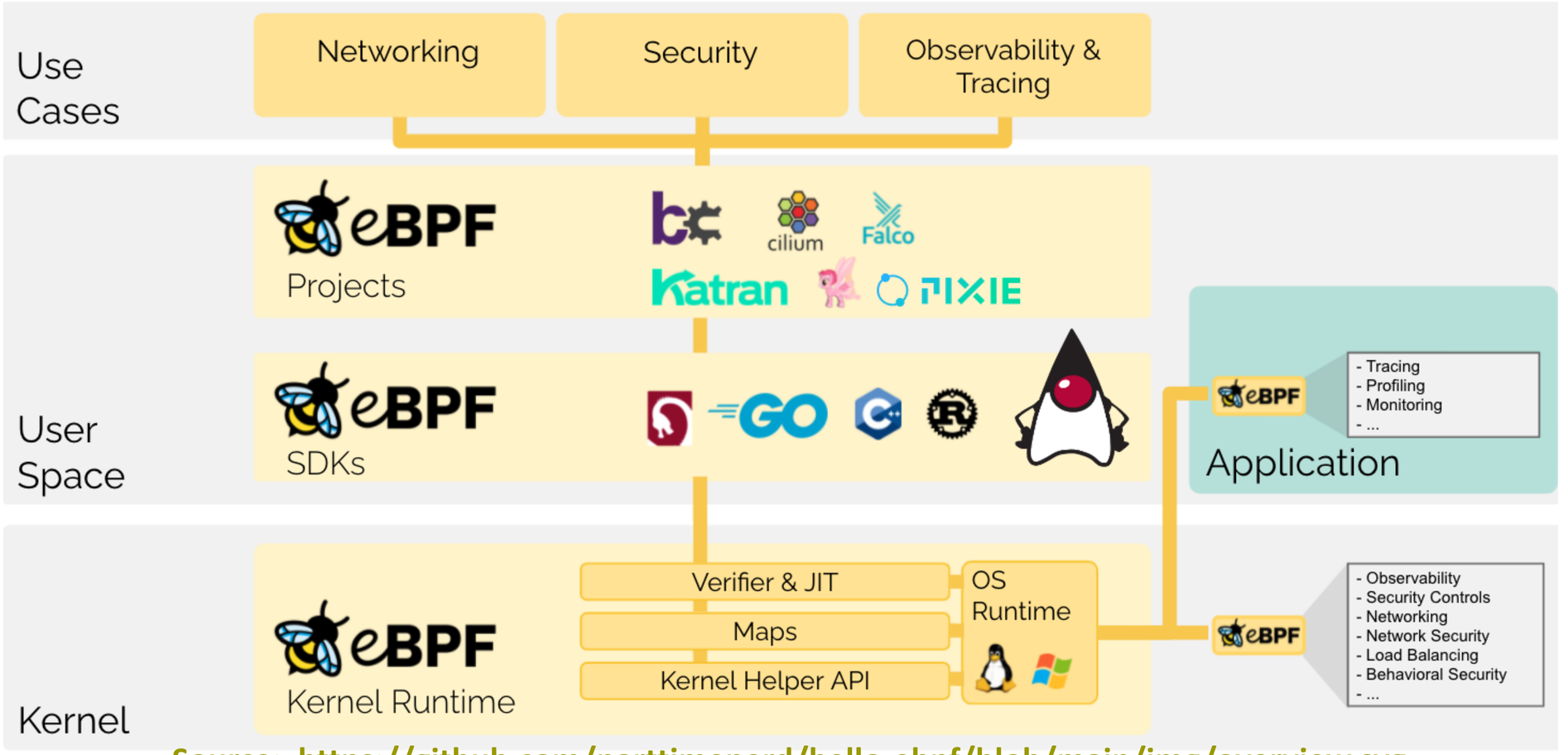
https://launchpad.net/~arighi/+archive/ubuntu/sched-ext-unstable

II. Project hello-ebpf

Goals

- Provide a library (and documentation) for Java developers to explore eBPF and write their own eBPF programs, like firewalls, directly in Java, using the libbpf under the hood.

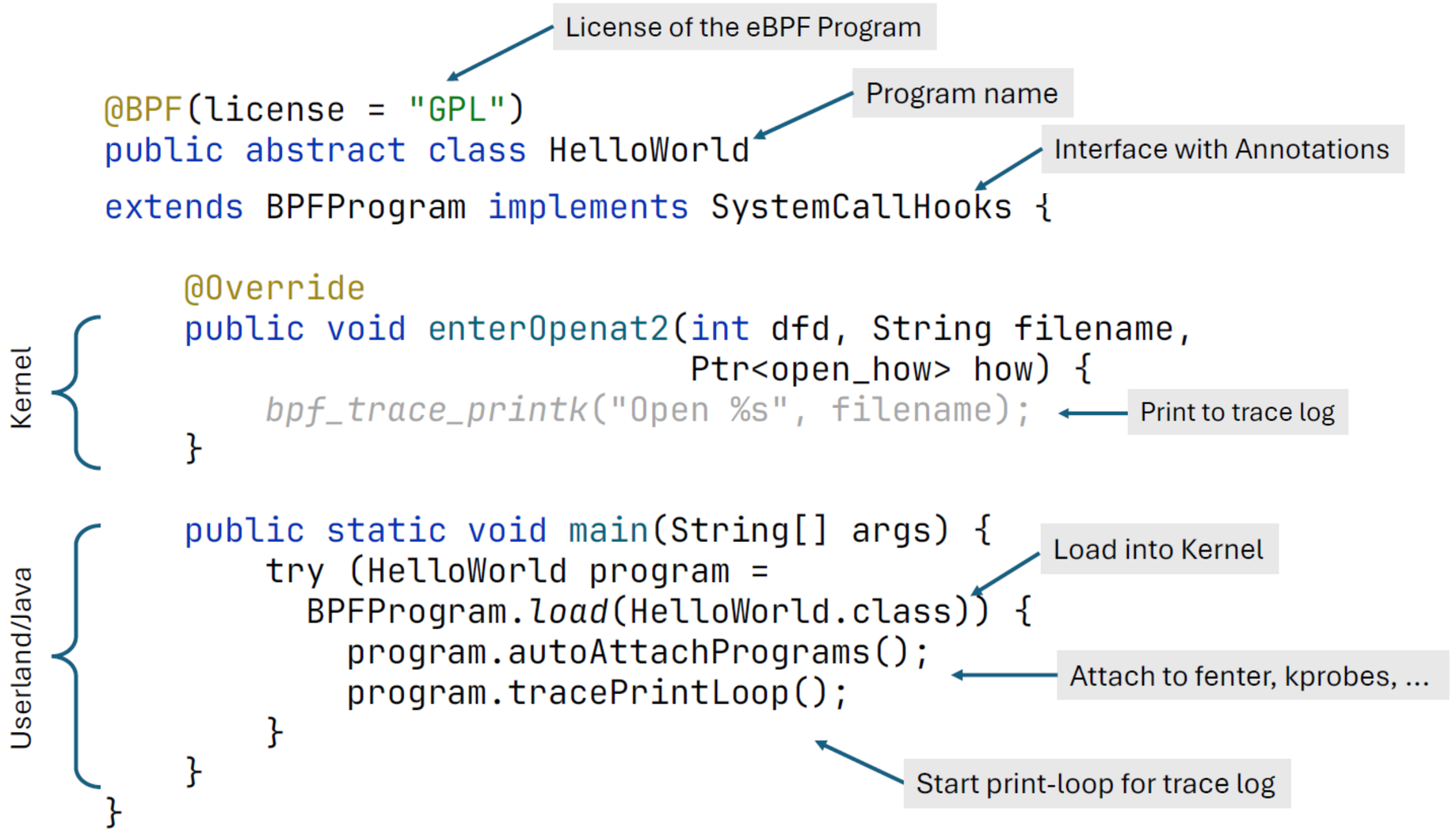
The goal is neither to replace existing eBPF libraries nor to provide a higher abstractions.



Source: https://github.com/parttimenerd/hello-ebpf/blob/main/img/overview.svg

II. Project hello-ebpf

HelloWorld




```
@BPF(license = "GPL")
public abstract class HelloWorld
extends BPFProgram implements SystemCallHooks {

    @Override
    public void enterOpenat2(int dfd, String filename,
                             Ptr<open_how> how) {
        bpf_trace_printk("Open %s", filename);
    }

    public static void main(String[] args) {
        try (HelloWorld program =
          BPFProgram.load(HelloWorld.class)) {
            program.autoAttachPrograms();
            program.tracePrintLoop();
        }
    }
}
```

Source: “hello-ebpf: Writing eBPF programs directly in Java”, Johannes Bechberger, LPC 2024.

II. Project hello-ebpf

Build from src

- JDK & Samples

```
[mydev11@koonuc15x-1 hello-ebpf-main]$ java --version
openjdk 21.0.5 2024-10-15
OpenJDK Runtime Environment (Red_Hat-21.0.5.0.11-1) (build 21.0.5+11)
OpenJDK 64-Bit Server VM (Red_Hat-21.0.5.0.11-1) (build 21.0.5+11, mixed mode, sharing)
[mydev11@koonuc15x-1 hello-ebpf-main]$
[mydev11@koonuc15x-1 hello-ebpf-main]$ export JAVA_HOME=/opt/MyWorkSpace/DevSW/Java/JDK/GraalVM/CE/Java24/v24.2.0-dev-20241213
[mydev11@koonuc15x-1 hello-ebpf-main]$ export PATH=$JAVA_HOME/bin:$PATH
[mydev11@koonuc15x-1 hello-ebpf-main]$
[mydev11@koonuc15x-1 hello-ebpf-main]$ which java
/opt/MyWorkSpace/DevSW/Java/JDK/GraalVM/CE/Java24/v24.2.0-dev-20241213/bin/java
[mydev11@koonuc15x-1 hello-ebpf-main]$ java --version
openjdk 24 2025-03-18
OpenJDK Runtime Environment GraalVM CE 24-dev+27.1 (build 24+27-jvmci-b01)
OpenJDK 64-Bit Server VM GraalVM CE 24-dev+27.1 (build 24+27-jvmci-b01, mixed mode, sharing)
[mydev11@koonuc15x-1 hello-ebpf-main]$
```

```
[mydev11@koonuc15x-1 hello-ebpf-main]$ ./run.sh
Usage: ./run.sh <sample>
Available samples:
BasePacketParser                - Parse helpers for IP packets.
CGroupBlockHTTPEgress           - Block all user processes from using HTTP
Firewall                        - FirewallSpring                        - A spring boot based front-end for the Firewall
HashMapSample                   - Print the number of openat syscalls per process, using a hash map to count them
HelloWorld                      - Hello, World from BPF
LogOpenAt2Calls                 - Logs all openat2 calls
MinimalScheduler                - PacketLogger                          - TC and XDP based packet logger, capturing incoming
 and outgoing packets
RingSample                      - Ring buffer sample that traces the openat2 syscall and prints the filename, process name
, and PID
SampleScheduler                 - TCDropEveryThirdOutgoingPacket        - Implement a Traffic Control to block every third o
utgoing packet at random
XDPDropEveryThirdPacket         - Use XDP to block every third incoming packet
XDPPacketFilter                 - Use XDP to block incoming packages from specific URLs in Java
XDPPacketFilter2                - Use XDP to block incoming packages from specific URLs
demo.BlockHTTP                  - Block incoming packets from or to the HTTP port
demo.ForbiddenFile              - Prohibits access to a file named "/tmp/forbidden"
demo.ForbiddenFile2             - Prohibits access to a file named "/tmp/forbidden" using LSM hooks, currently doesn't wor
k
demo.GlobalVariableSample       - Global variable sample that counts the number of openat2 calls
demo.HelloWorld                 - Log "Hello, World!" when openat2 is called
demo.MapSample                  - Count the number of files opened per process
demo.RingSample                 - Log the filenames of openat2 calls in a ring buffer
demo.Sample                     - Hello, World from BPF
demo.XDPDropEveryThirdPacket    - Use XDP to block every third incoming packet
HttpUtil                        - LotteryScheduler                      -       * Dispatch tasks
[mydev11@koonuc15x-1 hello-ebpf-main]$
```

II. Project hello-ebpf

- ./build.sh

```
2024-12-27 23:53:32 Downloaded from aliyunmaven: https://maven.aliyun.com/repository/public/org/junit/jupiter/junit-jupiter-engine/5.10.2/junit-jupiter-engine-5.10.2.jar (245 kB at 853 kB/s)
2024-12-27 23:53:32 [INFO] ------------------------------------------------------------------------
2024-12-27 23:53:32 [INFO] Reactor Summary for hello-ebpf 0.1.1-scx-enabled-SNAPSHOT:
2024-12-27 23:53:32 [INFO]
2024-12-27 23:53:32 [INFO] hello-ebpf ..................................... SUCCESS [  0.000 s]
2024-12-27 23:53:32 [INFO] ebpf-annotations ............................... SUCCESS [ 50.237 s]
2024-12-27 23:53:32 [INFO] bpf-processor .................................. SUCCESS [ 16.352 s]
2024-12-27 23:53:32 [INFO] ebpf-shared .................................... SUCCESS [  0.142 s]
2024-12-27 23:53:32 [INFO] bpf-compiler-plugin ............................ SUCCESS [ 37.827 s]
2024-12-27 23:53:32 [INFO] bpf ............................................ FAILURE [  4.892 s]
2024-12-27 23:53:32 [INFO] bpf-compiler-plugin-test ....................... SKIPPED
2024-12-27 23:53:32 [INFO] bpf-gen ........................................ SKIPPED
2024-12-27 23:53:32 [INFO] bpf-samples .................................... SKIPPED
2024-12-27 23:53:32 [INFO] ------------------------------------------------------------------------
2024-12-27 23:53:32 [INFO] BUILD FAILURE
2024-12-27 23:53:32 [INFO] ------------------------------------------------------------------------
2024-12-27 23:53:32 [INFO] Total time:  02:03 min
2024-12-27 23:53:32 [INFO] Finished at: 2024-12-27T23:53:32+08:00
2024-12-27 23:53:32 [INFO] ------------------------------------------------------------------------
2024-12-27 23:53:32 [ERROR] Failed to execute goal on project bpf: Could not resolve dependencies for project me.bechberger:bpf:jar:0.1.1-scx-enabled-SNAPSHOT: The following artifacts could not be resolved: me.bechberger:bpf-runtime:jar:0.1.10-scx-enabled-SNAPSHOT, me.bechberger:rawbpf:jar:0.1.7-scx-enabled-SNAPSHOT: Could not find artifact me.bechberger:bpf-runtime:jar:0.1.10-scx-enabled-SNAPSHOT in aliyunmaven (https://maven.aliyun.com/repository/public) -> [Help 1]
2024-12-27 23:53:32 [ERROR]
2024-12-27 23:53:32 [ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
2024-12-27 23:53:32 [ERROR] Re-run Maven using the -X switch to enable full debug logging.
2024-12-27 23:53:32 [ERROR]
2024-12-27 23:53:32 [ERROR] For more information about the errors and possible solutions, please read the following articles:
2024-12-27 23:53:32 [ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/DependencyResolutionException
2024-12-27 23:53:32 [ERROR]
2024-12-27 23:53:32 [ERROR] After correcting the problems, you can resume the build with the command
2024-12-27 23:53:32 [ERROR]   mvn <args> -rf :bpf
```

II. Project hello-ebpf

Working on resolve this issue…still broken build though the official repository was added according to the following guidelines:

Usage as a library

The library is available as a maven package:

```
<dependency>
    <groupId>me.bechberger</groupId>
    <artifactId>bpf</artifactId>
    <version>0.1.1-scx-enabled-SNAPSHOT</version>
</dependency>
```

You might have to add the https://s01.oss.sonatype.org/content/repositories/releases/ repo:

```
<repositories>
    <repository>
        <id>snapshots</id>
        <url>https://s01.oss.sonatype.org/content/repositories/snapshots/</url>
        <releases>
            <enabled>false</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
</repositories>
```

- You have to copy the .mvn/jvm.config file and add the annotation processor to your project.

II. Project hello-ebpf

Moving from Maven to ? for hello-ebpf

- Mill

  https://mill-build.org/mill/index.html

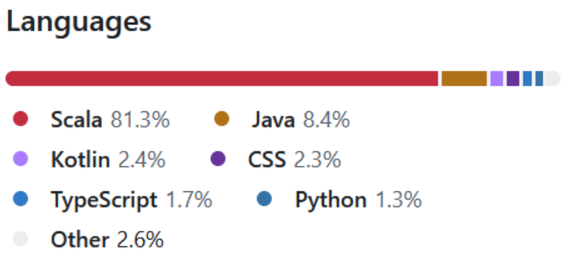
Mill: A Fast JVM Build Tool

Mill is a fast, scalable, multi-language build tool that supports Java, Scala, and Kotlin:

- Mill can build the same Java codebase 4-10x faster than Maven, or 2-4x faster than Gradle
- Mill's typed config language and immutable task graph helps keep builds clean and understandable
- Mill is an easier alternative to Bazel for large multi-language monorepos with hundreds of modules

To get started using Mill, see the language-specific introductory documentation linked below:

https://github.com/com-lihaoyi/mill



- Meson
- MX

II. Project hello-ebpf

- **Meson**

  https://mesonbuild.com/

Overview

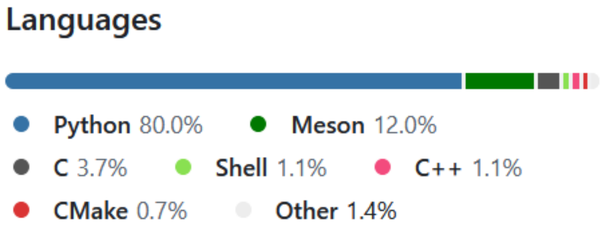
Meson is an open source build system meant to be both extremely fast, and, even more importantly, as user friendly as possible.

The main design point of Meson is that every moment a developer spends writing or debugging build definitions is a second wasted. So is every second spent waiting for the build system to actually start compiling code.

Features

- multiplatform support for Linux, macOS, Windows, GCC, Clang, Visual Studio and others
- supported languages include C, C++, D, Fortran, Java, Rust
- build definitions in a very readable and user friendly non-Turing complete DSL
- cross compilation for many operating systems as well as bare metal
- optimized for extremely fast full and incremental builds without sacrificing correctness
- built-in multiplatform dependency provider that works together with distro packages
- fun!

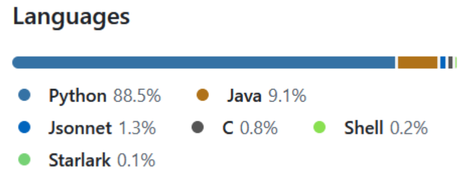
https://github.com/mesonbuild/meson



II. Project hello-ebpf

- **MX**

  https://github.com/graalvm/mx



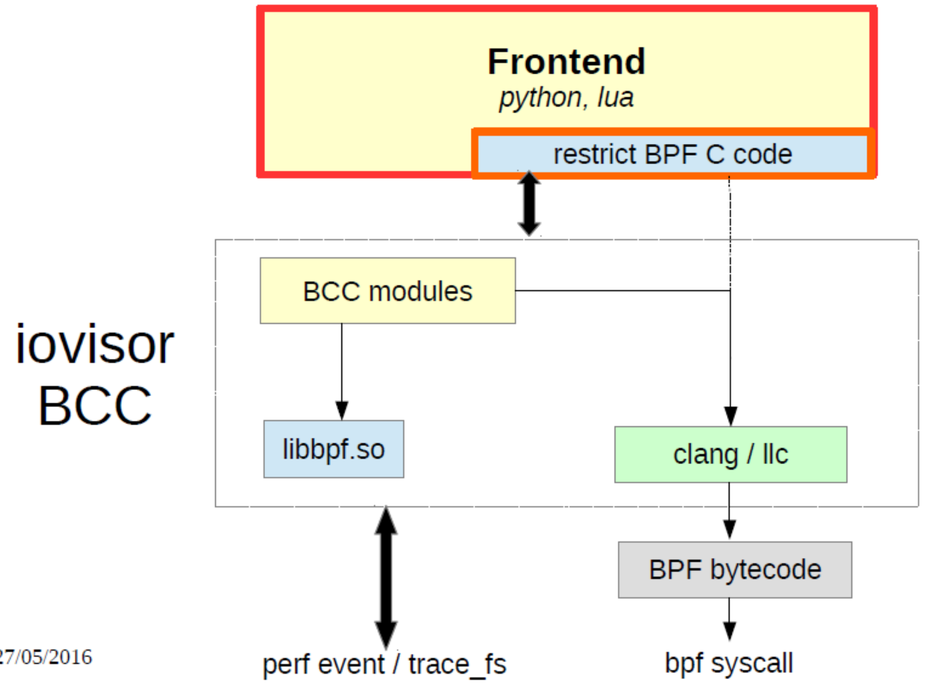
`mx` is a command line based tool for managing the development of (primarily) Java code. It includes a mechanism for specifying the dependencies as well as making it simple to build, test, run, update, etc the code and built artifacts. `mx` contains support for developing code spread across multiple source repositories. `mx` is written in Python and is extensible.

The organizing principle of `mx` is a *suite*. A *suite* is both a directory and the container for the components of the suite. A suite component is either a *project*, *library* or *distribution*. There are various flavors of each of these. A suite may import and depend on other suites. For an execution of mx, exactly one suite is the primary suite. This is either the suite in whose directory `mx` is executed or the value of the global `-p` mx option. The set of suites reachable from the primary suite by transitive closure of the imports relation form the set that `mx` operates on.

II. Project hello-ebpf

2) Architecture and design

Look back on BCC

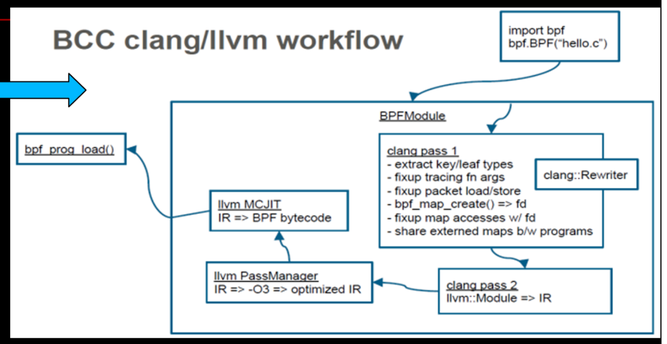


Source: http://www.slideshare.net/vh21/meet-cutebetweenebpfandtracing

A Sample

- https://lwn.net/Articles/747640/ //Some advanced BCC topics

```
#!/usr/bin/env python

from bcc import BPF
from time import sleep

program = """
    BPF_HASH(callers, u64, unsigned long);

    TRACEPOINT_PROBE(kmem, kmalloc) {
        u64 ip = args->call_site;
        unsigned long *count;
        unsigned long c = 1;

        count = callers.lookup((u64 *)&ip);
        if (count != 0)
            c += *count;

        callers.update(&ip, &c);

        return 0;
    }
"""
b = BPF(text=program)

while True:
    try:
        sleep(1)
        for k,v in sorted(b["callers"].items()):
            print ("%s %u" % (b.ksym(k.value), v.value))
        print
    except KeyboardInterrupt:
        exit()
```



Source: http://linuxplumbersconf.org/2015/ocw//system/presentations/3249/original/bpf_llvm_2015aug19.pdf

The output from this little program looks like:

```
# ./example.py
i915_sw_fence_await_dma_fence 4
intel_crtc_duplicate_state 4
SyS_memfd_create 1
drm_atomic_state_init 4
sg_kmalloc 7
intel_atomic_state_alloc 4
seq_open 504
SyS_bpf 22
```

Source: "Python for Linux Kernel Debugging", Feng Li, PyCon China(Hangzhou) 2019.

II. Project hello-ebpf

2.1 Hierarchical structure

-

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree -L 1 hello-ebpf-main
hello-ebpf-main
├── annotations
├── bin
├── bpf
├── bpf-compiler-plugin
├── bpf-compiler-plugin-test
├── bpf-gen
├── bpf-processor
├── bpf-runtime
├── bpf-samples
├── build.sh
├── debug_bpf.sh
├── debug.sh
├── hello-ebpf.yaml
├── img
├── LICENSE
├── mvnw
├── pom.xml
├── rawbpf
├── README.md
├── run_bpf.sh
├── run.sh
├── shared
└── testutil
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/testutil
hello-ebpf-main/testutil
├── bin
│   └── java
├── find_and_get_kernel.py
├── oci-lib.sh
├── README.md
├── run-in-container.sh
└── setup-and-run.sh
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ cat hello-ebpf-main/bin/install.sh
#!/bin/bash
sudo bash -c "echo \"\\\$nrconf{restart} = 'a'\" >> /etc/needrestart/needrestart.conf"
sudo apt-get install -y apt-transport-https ca-certificates curl clang llvm
sudo apt-get install -y libelf-dev libpcap-dev libbfd-dev binutils-dev build-essential make
sudo apt-get install -y linux-tools-common linux-tools-$(uname -r)
sudo apt-get install -y libbpf-dev
sudo apt-get install -y python3-pip zsh tmux
sudo apt-get install -y zip unzip git
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
sdk install java 22-sapmchn
sdk install maven[mydev11@koonuc15x-1 Hello-eBPF]$
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ cat hello-ebpf-main/run.sh
#!/bin/zsh
# Run a sample program from the bpf-samples module
# Usage: ./run.sh <sample> [args]

# Navigate to current folder
cd "$(dirname "$0")"/bpf-samples || exit

# if empty arguments or help flag, print help
if [ $# -eq 0 ] || [ "$1" = "-h" ] || [ "$1" = "--help" ]; then
    echo "Usage: $0 <sample>"
    echo "Available samples:"
    (cd src/main/java/me/bechberger/ebpf/samples && (
      find . -name "*.java" | while read file; do
        f=$(echo "$file" | sed 's/\//\./g')
        f=${f:2}
        printf "%-35s - " "${f%.java}"
        awk '/\/\*\*/{getline; sub(/^ \* /, ""); print; exit}' "$file"
      done
    ))
    exit 0
fi

CLASS=$1

# Run the program
shift
java -cp target/bpf-samples.jar --enable-native-access=ALL-UNNAMED $JAVA_OPTS me.bechberger.ebpf.samples.$CLASS $@[mydev11@koo
nuc15x-1 Hello-eBPF]$
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ cat hello-ebpf-main/build.sh
#!/bin/sh

cd "$(dirname "$0")" || exit

./mvnw package[mydev11@koonuc15x-1 Hello-eBPF]$
```

II. Project hello-ebpf

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/bpf-runtime
hello-ebpf-main/bpf-runtime
├── build.sh
├── CHANGELOG
├── deploy.sh
├── pom.xml
└── README.md
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ cat hello-ebpf-main/bpf-runtime/build.sh
#!/usr/bin/env sh

set -e

# get parent-parent folder, shell agnostic
cd "$(dirname "$0")"/.. || exit

(cd bpf-runtime; mvn clean)

mvn package -U && MAVEN_OPTS="-Xss1000m" time java -jar bpf-gen/target/bpf-gen.jar bpf-runtime/src/main/java/ bpf-gen/data/hel
per-defs.json

(cd bpf-runtime; MAVEN_OPTS="-Xss1000m" mvn package -U)
[mydev11@koonuc15x-1 Hello-eBPF]$
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ cat hello-ebpf-main/bpf-runtime/deploy.sh
#!/usr/bin/env sh

set -e

# get parent-parent folder, shell agnostic
cd "$(dirname "$0")"/.. || exit

(cd bpf-runtime; mvn clean; MAVEN_OPTS="-Xss1000m" mvn package deploy -U)[mydev11@koonuc15x-1 Hello-eBPF]$
[mydev11@koonuc15x-1 Hello-eBPF]$
```

II. Project hello-ebpf

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/annotations
hello-ebpf-main/annotations
├── pom.xml
├── README.md
└── src
    └── main
        └── java
            └── me
                └── bechberger
                    └── ebpf
                        └── annotations
                            ├── AlwaysInline.java
                            ├── AnnotationInstances.java
                            ├── bpf
                            │   ├── BPFFunctionAlternative.java
                            │   ├── BPFFunction.java
                            │   ├── BPFImpl.java
                            │   ├── BPFInterface.java
                            │   ├── BPF.java
                            │   ├── BPFMapClass.java
                            │   ├── BPFMapDefinition.java
                            │   ├── BuiltinBPFFunction.java
                            │   ├── InternalBody.java
                            │   ├── InternalMethodDefinition.java
                            │   ├── MethodIsBPFRelatedFunction.java
                            │   ├── NotUsableInJava.java
                            │   ├── Properties.java
                            │   ├── PropertyDefinition.java
                            │   ├── PropertyDefinitions.java
                            │   ├── Property.java
                            │   └── Requires.java
                            ├── CustomType.java
                            ├── EnumMember.java
                            ├── Includes.java
                            ├── InlineUnion.java
                            ├── Offset.java
                            ├── OriginalName.java
                            ├── OriginalNames.java
                            ├── Size.java
                            ├── Sizes.java
                            ├── Type.java
                            └── Unsigned.java
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/bpf-processor
hello-ebpf-main/bpf-processor
├── pom.xml
├── README.md
└── src
    ├── main
    │   └── java
    │       └── me
    │           └── bechberger
    │               ├── cast
    │               │   └── CAST.java
    │               └── ebpf
    │                   ├── bpf
    │                   │   └── processor
    │                   │       ├── AnnotationUtils.java
    │                   │       ├── BPFTypeLike.java
    │                   │       ├── CompilationCache.java
    │                   │       ├── CompilerErrorProcessor.java
    │                   │       ├── DefinedTypes.java
    │                   │       ├── Processor.java
    │                   │       └── TypeProcessor.java
    │                   ├── NameUtil.java
    │                   └── type
    │                       ├── BoxHelper.java
    │                       ├── BPFType.java
    │                       ├── Enum.java
    │                       ├── Ptr.java
    │                       ├── Struct.java
    │                       ├── TypedefBase.java
    │                       ├── Typedef.java
    │                       ├── TypedEnum.java
    │                       ├── TypeUtils.java
    │                       └── Union.java
    └── test
        └── java
            └── me
                └── bechberger
                    ├── cast
                    │   └── ASTTest.java
                    └── ebpf
                        └── bpf
                            └── processor
                                └── CompilerErrorProcessorTest.java
```

II. Project hello-ebpf

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/shared
hello-ebpf-main/shared
├── pom.xml
├── README.md
└── src
    └── main
        └── java
            └── me
                └── bechberger
                    └── ebpf
                        └── shared
                            ├── Constants.java
                            ├── Disassembler.java
                            ├── KernelFeatures.java
                            ├── LibC.java
                            ├── PanamaUtil.java
                            ├── Syscalls.java
                            ├── TraceLog.java
                            ├── util
                            │   ├── DiffUtil.java
                            │   └── LineReader.java
                            └── Util.java
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/bpf-compiler-plugin
hello-ebpf-main/bpf-compiler-plugin
├── pom.xml
├── README.md
└── src
    └── main
        ├── java
        │   └── me
        │       └── bechberger
        │           └── ebpf
        │               └── bpf
        │                   └── compiler
        │                       ├── CompilerPlugin.java
        │                       ├── MethodHeaderTemplate.java
        │                       ├── MethodTemplateCache.java
        │                       ├── MethodTemplate.java
        │                       ├── NullHelpers.java
        │                       └── Translator.java
        └── resources
            └── META-INF
                └── services
                    └── com.sun.source.util.Plugin
```

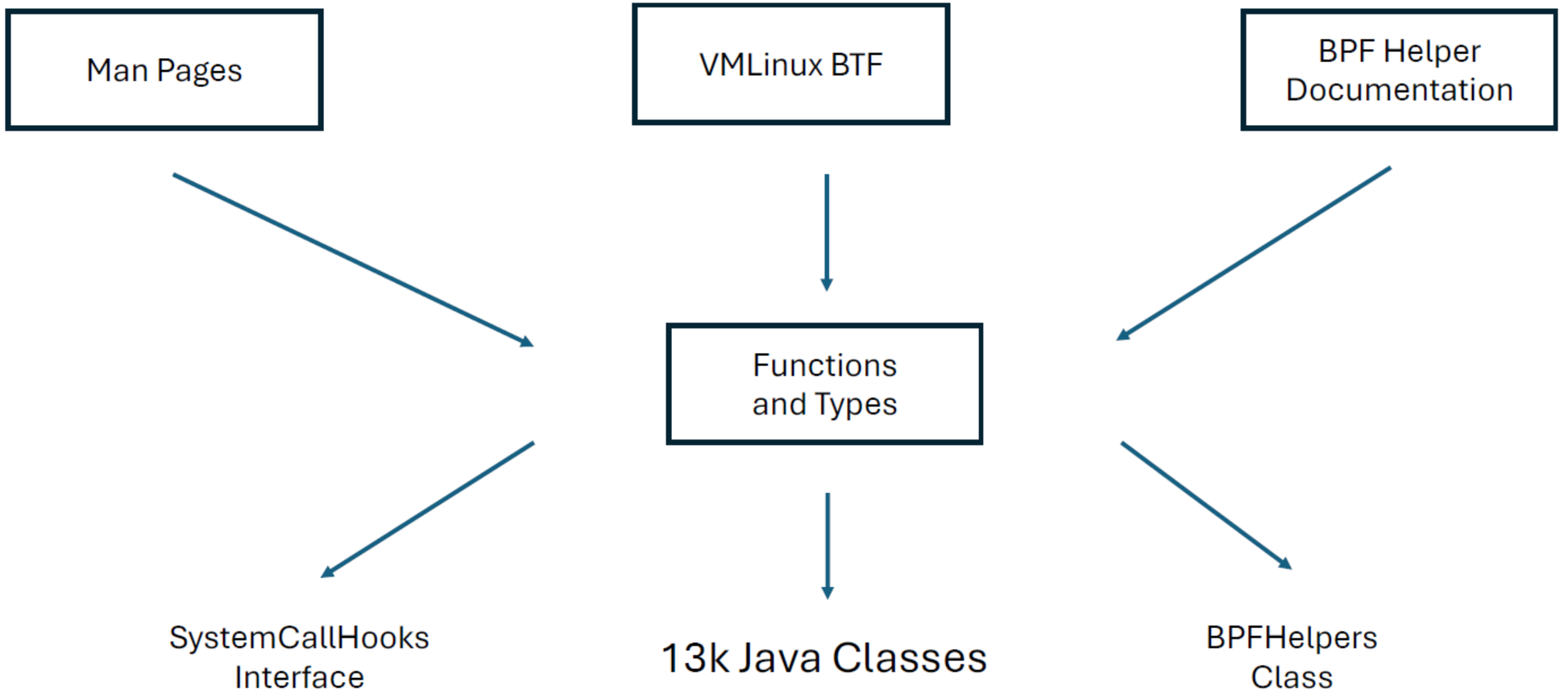
II. Project hello-ebpf

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/bpf
hello-ebpf-main/bpf
├── pom.xml
├── README.md
└── src
    ├── main
    │   └── java
    │       └── me
    │           └── bechberger
    │               └── ebpf
    │                   └── bpf
    │                       ├── BPFError.java
    │                       ├── BPFJ.java
    │                       ├── BPFProgram.java
    │                       ├── CGroupHook.java
    │                       ├── GlobalVariable.java
    │                       ├── LSMHook.java
    │                       ├── map
    │                       │   ├── BPFArray.java
    │                       │   ├── BPFBaseMap.java
    │                       │   ├── BPFHashMap.java
    │                       │   ├── BPFLRUHashMap.java
    │                       │   ├── BPFMap.java
    │                       │   ├── BPFQueueAndStack.java
    │                       │   ├── BPFQueue.java
    │                       │   ├── BPFRingBuffer.java
    │                       │   ├── BPFStack.java
    │                       │   ├── FileDescriptor.java
    │                       │   └── MapTypeId.java
    │                       ├── NetworkUtil.java
    │                       ├── Scheduler.java
    │                       ├── TCHook.java
    │                       ├── Util.java
    │                       └── XDPHook.java
    └── test
        └── java
            └── me
                └── bechberger
                    └── ebpf
                        └── bpf
                            ├── ArrayMapTest.java
                            ├── DataTypeTest.java
                            ├── FEntryExitAutoAttachTest.java
                            ├── GlobalVariableTest.java
                            ├── HashMapTest.java
                            ├── HelloWorldTest.java
                            ├── MapGenerationTest.java
                            ├── QueueMapTest.java
                            ├── RingBufferTest.java
                            ├── TestUtil.java
                            ├── TypeLayoutTest.java
                            └── TypeProcessingTest.java
```

```
[mydev11@koonuc15x-1 Hello-eBPF]$ tree hello-ebpf-main/bpf-gen
hello-ebpf-main/bpf-gen
├── data
│   ├── helper-defs.json
│   └── README.md
├── pom.xml
├── README.md
└── src
    ├── main
    │   └── java
    │       └── me
    │           └── bechberger
    │               └── ebpf
    │                   └── gen
    │                       ├── BTF.java
    │                       ├── DeclarationParser.java
    │                       ├── Generator.java
    │                       ├── HelperJSONProcessor.java
    │                       ├── KnownTypes.java
    │                       ├── Main.java
    │                       ├── Markdown.java
    │                       └── SystemCallProcessor.java
    └── test
        └── java
            └── me
                └── bechberger
                    └── ebpf
                        └── gen
                            ├── DeclarationParserTest.java
                            ├── GeneratorTest.java
                            └── SystemCallProcessorTest.java
```

II. Project hello-ebpf

2.2 VMLinux

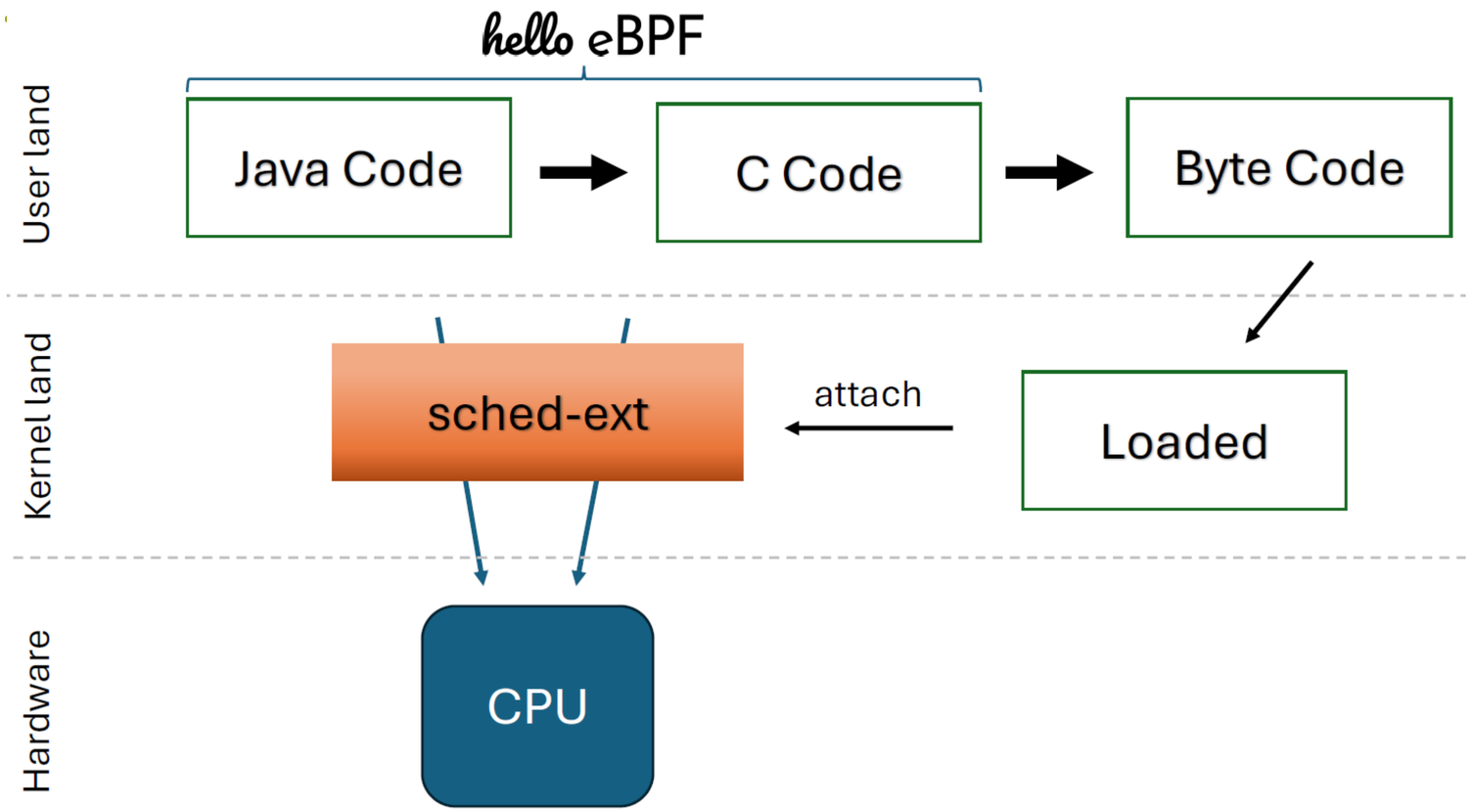


Source: “hello-ebpf: Writing eBPF programs directly in Java”, Johannes Bechberger, LPC 2024.

II. Project hello-ebpf

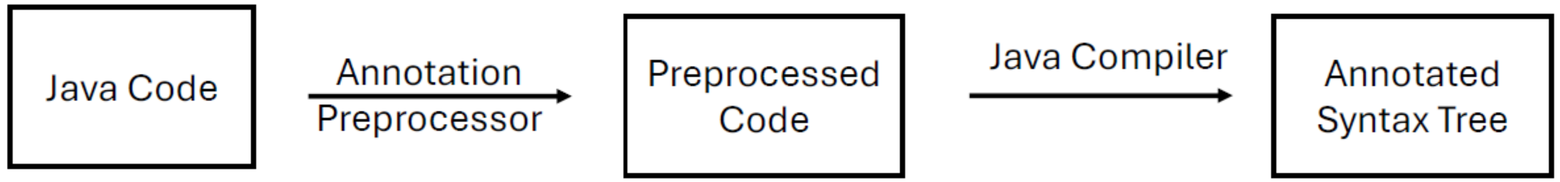
3) Internals

Workflow



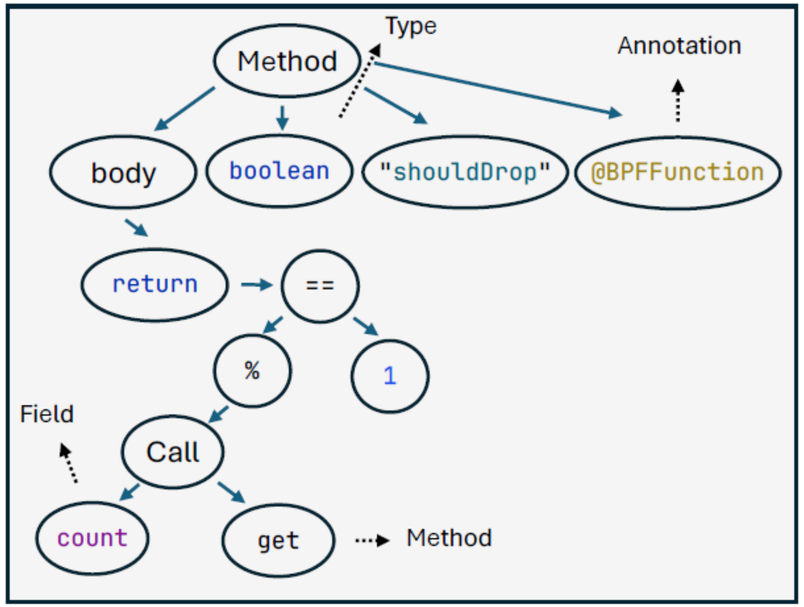
Source: "hello-ebpf: Writing eBPF programs directly in Java", Johannes Bechberger, LPC 2024.

II. Project hello-ebpf

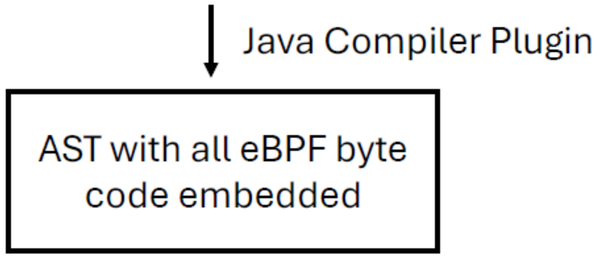



```
final GlobalVariable
 <@Unsigned Integer> count
  = ...;

@BPFFunction
public boolean shouldDrop() {
    return count.get() % 3
           == 1;
}
```

```
static final String
   EBPF_PROGRAM = """
        // license, ...
        u32 count SEC(".data");
        """;


@BPFFunction
public boolean shouldDrop() {
    return count.get() % 3
           == 1;
}
```




```
@BPF(license = "GPL")
public abstract class XDPDropEveryThirdPacket
extends BPFProgram implements XDPHook {

    final GlobalVariable<@Unsigned Integer> count = new GlobalVariable<>(0);

    @BPFFunction
    public boolean shouldDrop() {
        return count.get() % 3 == 1;
    }

    @Override
    public xdp_action xdpHandlePacket(Ptr<xdp_md> ctx) {
        count.set(count.get() + 1);
        return shouldDrop() ? xdp_action.XDP_DROP : xdp_action.XDP_PASS;
    }

    // ...
}
```



Source: “hello-ebpf: Writing eBPF programs directly in Java”, Johannes Bechberger, LPC 2024.

II. Project hello-ebpf

Data structures

- **Structs**

```
@Type
struct Event {
    @Unsigned int pid;
    @Size(256) String f;
}

@BPFFunction
int access(       Event event) {
    event.pid = 5;
    return event.pid;
}
```

```
struct Event {
    u32 pid;
    u8 filename[256];
};


s32 access(struct Event event) {
    event.pid = 5;
    return event.pid;
}
```

- **Unions**

```
@Type
static class SampleUnion extends Union {
    @Unsigned
    int ipv4;
    long count;
}
```

Source: “hello-ebpf: Writing eBPF programs directly in Java”, Johannes Bechberger, LPC 2024.

II. Project hello-ebpf

- **Pointers** Ptr<T>

```java
public class Ptr<T> {

    @BuiltinBPFFunction("(*($this))")
    public T val() {}

    @BuiltinBPFFunction("&($arg1)")
    public static <T> Ptr<T> of(@Nullable T value){}

    @BuiltinBPFFunction("((void*)0)")
    public static Ptr<?> ofNull() {}

    @BuiltinBPFFunction("(($T1*)$this)")
    public <S> Ptr<S> cast() {}
    // ...
}
```

```java
@BPFFunction
public int refAndDeref() {
    int value = 3;
    Ptr<Integer> ptr = Ptr.of(value);
    return ptr == Ptr.ofNull() ? 1 : 0;
}
```

```
s32 refAndDeref() {
    s32 value = 3;
    s32*          ptr = &value;
    return ptr == ((void*)0)   ? 1 : 0;
}
```

Source: “hello-ebpf: Writing eBPF programs directly in Java”, Johannes Bechberger, LPC 2024.

II. Project hello-ebpf

- **Maps**

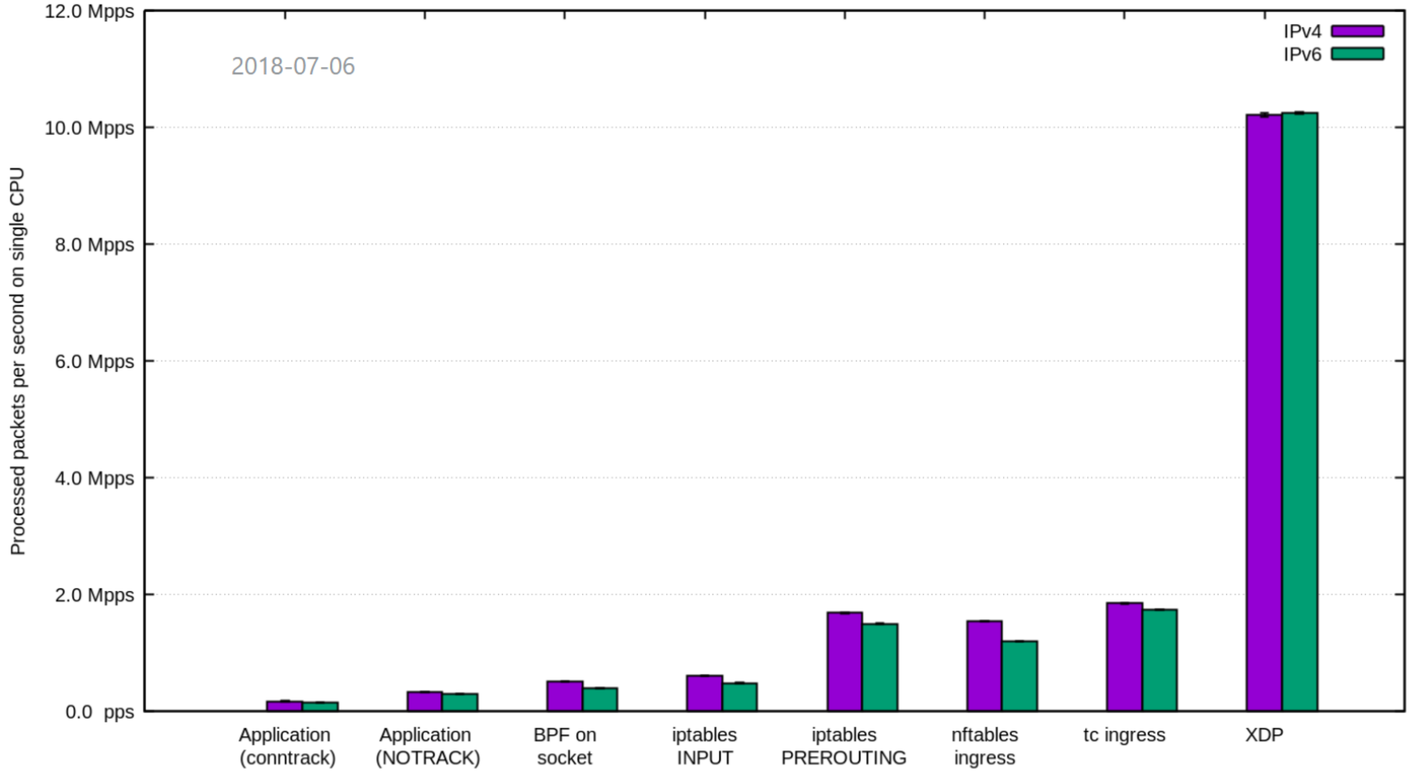
```
@BPFMapDefinition(maxEntries = 100 * 1024)
BPFHashMap<@Size(STRING_SIZE) String, Entry> map;
// in eBPF
String key = ...;
map.bpf_get(key); map.put(key, entry);
// in Java
program.map.get(key)
program.map.put(key, value)
```

Source: “hello-ebpf: Writing eBPF programs directly in Java”, Johannes Bechberger, LPC 2024.

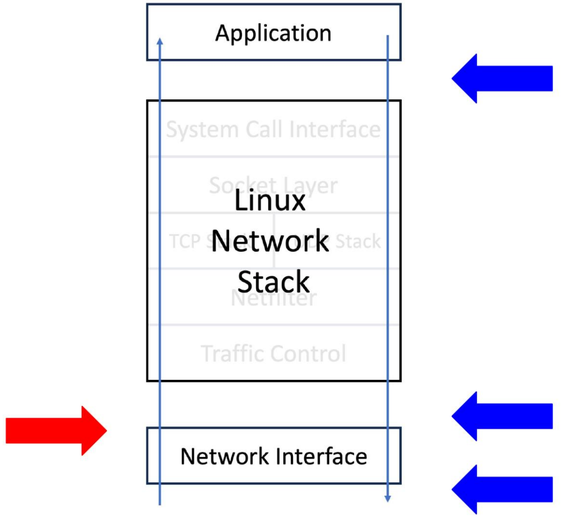
4) Write eBPF programs in pure Java

4.1 Building a firewall

Packet dropping performance



Source: https://blog.cloudflare.com/how-to-drop-10-million-packets/



Source: "Building a Lightning Fast Firewall in Java & eBPF", Johannes Bechberger, Javazone 2024.

II. Project hello-ebpf

4.1.1 Building a Lightning Fast Firewall with Java & eBPF

Libraries

- **Implement a packet parsing library**

```
@BPFInterface
public interface BasePacketParser {

    int HTTP_PORT = 80;
    int HTTPS_PORT = 443;

    // ...

    @Type
    static class PacketInfo {
        public PacketDirection direction;
        public Protocol protocol;
        public IPAddress source;
        public IPAddress destination;
        public @Unsigned int destinationPort;
        public @Unsigned int sourcePort;
        public int length;
    }

    // ...

    @BPFFunction
    @AlwaysInline
    default boolean parsePacket2(@Unsigned int start,
      @Unsigned int end, Ptr<PacketInfo> info) {
        return parsePacket(Ptr.voidPointer(start), Ptr.voidPointer(e
    }

    /**
     * Parse a packet and extract the source and destination IP addr
     *
     * @param start start of the packet data
     * @param end   end of the packet data
     * @param info  output parameter for the extracted information
     * @return true if the packet is an IP packet and could be parse
     */
    @BPFFunction
    @AlwaysInline
    default boolean parsePacket(Ptr<?> start,
      Ptr<?> end, Ptr<PacketInfo> info) {

        // ...

        if (ethType == XDPHook.ETH_P_IP) {
            return parseIPPacket(
              start.add(offset).<runtime.iphdr>cast(), end, info);
        }
        if (ethType == XDPHook.ETH_P_IPV6) {
            return parseIPv6Packet(
              start.add(offset).<runtime.ipv6hdr>cast(), end, info);
        }
        return false;
    }
}
```

This library mechanism allows you to use the methods by simply implementing the interface in your BPF program, making it fairly easy to parse and filter packets (see BlockHTTP on GitHub):

```
@BPF(license = "GPL")
public abstract class BlockHTTP2 extends BPFProgram
  implements XDPHook, BasePacketParser {

    @Override
    public xdp_action xdpHandlePacket(Ptr<xdp_md> packet) {
        PacketInfo info = new PacketInfo();
        if (parsePacket2(packet.val().data,
                         packet.val().data_end,
                         Ptr.of(info))) {
            if (info.sourcePort == HTTP_PORT) {
                BPFJ.bpf_trace_printk("Dropping https packet\n");
                return xdp_action.XDP_DROP;
            }
        }
        return xdp_action.XDP_PASS;
    }

    // ...
}
```

The main advantage of implementing libraries as Java interfaces is that their properties align:

- Interfaces can't have instance variables, so defining global variables or maps is impossible. This constraint simplifies the implementation of the library mechanism, as the Java compiler plugin has to deal with less complexity.
- A program can implement and, therefore, use multiple interfaces.
- Finding the used libraries is easy, as they are all declared initially.

We can extend this into a firewall with rules. The implementation is split into the base firewall (GitHub) and the Spring based frontend (GitHub).

hello-ebpf / bpf-samples / src / main / java / me / bechberger / ebpf / samples / demo / **BlockHTTP.java**

parttimenerd Rename demo classes

Code | Blame | 36 lines (29 loc) · 1.07 KB

```
1  package me.bechberger.ebpf.samples.demo;
2
3  import me.bechberger.ebpf.annotations.bpf.BPF;
4  import me.bechberger.ebpf.bpf.*;
5
6  import me.bechberger.ebpf.samples.BasePacketParser;
7  import me.bechberger.ebpf.type.Ptr;
8
9  import static me.bechberger.ebpf.runtime.XdpDefinitions.*;
10
11
12 /**
13  * Block incoming packets from or to the HTTP port
14  */
15 @BPF(license = "GPL")
16 public abstract class BlockHTTP extends BPFProgram implements XDPHook, BasePacketParser {
17
18     @Override
19     public xdp_action xdpHandlePacket(Ptr<xdp_md> packet) {
20         PacketInfo info = new PacketInfo();
21         if (parsePacket2(packet.val().data, packet.val().data_end, Ptr.of(info))) {
22             if (info.sourcePort == HTTP_PORT) {
23                 BPFJ.bpf_trace_printk("Dropping http packet");
24                 return xdp_action.XDP_DROP;
25             }
26         }
27         return xdp_action.XDP_PASS;
28     }
29
30     public static void main(String[] args) throws InterruptedException {
31         try (BlockHTTP program = BPFProgram.load(BlockHTTP.class)) {
32             program.xdpAttach();
33             program.tracePrintLoop();
34         }
35     }
36 }
```

Source: https://mostlynerdless.de/blog/2024/08/27/hello-ebpf-building-a-lightning-fast-firewall-with-java-ebpf-14/

II. Project hello-ebpf

Ruling

Rule Data Structures

First, we define our data structures. Rules are based on the IP address and the source port:

```
/** A connection consists of an IP address and a port */
@Type
record IPAndPort(int ip, int port) {
}

@Type
record FirewallRule(
    @Unsigned int ip,
    /**
      * Ignore the low n bytes for matching,
      * these bytes have to be zero in the ip field
      */
    int ignoreLowBytes,
    /* or -1 to match all ports */
    int port) {
}

@Type
enum FirewallAction implements Enum<FirewallAction> {
    ALLOW, DROP,
    /** No rule applies */
    NONE
}
```

The firewall rules are stored in a map:

```
@BPFMapDefinition(maxEntries = 1000)
BPFHashMap<FirewallRule, FirewallAction> firewallRules;
```

Finding the matching Firewall Rule

Given a connection (`IPAndPort`), how can we find the associated action? We can observe that an IP address `W.X.Y.Z` is matched by the rules, where the most specific rule wins:

- `W.X.Y.Z` ignoring no bytes
- `W.X.Y.0` ignoring the lowest byte
- ...
- `0.0.0.0` ignoring all bytes

```
@BPFFunction
@AlwaysInline
FirewallAction computeAction(Ptr<IPAndPort> info) {
    // for all possible ip address matching bytes
    for (int i = 0; i < 5; i++) {
        var action = computeSpecificAction(info, i);
        if (action != FirewallAction.NONE) {
            return action;
        }
    }
    return FirewallAction.NONE;
}
```

The `computeSpecificAction` method checks for a rule with a matching IP address, which ignores the specified lowest bytes. Each rule has a specified port, so we have to check for the two values, `-1` matching every port, and the port of the connection:

```
// This method can be both used in the eBPF and the Java part
@BPFFunction
@AlwaysInline
static int zeroLowBytes(int ip, int ignoreLowBytes) {
    return ip & (0xFFFFFFFF << (ignoreLowBytes * 8));
}

@BPFFunction
@AlwaysInline
FirewallAction computeSpecificAction(
  Ptr<IPAndPort> info, int ignoreLowBytes) {
    int ip = info.val().ip;
    // first null the bytes that should be ignored
    int matchingAddressBytes = zeroLowBytes(ip, ignoreLowBytes);
    if (matchingAddressBytes == 100) {
        // don't ask, the code only works with this line
        // it's probably a compiler bug and this line adds
        // some memory dependencies, influencing the compiler
        // optimizations
        bpf_trace_printk("Checking rule for %d:%d\n",
                         matchingAddressBytes, info.val().port);
    }
    var rule = new FirewallRule(
                  matchingAddressBytes,
                  ignoreLowBytes,
                  info.val().port);
    var action = firewallRules.bpf_get(rule);
    if (action != null) {
        return action.val();
    }
    rule = new FirewallRule(
                  matchingAddressBytes,
                  ignoreLowBytes, -1);
    action = firewallRules.bpf_get(rule);
    if (action != null) {
        return action.val();
    }
    return FirewallAction.NONE;
}
```

We're querying the `firewallRules` map between one and ten times for every connection. This amount of querying is quite a lot, so we cache this in the `resolvedRules` map:

```
@BPFMapDefinition(maxEntries = 1000)
BPFLRUHashMap<IPAndPort, FirewallAction> resolvedRules;

@BPFFunction
@AlwaysInline
FirewallAction getAction(Ptr<PacketInfo> packetInfo) {
    IPAndPort ipAndPort = new IPAndPort(
            packetInfo.val().source.ipv4(),
            packetInfo.val().sourcePort);
    Ptr<FirewallAction> action = resolvedRules.bpf_get(ipAndPort);
    if (action != null) {
        return action.val();
    }
    var newAction = computeAction(Ptr.of(ipAndPort));
    resolvedRules.put(ipAndPort, newAction);
    return newAction;
}
```

Firewall

With the ability to find a matching action, extending the previous XDP example into a proper firewall is pretty simple. The only two additions are code to parse firewall rules like:

```
// Drop all HTTP (port 80) packets from the first
// IP address for google.com
google.com:HTTP drop

// Let all packets from the local network pass
192.168.0.0:-1/16 pass
```

And code that logs all dropped connections in a ring buffer:

```
@Type
record LogEntry(IPAndPort connection, long timeInMs) {
}

@BPFMapDefinition(maxEntries = 1000 * 128)
BPFRingBuffer<LogEntry> blockedConnections;
```

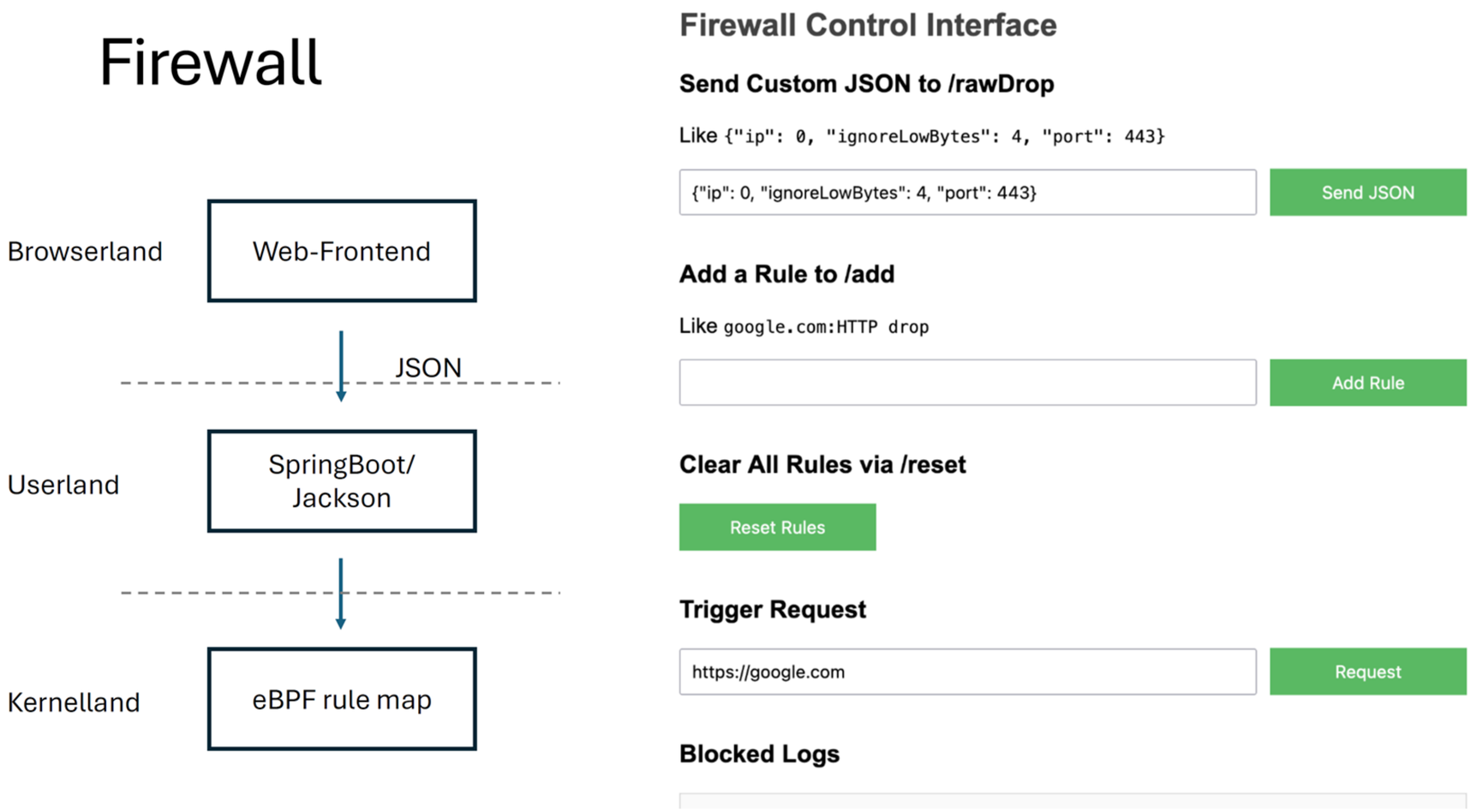
I spare you the details of both, but I show you the result:

```
> ./run.sh Firewall "google.com:HTTP drop"
Rule: FirewallRule[ip=1857682062, ignoreLowBytes=0, port=80] action:
# run wget http://google.com in a separate shell
Blocked packet from 142.250.185.110 port 80
Blocked packet from 142.250.185.110 port 80
Blocked packet from 142.250.185.110 port 80
Blocked packet from 142.250.185.110 port 80
```

Source: https://mostlynerdless.de/blog/2024/08/27/hello-ebpf-building-a-lightning-fast-firewall-with-java-ebpf-14/

II. Project hello-ebpf

Demo



Source: “hello-ebpf: Writing eBPF programs directly in Java”, Johannes Bechberger, LPC 2024.

II. Project hello-ebpf

4.1.2 bpfilter

- #CONFIG_BPFILTER //removed

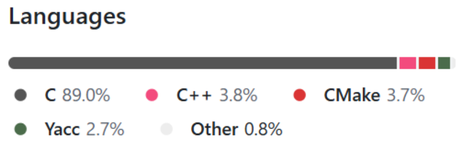
Project bpfilter

- https://bpfilter.io/
- https://github.com/facebook/bpfilter

bpfilter is an eBPF-based packet filtering framework designed to translate filtering rules into BPF programs. It comprises three main components:

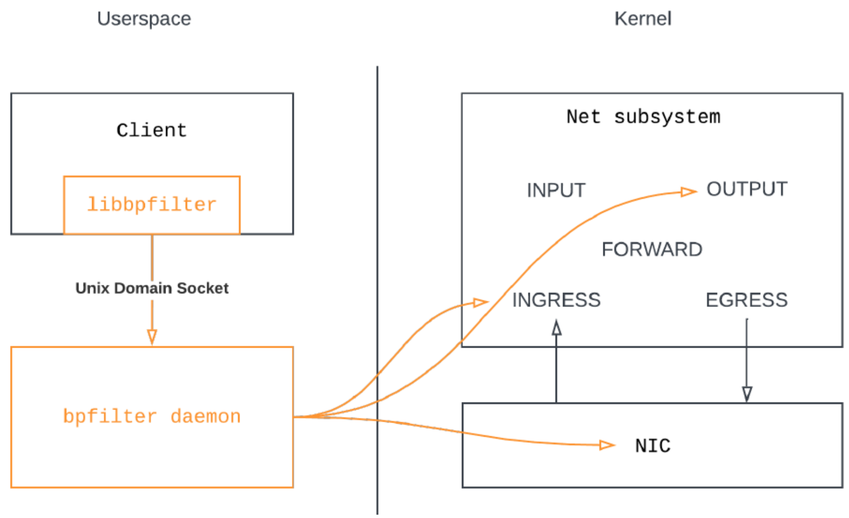
1. A daemon that runs on the host, translating filtering rules into BPF programs.
2. A lightweight library to facilitate communication with the daemon.
3. A dedicated command line interface to define the filtering rules.

A typical usage workflow would be to start the `bpfilter` daemon, then define the filtering rules using `bfcli` (part of the `bpfilter` project), `nftables` or `iptables`. The `bpfilter` daemon will be responsible for translating the filtering rules into custom BPF programs, and loading them on the system.



New bpfilter

- Complete refactor of the project
- Two main parts now:
  - `libbpfilter`
  - `bpfilter` daemon



Source: https://www.socallinuxexpo.org/scale/21x/presentations/bpfilter-bpf-based-packet-filtering-framework

II. Project hello-ebpf

4.2 Writing a Linux scheduler

4.2.1 sched-ext

- **https://github.com/sched-ext/scx/**

Languages

C 96.4% Rust 3.3% Other 0.3%

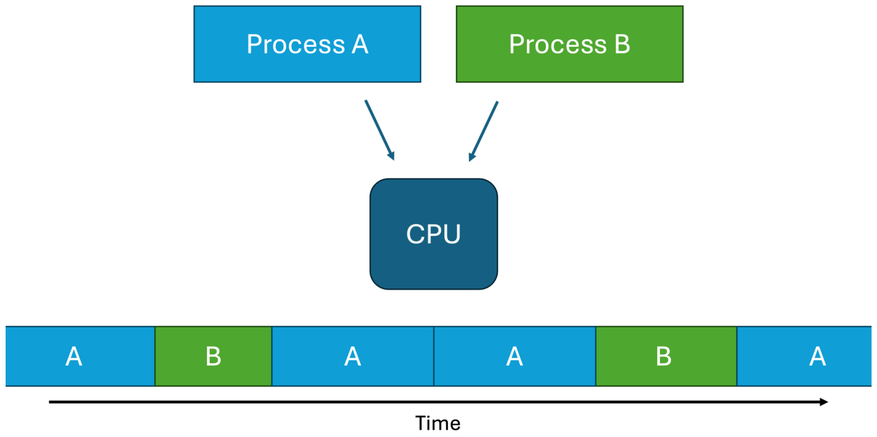
Sched_ext Schedulers and Tools

sched_ext is a Linux kernel feature which enables implementing kernel thread schedulers in BPF and dynamically loading them. This repository contains various scheduler implementations and support utilities.

sched_ext enables safe and rapid iterations of scheduler implementations, thus radically widening the scope of scheduling strategies that can be experimented with and deployed; even in massive and complex production environments.

You can find more information, links to blog posts and recordings, in the wiki. The following are a few highlights of this repository.

- The scx_layered case study concretely demonstrates the power and benefits of sched_ext.
- For a high-level but thorough overview of the sched_ext (especially its motivation), please refer to the overview document.
- For a description of the schedulers shipped with this tree, please refer to the schedulers document.
- The following video is the scx_rustland scheduler which makes most scheduling decisions in userspace Rust code showing better FPS in terraria while kernel is being compiled. This doesn't mean that scx_rustland is a better scheduler but does demonstrate how safe and easy it is to implement a scheduler which is generally usable and can outperform the default scheduler in certain scenarios.



Source: https://mostlynerdless.de/blog/2024/09/10/hello-ebpf-writing-a-linux-scheduler-in-java-with-ebpf-15/

- **https://www.kernel.org/doc/html/next/scheduler/sched-ext.html**

II. Project hello-ebpf

Kernel 6.12

- **https://kernelnewbies.org/Linux_6.12**

Linux 6.12 was released on Sunday, 17 Nov 2024 .

Summary: This release includes realtime support (PREEMPT_RT), a feature that has been in the works for 20 years. It also includes complete support for the EEVDF task scheduler; the ability to write task scheduling algorithms using BPF; support for printing a QR code on panic screens with debug information; support for zero-copy receive TCP payloads to a DMABUF region of memory while packet headers land separately in normal kernel buffers; a new linux security modules that enforces that binaries must come from integrity-protected storage; support for Memory Protection Keys in ARM; and XFS support for block sizes larger than a memory page. As always, there are many other features, new drivers, improvements and fixes.

1.3. BPF based task scheduling algorithms with sched_ext

Task scheduling algorithms are a complex topic, and experimentation or even personalization can provide great improvements. This release includes the first pieces of `sched_ext`, a feature that enables to write task scheduler algorithms in BPF, which provides a much faster development iteration and enables personalization of task scheduling

Recommended LWN article: The extensible scheduler class ← **https://lwn.net/Articles/922405/**

Documentation:

- Extensible Scheduler Class
- sched_ext overview

https://github.com/sched-ext/scx/blob/main/OVERVIEW.md

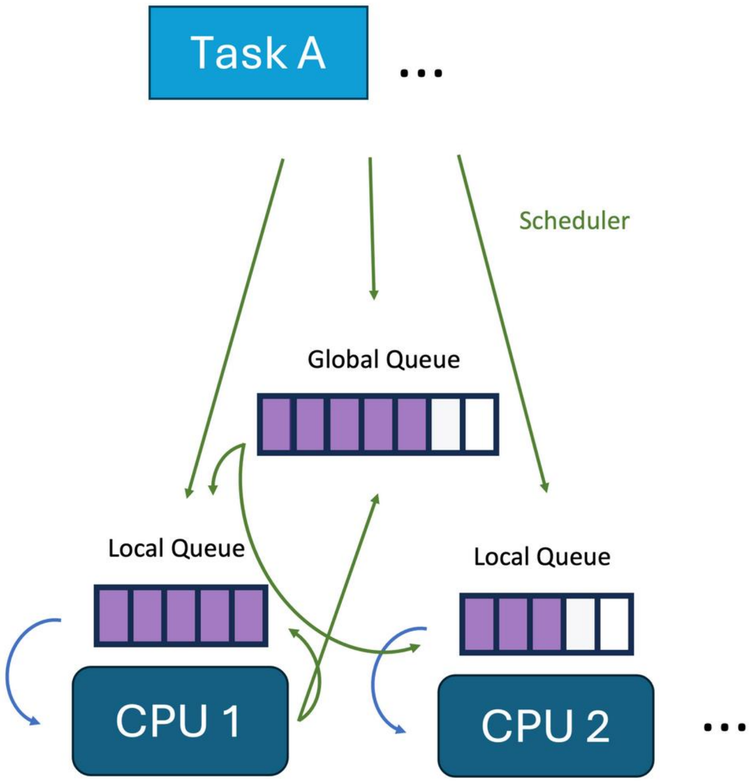
II. Project hello-ebpf

4.2.2 Writing a Linux scheduler in Java with eBPF

Design

Basic Idea

Most basic schedulers have two sets of queues: local scheduling queues per CPU each CPU uses to get the next task to execute and global queues used for inter-CPU distribution of functions. The task of the scheduler is to move the functions between the queues based on its scheduling policy:



The Scheduler Interface

The primary data structure of sched-ext is the sched_ext_ops operations table. Implementing the Scheduler interface methods allows you to populate this table with custom methods. Let's take a look at the different methods in the following; we see later how we can implement them to create two small schedulers. I recommend reading the ext.c kernel source file, which defines all sched-ext related methods and structs and is well documented. The following description is a shortened version of the comments on sched_ext_ops. To avoid the confusing terms "thread" and "process," I'll be using "task" in the following, which is the process in the Linux kernel sense of the word and effectively means a thread in most applications.

Now to the description of the methods, albeit just of the subset that is required for the small demo sampler:

- **int init()**: Called when the scheduler is initialized, used typically to initialize scheduling queues and other data structures, might return an error code (as far as I understand).
- **void exit(Ptr<scx_exit_info> ei)**: Called when the scheduler exits, the exit info contains, for example, the exit reason.
- **int selectCPU(Ptr<task_struct> p, int prev_cpu, long wake_flags)**: Called to select the CPU for a task that is just woken up (and is soon to be scheduled). A task can be directly dispatched to an idle CPU by using the `scx_bpf_dispatch()` in this method, dispatching it to the returned CPU (as far as I understand). If the task is later dispatched to `SCX_DSQ_LOCAL`, then it will be dispatched to the scheduling CPU of the selected CPU.
- **void enqueue(Ptr<task_struct> p, long enq_flags)**: Called whenever a task is ready to be dispatched again if it is not already dispatched in `selectCPU()`. This method typically enqueues the task in a queue, see `selectCPU`, or dispatches the task directly to the selected CPU using the `scx_bpf_dispatch()` method.
- **void dispatch(int cpu, Ptr<task_struct> prev)**: Called when the CPU's local scheduling queue is empty to fill it by getting a task via `scx_bpf_consume()` from another scheduling queue or by directly dispatching a task via `scx_bpf_dispatch()`.
- **void running(Ptr<task_struct> p)**: Called when a task starts to run on its associated CPU.
- **void stopping(Ptr<task_struct> p, boolean runnable)**: Called when a task stops its execution on a CPU.
- **void enable(Ptr<task_struct> p)**: Called whenever a task starts to be scheduled by the currently implemented scheduler.

Source: https://mostlynerdless.de/blog/2024/09/10/hello-ebpf-writing-a-linux-scheduler-in-java-with-ebpf-15/

II. Project hello-ebpf

Implementations

FIFO Scheduler

The First-In-First-Out (FIFO) scheduler is the first scheduler we'll implement, based on the scx_simple from scx. This scheduler has a pretty simple policy: It schedules the tasks in order of their arrival for a time slice of 20 ms. The advantage is that scheduling decisions are made quickly.

But there is a major drawback: The scheduler isn't fair. Consider two tasks: a task `A` that just runs for a burst of 1 ms and then sleeps and a task B that runs continuously. Even though A uses only 5% of its allotted time slice every time it is scheduled, it will, on average, be scheduled the same number of times as `B`, thus `B` runs 20 times longer on the CPU. We see later why this is a problem.

But first, let's look at the implementation, omitting the user-land and logging code:

```
@BPF(license = "GPL")
public abstract class FIFOScheduler extends BPFProgram
                                    implements Scheduler {

    /**
     * A custom scheduling queue
     */
    static final long SHARED_DSQ_ID = 0;

    @Override
    public int init() {
        // init the scheduling queue
        return scx_bpf_create_dsq(SHARED_DSQ_ID, -1);
    }

    @Override
    public int selectCPU(Ptr<task_struct> p, int prev_cpu,
                         long wake_flags) {
        boolean is_idle = false;
        // let sched-ext select the best CPU
        int cpu = scx_bpf_select_cpu_dfl(p, prev_cpu,
                                         wake_flags,
                                         Ptr.of(is_idle));
        if (is_idle) {
            // directly dispatch to the CPU if it is idle
            scx_bpf_dispatch(p, SCX_DSQ_LOCAL.value(),
                             SCX_SLICE_DFL.value(),0);
        }
        return cpu;
    }

    @Override
    public void enqueue(Ptr<task_struct> p, long enq_flags) {
        // directly dispatch to the selected CPU's local queue
        scx_bpf_dispatch(p, SHARED_DSQ_ID,
                         SCX_SLICE_DFL.value(), enq_flags);
    }

    // ...

    public static void main(String[] args) {
        try (var program =
                 BPFProgram.load(FIFOScheduler.class)) {
            // ...
        }
    }
}
```

But what about a slightly more complex scheduler? Can we write a scheduler that is fairer to tasks that don't run for their whole time slice?

Weighted Scheduler

A simple solution to our fairness is prioritizing tasks according to their actual runtime on the CPU While we're at it, we can also factor in the priority (or weight) of a process by scaling the runtime accordingly. This policy is far better fairness-wise than the FIFO policy, but of course, there is still space improvement. There are reasons why proper schedulers are complex. To learn more, I recommend listening to the Tech Over Tea podcast episode with sched-ext developer David Vernet, which you can find on YouTube or Spotify.

Before I show you the scheduler implementation, I want to bring into focus the properties of the `sched_ext_entity` typed `scx` field of type of the `task_struct`, which are valuable for implementing the described policy (adapted from the C source):

```
@Type
class sched_ext_entity {

    // ...

    // priority of the task
    @Unsigned int weight;

    // ...
    /*
     * Runtime budget in nsecs. This is usually set through
     * scx_bpf_dispatch() but can also be modified directly
     * by the BPF scheduler. Automatically decreased by SCX
     * as the task executes. On depletion, a scheduling event
     * is triggered.
     *
     * This value is cleared to zero if the task is preempted
     * by %SCX_KICK_PREEMPT and shouldn't be used to determine
     * how long the task ran. Use p->se.sum_exec_runtime
     * instead.
     */
    @Unsigned long slice;

    /*
     * Used to order tasks when dispatching to the
     * vtime-ordered priority queue of a dsq. This is usually
     * set through scx_bpf_dispatch_vtime() but can also be
     * modified directly by the BPF scheduler. Modifying it
     * while a task is queued on a dsq may mangle the ordering
     * and is not recommended.
     */
    @Unsigned long dsq_vtime;

    // ...

}
```

Back to the scheduler implementation, the new scheduler is not too dissimilar to the FIFO scheduler but makes full use of the `enqueue` and `dispatch` methods:

```
@BPF(license = "GPL")
public abstract class WeightedScheduler extends BPFProgram
                                        implements Scheduler {

    // current vtime
    final GlobalVariable<@Unsigned Long> vtime_now =
            new GlobalVariable<>(0L);

    /*
     * Built-in DSQs such as SCX_DSQ_GLOBAL cannot be used as
     * priority queues (meaning, cannot be dispatched to with
     * scx_bpf_dispatch_vtime()). We therefore create a
     * separate DSQ with ID 0 that we dispatch to and consume
     * from. If scx_simple only supported global FIFO scheduling,
     * then we could just use SCX_DSQ_GLOBAL.
     */
    static final long SHARED_DSQ_ID = 0;

    @BPFFunction
    @AlwaysInline
    boolean isSmaller(@Unsigned long a, @Unsigned long b) {
        return (long)(a - b) < 0;
    }

    @Override
    public int selectCPU(Ptr<task_struct> p, int prev_cpu,
                         long wake_flags) {
        // same as before
        boolean is_idle = false;
        int cpu = scx_bpf_select_cpu_dfl(p, prev_cpu,
                wake_flags, Ptr.of(is_idle));
        if (is_idle) {
            scx_bpf_dispatch(p, SCX_DSQ_LOCAL.value(),
                    SCX_SLICE_DFL.value(),0);
        }
        return cpu;
    }

    @Override
    public void enqueue(Ptr<task_struct> p, long enq_flags) {
        // get the weighted vtime, specified in the stopping
        // method
        @Unsigned long vtime = p.val().scx.dsq_vtime;

        /*
         * Limit the amount of budget that an idling task can
         * accumulate to one slice.
         */
        if (isSmaller(vtime,
                      vtime_now.get() - SCX_SLICE_DFL.value())) {
            vtime = vtime_now.get() - SCX_SLICE_DFL.value();
        }
        /*
         * Dispatch the task to dsq_vtime-ordered priority
         * queue, which prioritizes tasks with smaller vtime
         */
        scx_bpf_dispatch_vtime(p, SHARED_DSQ_ID,
                               SCX_SLICE_DFL.value(), vtime,
                               enq_flags);
    }

    @Override
    public void dispatch(int cpu, Ptr<task_struct> prev) {
        scx_bpf_consume(SHARED_DSQ_ID);
    }

    @Override
    public void running(Ptr<task_struct> p) {
        /*
         * Global vtime always progresses forward as tasks
         * start executing. The test and update can be
         * performed concurrently from multiple CPUs and
         * thus racy. Any error should be contained and
         * temporary. Let's just live with it.
         */
        @Unsigned long vtime = p.val().scx.dsq_vtime;
        if (isSmaller(vtime_now.get(), vtime)) {
            vtime_now.set(vtime);
        }
    }

    @Override
    public void stopping(Ptr<task_struct> p, boolean runnable) {
        /*
         * Scale the execution time by the inverse of the weight
         * and charge.
         *
         * Note that the default yield implementation yields by
         * setting @p->scx.slice to zero and the following would
         * treat the yielding task
         * as if it has consumed all its slice. If this penalizes
         * yielding tasks too much, determine the execution time
         * by taking explicit timestamps instead of depending on
         * @p->scx.slice.
         */
        p.val().scx.dsq_vtime +=
            (SCX_SLICE_DFL.value() - p.val().scx.slice) * 100
            / p.val().scx.weight;
    }

    @Override
    public void enable(Ptr<task_struct> p) {
        /*
         * Set the virtual time to the current vtime, when the task
         * is about to be scheduled for the first time
         */
        p.val().scx.dsq_vtime = vtime_now.get();
    }

    public static void main(String[] args) {
        try (var program =
                 BPFProgram.load(WeightedScheduler.class)) {
            // ...
        }
    }
}
```

I merged the FIFO and the weighted scheduler implementations into the SampleScheduler class, which you can find on GitHub.

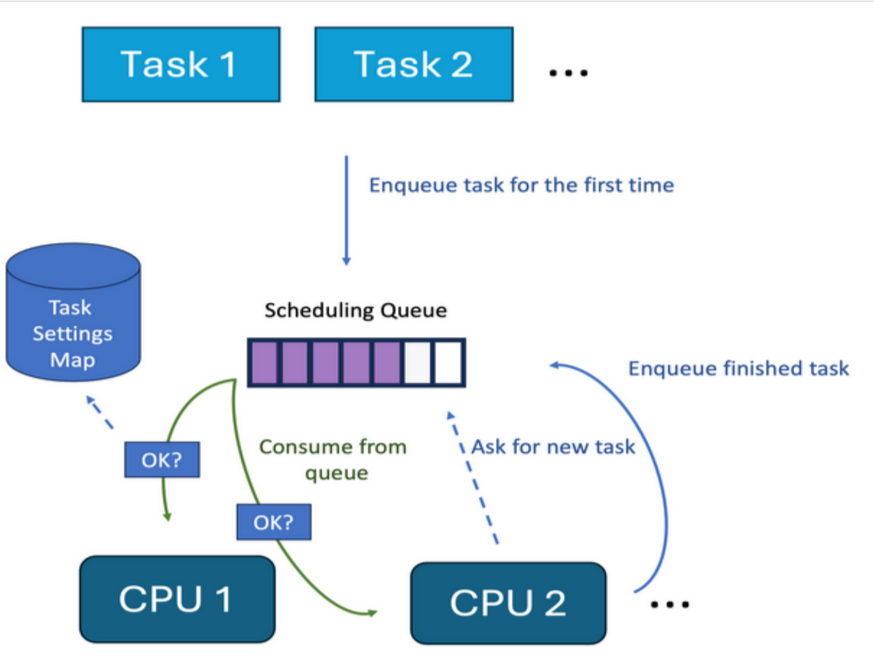
Source: https://mostlynerdless.de/blog/2024/09/10/hello-ebpf-writing-a-linux-scheduler-in-java-with-ebpf-15/

II. Project hello-ebpf

Control task scheduling

Idea

I showed before that we can write schedulers ourselves, so why not use them for this case? The main idea is to only schedule processes in the scheduler that are allowed to be scheduled, according to a task (and task group) settings BPF map:



This is essentially our minimal scheduler, with one slight modification that I show you later.

And yes, the tasks are not stopped immediately, but with a maximum of 5ms scheduling intervals, we only have a small delay.

I implemented all this in the taskcontrol project, which you can find on GitHub. This is where you also find information on the required dependencies and how to install a 6.12 Kernel if you're on Ubuntu 24.10.

https://github.com/parttimenerd/taskcontrol

Taskcontrol

Ever wanted to control which tasks are scheduled on your system via a neat API? This project is for you!

Usage

```
./scheduler.sh
```

Then use the following APIs:

```
GET localhost:PORT/task/{id} to get the status of a task
GET localhost:PORT/task/{id}?stopping=true|false to stop or resume a task
GET localhost:PORT/task/{id}?lotteryPriority=N positive priority for the LotteryScheduler (large
GET localhost:PORT/task/plan/{id}?plan=s10,r10 to set the plan for a task (e.g. 10s running, 10s
GET localhost:PORT/task/plan/{id} to get the current plan for a task
GET localhost:PORT/plans the current plans as JSON

The same for taskGroup (process)
```

Be aware that stopping a task for more than 30s will kill the scheduler.

You can select multiple schedulers via `./scheduler.sh` or set the server port:

```
Usage: scheduler [-hV] [-p=<port>] [-s=<schedulerType>]
A FIFO scheduler with a rest API to stop tasks
  -h, --help          Show this help message and exit.
  -p, --port=<port>   The port to listen on
  -s, --scheduler=<schedulerType>
                      The scheduler to use: fifo, lottery
  -V, --version       Print version information and exit.
```

Currently only the FIFO scheduler is implemented.

Source: /https://mostlynerdless.de/blog/2024/12/03/hello-ebpf-control-task-scheduling-with-a-custom-scheduler-written-in-java-16/

III. Discussion on GraalVM-based eBPF development

1) uBPF on GraalVM

Our previous research work

- Please refer to my previous talks and upcoming follow-ups:

1. "GraalVM-based unified runtime for eBPF & Wasm" at GOTC 2021 (Shenzhen), on GraalVM CE Java11-21.2.0:
   - Successfully ported project BPF-Graal-Truffle(https://github.com/mattmurante/bpf-graal-truffle except for AOT part and some test cases.
   - Successfully built uBPF(https://github.com/iovisor/ubpf) to LLVM bitcode and run on GraalVM.
   - Successfully run some basic tests with the official solution GraalWasm(https://github.com/oracle/graal/tree/master/wasm), but failed with the Polyglot case…

Issues

- Most of the uBPF implementations do not support ARM.
- The GraalVM LLVM toolchain does not work as expected for some cases.

III. Discussion on GraalVM-based eBPF development

2. "Revisiting GraalVM-based unified runtime for eBPF & Wasm" at OpenInfra Days China 2021(Beijing), on GraalVM CE Java11-21.3.0-dev:
   - Failed to run project rBPF(https://github.com/qmonnet/rbpf) together with Solana rBPF(https://github.com/solana-labs/rbpf) on GraalVM.
     May need to hack rustc/cargo as the blocking issue is that Rust projects are not built in a finely-grained way for generating the required LLVM bitcode file against each individual source code file.
   - Successfully make wasm3(https://github.com/wasm3/wasm3) to run on GraalVM.

Issues

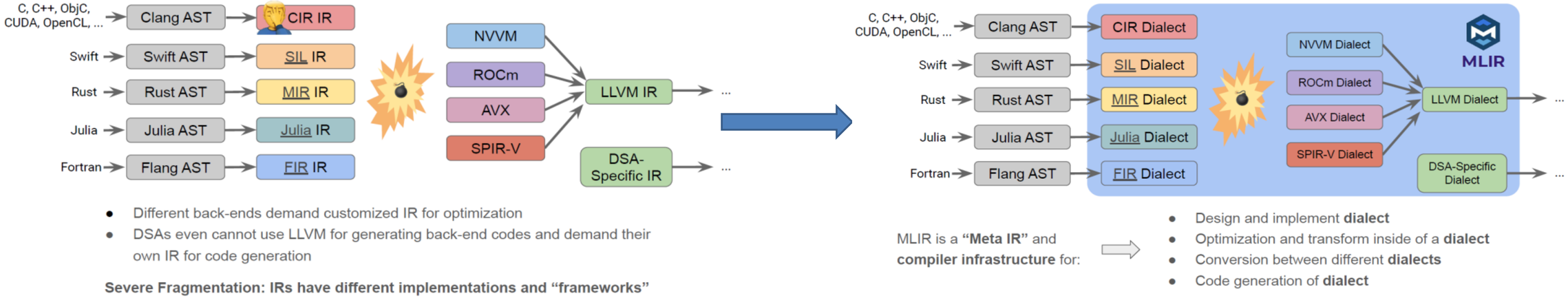
- Rust projects are not GraalVM friendly.
- Confirmed that there is something wrong with the GraalVM LLVM toolchain from the official GraalVM CE releases, especially the linker.

III. Discussion on GraalVM-based eBPF development

2) Natively support eBPF in GraalVM?

From LLVM to MLIR

- **MLIR: "Meta IR" and Compiler Infrastructure**



- Different back-ends demand customized IR for optimization
- DSAs even cannot use LLVM for generating back-end codes and demand their own IR for code generation

Severe Fragmentation: IRs have different implementations and "frameworks"

MLIR is a **"Meta IR"** and **compiler infrastructure** for:

- Design and implement **dialect**
- Optimization and transform inside of a **dialect**
- Conversion between different **dialects**
- Code generation of **dialect**

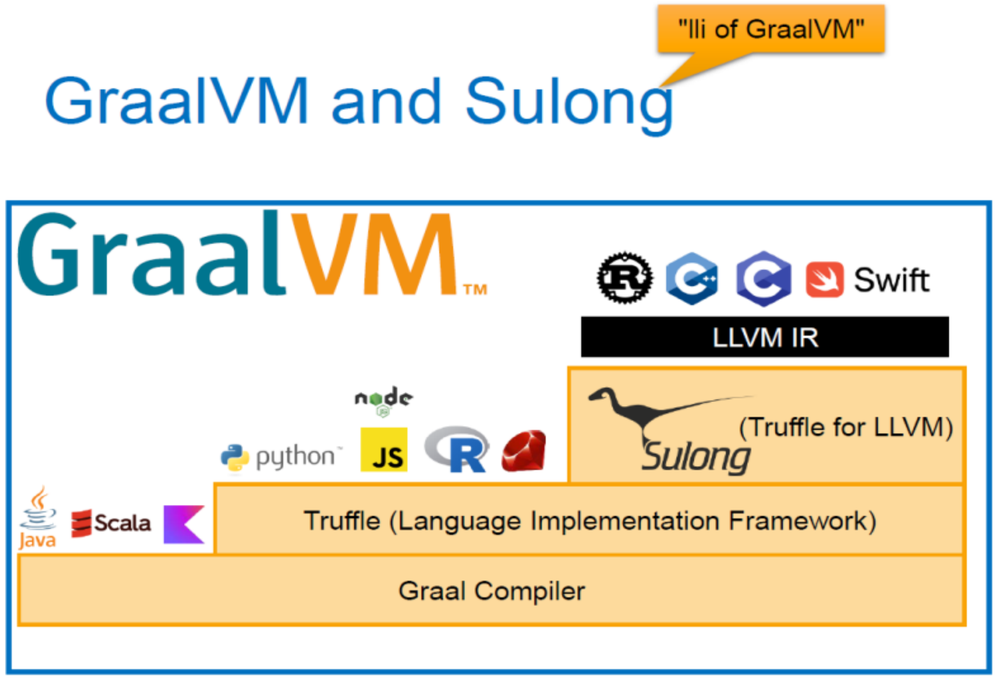
Source: https://hanchenye.com/assets/pdf/Gatech_CIRCT_Slides_Hanchen.pdf

- **But how about a lightweight re-implementation of MLIR/LLVM?**

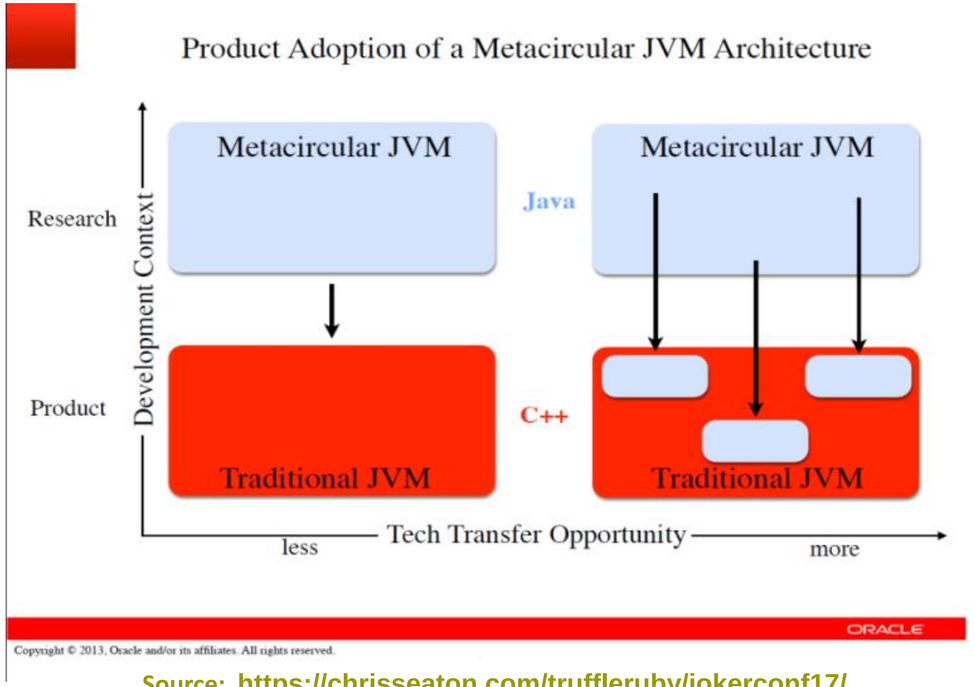
III. Discussion on GraalVM-based eBPF development

Rethinking of GraalVM

- One VM to Rule Them All



Source: https://llvm.org/devmtg/2022-05/slides/2022EuroLLVM-ExtendingSulong.pdf



Source: https://chrisseaton.com/truffleruby/jokerconf17/

- Is it possible to re-implement MLIR/LLVM by GraalVM?

III. Discussion on GraalVM-based eBPF development

3) A customized GraalVM

- **Similar to project Mandrel(https://github.com/graalvm/mandrel) for Quarkus**

  A downstream distribution of the GraalVM community edition. Mandrel's main goal is to provide a native-image release specifically to support Quarkus.

- **Restart GraalVM Updater**

  The GraalVM Updater (gu) has been removed by Oracle:

  https://github.com/oracle/graal/issues/6855

[GR-46219] Remove the GraalVM Updater #6855

Closed **fniephaus** opened this issue on Jun 22, 2023 · 20 comments

fniephaus commented on Jun 22, 2023 • edited by chumer — Member

TL;DR

We plan to remove the GraalVM Updater and provide GraalVM language plugins and other GraalVM extensions in new ways.

Goals

- Installed applications and runtimes are supposed to stay immutable. GraalVM Updater modified the installation, making it hard to work with systems and package managers requiring immutability. We, therefore, will remove GraalVM Updater and find replacements for its use cases with the GraalVM for the JDK 21 release.
- For polyglot embedding, all languages should be usable as Maven dependency.
- Language launcher users should be guided to use standalone distributions instead of using the launchers installed by gu.

All non-language components and their migration:

Component ID	Component Name	How to use in GraalVM for JDK 21+
`jipher`	Jipher JCE Provider	to be included in Oracle GraalVM (not available in GraalVM CE)
`llvm-toolchain`	LLVM.org toolchain	Build from source or usable as part of the Ruby standalone
`native-image-llvm-backend`	Native Image LLVM backend	Build from source
`visualvm`	VisualVM	GitHub release (VisualVM standalone)

Non-Goals

- Remove any of the components themselves
- Introduce a new component installer or rework `gu`

Try to add support for TornadoVM, etc. in the new gu…

III. Discussion on GraalVM-based eBPF development

4) A new GraalVM-based DSL for eBPF?

4.2 DSLs for eBPF

ply

- **https://github.com/iovisor/ply**

A light-weight dynamic tracer for Linux that leverages the kernel's BPF VM in concert with kprobes and tracepoints to attach probes to arbitrary points in the kernel. Most tracers that generate BPF bytecode are based on the LLVM based BCC toolchain. ply on the other hand has no required external dependencies except for `libc`. In addition to `x86_64`, ply also runs on `aarch64`, `arm`, `riscv64`, `riscv32`, and `powerpc`. Adding support for more ISAs is easy.

`ply` follows the Little Language approach of yore, compiling ply scripts into Linux BPF programs that are attached to kprobes and tracepoints in the kernel. The scripts have a C-like syntax, heavily inspired by `dtrace(1)` and, by extension, `awk(1)`.

The primary goals of `ply` are:

- Expose most of the BPF tracing feature-set in such a way that new scripts can be whipped up very quickly to test different hypotheses.
- Keep dependencies to a minimum. Right now Flex and Bison are required at build-time, leaving `libc` as the only runtime dependency. Thus, `ply` is well suited for embedded targets.

If you need more fine-grained control over the kernel/userspace interaction in your tracing, checkout the bcc project which compiles C programs to BPF using LLVM in combination with a python userspace recipient to give you the full six degrees of freedom.

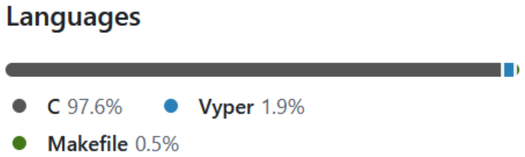
Languages

C 95.9% Makefile 1.1%
Shell 1.0% Yacc 0.8%
Python 0.6% Lex 0.5% M4 0.1%

III. Discussion on GraalVM-based eBPF development

voyant

- **https://github.com/yumosx/voyant**



voyant is domain specific language based on the eBPF instruction set (insn) and system calls;

ulike other eBPF tools, it is designed to be lightweight and easy extendable，There are three aspectes that can account for my option;

1. **Ligtht tool** First of all, no LLVM, indeed LLVM is an exceptional tool for building complier backends, but LLVM is counted in ten millions， the light weight is one of my goals and the rules out LLVM.

2. **Easy**: Due to its lightweight design, our DSL is easier to install and can perform effectively even in resource-constrained environments.

3. **Clear:** The third, our dsl will offer the similarly level of expressivity as general-purpose programming language, also extending the semantics in certain aspectes;

III. Discussion on GraalVM-based eBPF development

4.2 Zig

- https://ziglang.org/

A general-purpose programming language and toolchain for maintaining robust, optimal and reusable software.

⚡ A Simple Language

Focus on debugging your application rather than debugging your programming language knowledge.

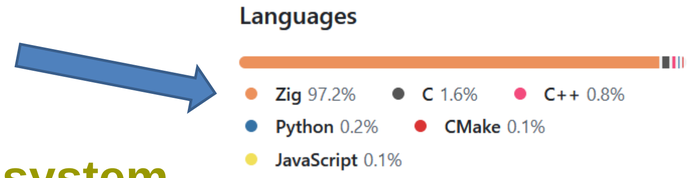
- No hidden control flow.
- No hidden memory allocations.
- No preprocessor, no macros.

⚡ Comptime

A fresh approach to metaprogramming based on compile-time code execution and lazy evaluation.

- Call any function at compile-time.
- Manipulate types as values without runtime overhead.
- Comptime emulates the target architecture.

- https://github.com/ziglang/zig




```
[mydev11@koonuc15x-1 zig-master]$ tree -L 1 src/arch/
src/arch/
├── aarch64
├── arm
├── mips
├── riscv64
├── sparc64
├── wasm
├── x86
└── x86_64
```

zig cc

- https://ziglang.org/learn/build-system
- https://zig.news/kristoff/compile-a-c-c-project-with-zig-368j

C/C++/Zig (4 Part Series)

1. **Compile a C/C++ Project with Zig**
2. Cross-compile a C/C++ Project with Zig
3. Make Zig Your C/C++ Build System
4. Extend a C/C++ Project with Zig

- https://andrewkelley.me/post/zig-cc-powerful-drop-in-replacement-gcc-clang.html
- https://github.com/cross-rs/cross/wiki/Configuration

III. Discussion on GraalVM-based eBPF development

Moving to self-hosted compiler

- https://ziglang.org/news/goodbye-cpp/
- https://github.com/ziglang/zig/issues/16270

make the main zig executable no longer depend on LLVM, LLD, and Clang libraries

This issue is to fully eliminate LLVM, Clang, and LLD *libraries* from the Zig project. The remaining ties to these projects are as follows:

- [ ] completely eliminate dependency on LLD #8726
  - [x] eliminate dependency on LLD for Mach-O #8727
  - [ ] eliminate dependency on LLD for ELF #17749
  - [ ] eliminate dependency on LLD for COFF/PE #17751
  - [ ] eliminate dependency on LLD for WebAssembly #17750
- [ ] LLVM
  - [x] directly output LLVM bitcode rather than using LLVM's IRBuilder API #13265
  - [x] C backend - 1742/1792 tests passing (97%) (**@jacobly0** et al.)
  - [ ] x86 backend - 1721/1792 tests passing (96%) (**@jacobly0**, **@kubkon**, et al.)
  - [ ] wasm backend - 1611/1765 tests passing (91%) (**@Luukdegram**)
  - [ ] aarch64 backend
  - [ ] ~~optimization passes~~ not actually a prerequisite for this proposal. The clang package mentioned below will be able to provide LLVM optimizations as well.
  - [ ] ability to create import libs from def files without LLVM #17807
- [ ] Clang
  - [ ] ~~upstream Aro and use it for translate-c instead of clang #16268~~ not actually a prerequisite for this proposal
  - [ ] ~~When the compiler is built without LLVM, use Aro to compile C code instead of Clang #16269~~ not actually a prerequisite for this proposal
  - [x] C++ source files in zig's repository are built by clang when bootstrapping
    - [x] src/windows_sdk.cpp: port to Zig #15657
  - [ ] compiling assembly files
  - [ ] Compiling C++, Objective-C, Objective-C++, etc. files. Solve with a clang package that can be depended on via the zig build system.
  - [ ] `zig cc` / `zig c++`. Solve with an independent package that serves the exact same use case.
  - [x] make resinator use aro's preprocessor instead of clang #17752
  - [x] make mingw .def.in file parsing use aro's preprocessor instead of clang #17753
- [ ] zig ar: a drop-in llvm-ar replacement #9828

Note that there would still be an LLVM backend for outputting `.bc` files (#13265), but the Zig compiler would lack the capability to compile .bc files into object files. LLVM or Clang would need to be installed and invoked separately for that use case.

In exchange, Zig gains these benefits:

- All our bugs are belong to us.
- The compiler becomes trivial to build from source and to bootstrap with only a C compiler on the host system.
- We stop dealing with annoying problems introduced by Linux distributions and package managers such as Homebrew related to LLVM, Clang, and LLD. There have been and continue to be many.
- The Zig compiler binary goes from about 150 MiB to 5 MiB.
- Compilation speed is increased by orders of magnitude.
- We can implement our own optimization passes that push the state of the art of computing forward.
- We can attract research projects such as alive2
- We can attract direct contributions from Intel, ARM, RISC-V chip manufacturers, etc., who have a vested interest in making our machine code better on their CPUs.

III. Discussion on GraalVM-based eBPF development

4.2.1 Zig for eBPF

zbpf

- **https://github.com/tw4452852/zbpf**

Writing eBPF in Zig. Thanks to Zig's comptime and BTF, we can equip eBPF with strong type system both at comptime and runtime!

Notable advantages when writing eBPF program with `zbpf`

Different available methods based on the type of program's context

Suppose you want to trace the kernel function path_listxattr, and here's its prototype:

```
static ssize_t path_listxattr(const char __user *pathname, char __user *list,
                              size_t size, unsigned int lookup_flags)
```

As you can see, it has 4 input parameters and return type is `ssize_t`. With `ctx = bpf.Kprobe{.name = "path_listxattr"}.Ctx()`, you could retrieve the input parameter with `ctx.arg0()`, `ctx.arg1()`, `ctx.arg2()` and `ctx.arg3()` respectively, and return value with `ctx.ret()`. the type will be consistent with the above prototype. If you try to access a non-existing parameter, e.g. `ctx.arg4()`, you will get a compilation error.

This also applies to `syscall` with `bpf.Ksyscall`, `tracepoint` with `bpf.Tracepoint` and `fentry` with `bpf.Fentry`.

Languages

Zig 99.3% Other 0.7%

No more tedious error handling

When writing in C, you always have to check the error conditions (the return value of the helper function, pointer validation, ...) With `zbpf`, you won't care about the these cases, we handle it under the hood for you, just focus on the business logic.

The following are some examples:

- `bpf.Map` takes care BPF map's `update` and `delete` error.
- `bpf.PerfEventArray` handles event output failure.
- `bpf.RingBuffer` also handles space reservation.
- `bpf.Xdp` validates the pointer for you.

If some error happens, you could get all the information (file, line number, return value ...) you need to debug in the kernel trace buffer:

```
~> sudo bpftool prog tracelog
test-11717   [005] d..21 10990692.273976: bpf_trace_printk: error occur at src/bpf/map.zig:110 return -
```

IV. Wrap-up

Ethos:

eBPF is the future of infrastructure

Source: "Building an Open Source Community One Friend at a Time", Bill Mulligan, FOSDEM 2024.

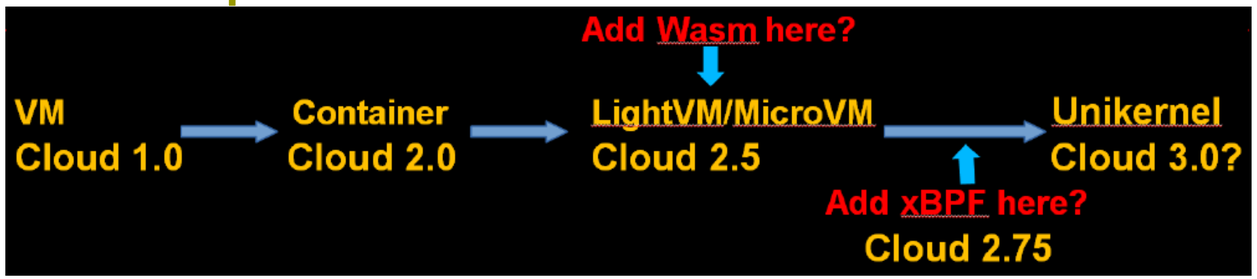
"eBPF is a crazy technology, it's like putting Java~~Script~~ into the Linux kernel

– Brendan Gregg

https://www.facesofopensource.com/brendan-gregg/

Source: "hello-ebpf: Writing eBPF programs directly in Java", Johannes Bechberger, LPC 2024.

- From our point of view:



- You can look forward to our third-round and perhaps the forth-round discussion of "The eBPF-centric new approach for Hyper-Converged Infrastructure & Edge Computing" that will be divided into two series according to different technology roadmap:
  "ARM + xPython + Rust + Lua + GraalVM + …" and
  "RISC-V + xPython + D + Zig + Lua + SmartRuntime…"

IV. Wrap-up

Our RayII Series

- RayII.Rust (got some work done, you may refer to our previous talk "Ray--A Swiss Army Knife for Distributed Computing & AI" at COSCon 2022(online) for the initial discussion, and the follow-ups are coming).
- RayII.Java (at the design and early experimental stage, you may refer to our previous talk "RayII.Java--a Java-based new design and re-implementation of project Ray" at CommunityOverCode Asia 2024 (Hangzhou) for the initial discussion, and the follow-ups are coming).
- RayII.Graal (long-term)
- RayII.Net (you may refer to our previous lightning talk "RayII.Net: a .NET based new design and re-implementation of project Ray" at .NET Conf China 2024(Shanghai) for the initial discussion, and the follow-ups are coming).
- RayII.Acton (you may refer to our previous talk "Acton: a statically typed Python variant with native support for distributed computing" at COSCon 2024(Beijing) for the initial discussion. And we are currently focusing on replace Haskell with Zig in the compiler of Acton)
- RayII.Mojo (long-term, and we are working on a new Python eDSL that combines the necessary features from Mojo, Triton, Acton and more)
- RayII.D/Zig (the initial discussion will come in 2025)
- RayII4HCI (high priority, and the initial work has been discussed in our previous talk "The first exploration of Beyonding Kubernetes" at OpenInfra Summit Asia 2024(Korea), and the follow-ups are coming).
- ...


Q
&
A

THANKS