XuanTie Linux 开发平台使用说明

第一章 T-HEAD Linux System 概述

平头哥玄铁处理器 Linux SDK 介绍

第二章 快速上手指南

2.1 用户说明文档 readme.txt

2.2 OpenSBI 启动引导

2.3 模拟器运行介绍

2.3.1 RISC-V Qemu 命令介绍

2.3.2 C-SKY Qemu 命令介绍

2.4 FPGA调试指南

2.4.1 run.sh - 启动脚本

2.4.2 ddrinit.fpga.txt - DDR初始化脚本

2.4.3 gdbinit.fpga.txt - GDB初始化脚本

2.4.4 fpga.dts.txt- 设备树配置

2.5 Q&A收集

第3章 调试工具指南

3.1 Linux K/Uprobe 使用指南

3.1.1 Kprobe 介绍

基本使用指南

3.1.2 Trace-cmd 和 kernelshark:

3.1.3 Uprobe

3.1.3.1 ftrace 基本使用说明

3.1.3.2 Perf probe 使用说明

3.2 Perf使用指南

3.2.1 Perf 简介

3.2.2 Perf 分析流程

3.2.3 Perf 事件支持

3.2.4 使用 Perf 的准备工作

3.2.5 使用火焰图分析函数热点

3.3 KASAN使用指南
3.3.1 简介
3.3.2 使用方法
3.3.3 出错报告
3.4 Kgdb使用指南
3.4.1 介绍
3.4.2 编译内核
3.5 Lockdep使用指南
3.5.1 简介
3.5.2 使用介绍
第4章 展望未来
4.1 极简开发，轻松上手
4.2 软件丰富，定制灵活
4.3 易于诊断，高效定位
4.4 持续集成，质量保证
4.5 源于开源，回馈社区

第一章 T-HEAD Linux System 概述

平头哥玄铁处理器 Linux SDK 介绍

为了便于客户 CPU 评估，系统集成，快速上手玄铁处理器 Linux 操作系统，平头哥更新了玄铁处理器 Linux 版本，结合 gitlab 开源 CI/CD 系统，为已经发布到开源社区的玄铁架构 CPU 软件生态提供保障。

XuanTie Linux 发布地址：https://github.com/c-sky/buildroot/releases

Why GitHub? Team Enterprise Explore Marketplace Pricing Search Sign in Sign up

c-sky / buildroot

Watch 11 Star 32 Fork 23

Code Issues 14 Pull requests 1 Actions Projects 0 Security 0 Insights

Releases Tags

Latest release

v1.0.9

2854eb8

Compare

XuanTie Linux v1.0.9 Release Note

majun258 released this on 23 Apr

(Test Environment: Ubuntu-16.04-64bit)

c860 linux-4.19 glibc:
https://c-sky.gitlab.io/-/buildroot/-/jobs/511404133/artifacts/output/images/readme.txt
c860 linux-4.9 glibc:
https://c-sky.gitlab.io/-/buildroot/-/jobs/511404129/artifacts/output/images/readme.txt

c810/c807 linux-4.19 glibc:
https://c-sky.gitlab.io/-/buildroot/-/jobs/511404127/artifacts/output/images/readme.txt
c810/c807 linux-4.9 glibc:
https://c-sky.gitlab.io/-/buildroot/-/jobs/511404124/artifacts/output/images/readme.txt

c9xx series compact linux-5.1 glibc:
https://c-sky.gitlab.io/-/buildroot/-/jobs/511404135/artifacts/output/images/readme.txt
c9xx series ehanced linux-5.1 glibc:
https://c-sky.gitlab.io/-/buildroot/-/jobs/511404184/artifacts/output/images/readme.txt

More configs (See 'Build' & 'Build-enhanced' columns):
https://gitlab.com/c-sky/buildroot/pipelines/136092382

重要特性:

- 支持 kprobe/uprobe (c8xx)
- 使能 tracecmd + kernelshark
- 支持 Perf probe -a/x (c8xx)
- 支持 Gcov (c8xx)
- 支持 lockdep (c8xx)
- 支持 rseq syscall (c8xx)

图1.1 T-HEAD Linux System发布地址页面

目前 T-HEAD Linux 支持的玄铁系列 CPU 有:

64位系列 (RISCV):

- C906 单发射，5级流水线，兼容RV64IMA[FD]C[V] 指令集的超高能效处理器；
- C908 双发射，9级按序流水线，C908兼容RVA22标准，支持同构多核架构，支持多cluster；
- C910 三发射，深度乱序，12级流水线，搭载AI向量加速引擎的高性能多核处理器
- C920 三发射，深度乱序，12级流水线，特别加强了矢量运算能力，支持RISC-V V指令扩展；
- R910 三发射，深度乱序，12级流水线，全球首个完整支持GP TEE(全兼容ARM TEE软件生态)的多

核处理器；

32位系列(CSKY):

- C860 三发射，深度乱序，12级流水线，搭载AI加速引擎的高性能多核处理器
- C810 双发射，乱序，10级流水线，高性能超标量处理器
- C807 双发射，乱序，8级流水线，面向实时控制领域的高性能处理器。

第二章 快速上手指南

对于嵌入式开发者来说，若有一套易于上手的开发环境和操作简单的系统，是可以节省很多时间的。玄铁处理器 Linux 开发平台基于 buildroot.org 建立，帮助开发者快速构建玄铁 CPU Linux开发平台 的学习和实验环境，快速集成开源组件。

玄铁CPU Linux 开发平台融入代码仓库的动态 readme.txt 设计，将上手指引精确到每一个 config，逐行精简命令，一目了然（每一个步骤不超过 10 条命令，支持 'ctrl c' + 'ctrl v' 秒贴执行，极速上手）。

访问 https://github.com/c-sky/buildroot/releases ，下面我们将从您点击的目标 CPU readme.txt 开始，最快用时不到15秒，完成快速上手，查看内核启动信息。

2.1 用户说明文档 readme.txt

readme.txt 是通过 SDK 针对每款配置生成的，精确指引用户快速上手对应配置，其中包含以下8个部分:

- **(All is tested on 64bit-ubuntu 16.04)**

  环境说明，所有发布均使用 ubuntu 16.04 docker 环境编译与测试。用户在自己环境中运行，可能会遇到 wget, tar 等基础程序和库缺少的问题，请自行按提示安装对应程序包。如果不想逐个安装工具包，可以直接使用我们 CI 环境的 docker 镜像: 点此链接。

- **Quick Start for qemu run**

  指导用户使用云端预先编译好的二进制程序，直接运行基于 Qemu 模拟器的 Linux 环境。脚本内容包含下载、解压、运行三个部分，用户可以直接复制、粘贴该段内容到 Host 终端中运行。具体请参考 *2.3 模拟器运行介绍*。

- **Build linux kernel**

  指导用户单独编译对应的内核源码，脚本从 kernel.org 中下载标准内核源码包，再打上我们的差异补丁（这样做是因为内核源码包太大，为了减少 CI 系统的存储负担，采用 patch 差异保存的方式）。该补丁包含了编译前的配置信息，使用后即可直接编译，确保生成镜像与云端 CI 环境一致。

- **Build buildroot**

  指导用户编译制定版本、配置的 buildroot，也是云端 CI 环境使用的编译命令。虽然编译 buildroot 的步骤非常简单，但是由于要下载各个开源软件包，所需时间较长，同时对客户的网络环境有较高的要求。

- **Run with hardware**

  T-HEAD Linux 发布的代码和镜像有很强的适配性，同一个 Linux Image 通过配合不同的 dts 可以运行在不同的硬件平台上，包括 QEMU 模拟器也是一种硬件平台。具体请参考 *2.4 FPGA调试指南*。

- **How to get dmesg without uart**

  在没有串口的情况下，我们依然可以调试 Linux。利用 JTAG 直接获取 dmesg 信息，只需 CPU + 内存，不借助其他外设。

Shell 复制代码

```shell
wget -nc https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/admin-guide/kdump/gdbmacros.txt # 获取著名 kernel.org gdb 调试脚本
```

在 gdb 里加载 vmlinux 符号表，加载 gdbmacro.txt 脚本，运行 dmesg

Shell 复制代码

```shell
file linux/vmlinux
file linux/Documentation/admin-guide/kdump/gdbmacros.txt
dmesg
```

gdbmacro.txt 还包含很多其他调试命令，请自行学习。

- **Gitlab-CI url**

  提供了当前 readme.txt 对应版本和配置的编译结果（云端 CI 存储的地址）。readme.txt 中 wget 的内容均来自于此，可以点击其中的 *Download artifacts archive* ，一次性下载全部 artifacts.zip 包。解压后进入 artifacts 目录，此时粘贴复制 readme.txt 脚本，就不会再wget下载了。(wget 的参数 -nc 即 存在了就不再下载)

- **Versions**

  提供整体和各个模块的版本信息，buildroot SDK 内的所有组件的版本均被 buildroot 顶层版本号冻结，确保版本可回溯。

2.2 OpenSBI 启动引导

T-HEAD Linux 默认以 OpenSBI 来启动引导。一般来说，无需对 OpenSBI 做特殊处理，buildroot 已经帮我们做好了相关的启动引导程序。如需修改参数，在ddrinit.fpga.txt、gdbinit.fpga.txt 或者 fpga.dts.txt 三个文件中修改即可。

OpenSBI 简介：

玄铁 CPU提供了至少三种工作模式，包括 Machine Mode, Supervisor Mode 和 User Mode。

- Supervisor Mode : 用于运行操作系统；
- User Mode : 运行用户态应用程序；
- Machine Mode : 运行 Bootloader 并装载和执行 OS。

为了兼容不同模式的运行需求，OpenSBI 支持三种类型的 Firmware，分别为：

- dynamic：从上一级 Boot Stage 获取下一级 Boot Stage 的入口信息，以 `struct fw_dynamic_info` 结构体通过 `a2` 寄存器传递。
- jump：假设下一级 Boot Stage Entry 为固定地址，直接跳转过去运行。
- payload：在 jump 的基础上，直接打包进来下一级 Boot Stage 的 Binary。

OpenSBI下一级通常是 Bootloader 或 OS，比如 U-Boot，Linux。详细介绍可阅读官网或者 https://github.com/c-sky/opensbi#readme

2.3 模拟器运行介绍

模拟器基于开源 Qemu 项目开发，提供 T-HEAD CPU 基础功能验证环境，支持 2 种架构，8 款 CPU:

- RISC-V 架构: C906/C908/C910/C920/R910

- C-SKY 架构: 807/810/860

请参考 readme.txt 中 *Quick Start for qemu run* 部分，模拟器包含在 toolchain_thead_xxx.tar.xz 中（一般解压到 host 目录下）。

2.3.1 RISC-V Qemu 命令介绍

Shell 复制代码

```shell
LD_LIBRARY_PATH=./host/lib ./host/csky-qemu/bin/qemu-system-riscv64 -M virt -kernel fw_jump.elf -device loader,file=Image,addr=0x80200000 -append "rootwait root=/dev/vda ro" -drive file=rootfs.ext2,format=raw,id=hd0 -device virtio-blk-device,drive=hd0 -nographic -smp 1
```

- LD_LIBRARY_PATH=./host/lib：增加指定运行库；
- qemu-system-riscv64：RISC-V QEMU 可执行命令；
- -M virt： QEMU 标准，指定板型号，这里是著名的 virt machine；
- -kernel fw_jump.elf：指定 RISC-V Opensbi 为加载程序，此为 RISC-V QEMU 标准；
- -device loader, file=Image,addr=0x80200000： QEMU 标准 loader 设备，将 Image 文件，加载到 0x80200000 处；
- -append "rootwait root=/dev/vda ro"：指定 linux cmdline；
- -drive file=rootfs.ext2,format=raw,id=hd0： 指定 rootfs.ext2 为磁盘设备，对应 cmdline 中 root=/dev/vda 设置该设备为根文件系统；
- -device virtio-blk-device,drive=hd0：同上确定 -drive 中 id=hd0 的具体设备类型；
- -nographic: 禁用图形 framebuffer；
- -smp 1: 指定 cpu 数量；

2.3.2 C-SKY Qemu 命令介绍

Shell 复制代码

```shell
./host/csky-qemu/bin/qemu-system-cskyv2 -M virt -cpu c860v -kernel Image -nographic -append "console=ttyS0,115200 rdinit=/sbin/init rootwait root=/dev/vda ro" -drive file=rootfs.ext2,format=raw,id=hd0 -device virtio-blk-device,drive=hd0 -smp 1
```

- qemu-system-cskyv2: C-SKY QEMU 可执行命令

- –M virt: QEMU 标准，指定板型号，这里是著名的 virt machine
- –cpu c807/c810/c860: 指定 cpu 型号
- –kernel Image: 指定 C–SKY Linux 镜像，不需要 bootloader 辅助启动
- –append "console=ttyS0,115200 rdinit=/sbin/init rootwait root=/dev/vda ro": 指定 linux cmdline
- –drive file=rootfs.ext2,format=raw,id=hd0: 指定 rootfs.ext2 为磁盘设备，对应 cmdline 中 root=/dev/vda 设置该设备为根文件系统
- –device virtio–blk–device,drive=hd0: 同上确定 –drive 中 id=hd0 的具体设备类型
- –nographic: 禁用图形 framebuffer
- –smp 1: 指定 cpu 数量

2.4 FPGA调试指南

很多客户拿到我们的 CPU 后，第一时间会做 FPGA 集成测试。根据本指引，客户无需编译软件，即可直接用 run.sh 启动 Linux。而对硬件方面的要求，只要是 CPU + DDR 内存即可。

使用 run.sh 运行 Linux 包含以下4步：

- run.sh – 启动脚本
- ddrinit.fpga.txt – DDR初始化脚本
- gdbinit.fpga.txt – GDB初始化脚本
- fpga.dts.txt – 设备树配置

FPGA 即客户自定义的最小 Soc，需要客户自己建立 ddrinit.fpga.txt、gdbinit.fpga.txt、fpga.dts.txt 文件。

2.4.1 run.sh – 启动脚本

运行 run.sh 需要事先启动 DebugServer Console, 连接CKLink 与FPGA，之后运行 DebugServerConsole, 开启127.0.0.1:1025 端口

Shell 复制代码

```shell
run.sh <ip地址:端口> <硬件板类型> <cpu数量>

举例:
run.sh 127.0.0.1:1025 fpga 1
```

run.sh 脚本完成以下4项内容:

- 获取 rootfs.cpio.gz 的大小，并以此设置设备树文件的 initrd-start, initrd-end 字段
- 按需要运行 ddrinit.[board].txt 初始化程序，配置时钟，DDR
- 运行 gdbinit.[board].txt 下载 rootfs.cpio.gz, hw.dtb, Image 到内存
- 设置 pc 到指定内存执行

2.4.2 ddrinit.fpga.txt – DDR初始化脚本

Shell 复制代码

```shell
# cat ddrinit.ice_fpga.txt
# Set gdb environment
set height 0

# Don't confirm when quit
set confirm off

set *0x3fff78040 = 0xffff # xillinx FPGA 初始化 DDR PPL
shell $(sleep 3)
```

ddrinit 负责初始化 DDR 内存，PPL 等 Soc 基础设置，请客户根据需求自行调整。

2.4.3 gdbinit.fpga.txt – GDB初始化脚本

Shell 复制代码

```shell
# cat gdbinit.ice_fpga.txt
set confirm off
set height 0

# Invalidate & Clear IBP BTB BHT ICache & DCache
set $mcor = 0x70013

# Enable L2 Cache
set $mccr2 = 0xe0010009

# Enable L1 Cache
set $mhcr = 0x11ff

# Enable CPU Features
set $mxstatus = 0x638000
set $mhint = 0x16e30c

# 0x00000000 ~ 0x2 00000000 DDR, NAPOT rwx
set $pmpaddr0 = 0x0 >> 2 | ((0x200000000 - 1) >> 3)

# 0x3 f0000000 ~ 0x3 f8000000 plic, NAPOT rw
set $pmpaddr1 = 0x3f0000000 >> 2 | ((0x8000000 - 1) >> 3)

# 0x3 f8000000 ~ 0x4 00000000 peripherals, NAPOT rw
set $pmpaddr2 = 0x3f8000000 >> 2 | ((0x8000000 - 1) >> 3)

# 0x00000000 ~ 0x10000000 NAPOT no access
set $pmpaddr6 = 0x00000000 >> 2 | ((0x10000000 - 1) >> 3)

# 0x10000000 ~ 0x10000000000 TOR no access
set $pmpaddr7 = 0xffffffffff >> 2

# Be care! we must put background deny entries in the end of
# pmpaddrx with lowest priority and set lock bit for m state deny.
# Access needn't lock bit for the m state.
set $pmpcfg0 = 0x88980000001b1b1f

set $opensbi_addr = 0x00000000
set $vmlinux_addr = $opensbi_addr + 0x00200000
set $rootfs_addr  = $opensbi_addr + 0x02000000
set $dtb_addr  = $rootfs_addr - 0x00100000

# Load rootfs
restore ../rootfs.cpio.gz binary $rootfs_addr

```

```
# Load dtb
restore hw.dtb binary $dtb_addr
set $a1 = $dtb_addr

# Load kernel
restore ../Image binary $vmlinux_addr

# Load opensbi
restore fw_jump_0G.bin binary $opensbi_addr
set $pc = $opensbi_addr
```

gdbinit 是 CPU 通用设置，一般不需要客户修改。对于 910 客户依然需要正确设置 PMP，PMP 是对物理内存的权限管理寄存器，根据自身 IP 地址分布进行设置，一般需要设置 DDR 区域、PLIC 区域、外设 IO 区域，根据 RISCV 规范可以用 NAPOT 和 TOR 两种方式。

2.4.4 fpga.dts.txt– 设备树配置

Shell 复制代码

```
/dts-v1/;
/ {
        model = "T-HEAD c910 fpga";
        compatible = "thead,c910_fpga";
        #address-cells = <2>;
        #size-cells = <2>;

        memory@0 {
                device_type = "memory";
                reg = <0x0 0x200000 0x0 0xffe00000>;
        };

        cpus {
                #address-cells = <1>;
                #size-cells = <0>;
                timebase-frequency = <6250000>;
                cpu@0 {
                        device_type = "cpu";
                        reg = <0>;
                        status = "okay";
                        compatible = "riscv";
                        riscv,isa = "rv64imafdcvsu";
                        mmu-type = "riscv,sv39";
                        cpu0_intc: interrupt-controller {
                                #interrupt-cells = <1>;
                                compatible = "riscv,cpu-intc";
                                interrupt-controller;
                        };
                };
                cpu@1 {
                        device_type = "cpu";
                        reg = <1>;
                        status = "fail";
                        compatible = "riscv";
                        riscv,isa = "rv64imafdcvsu";
                        mmu-type = "riscv,sv39";
                        cpu1_intc: interrupt-controller {
                                #interrupt-cells = <1>;
                                compatible = "riscv,cpu-intc";
                                interrupt-controller;
                        };
                };
        };
        soc {
                #address-cells = <2>;
```

```
46                  #size-cells = <2>;
47
48                  compatible = "simple-bus";
49                  ranges;
50
51                  intc: interrupt-controller@3f0000000 {
52                          #interrupt-cells = <1>;
53                          compatible = "riscv,plic0";
54                          interrupt-controller;
55                          interrupts-extended = <
56                                  &cpu0_intc  0xffffffff &cpu0_intc  9
57                                  &cpu1_intc  0xffffffff &cpu1_intc  9
58                                  >;
59                          reg = <0x3 0xf0000000 0x0 0x04000000>;
60                          reg-names = "control";
61                          riscv,max-priority = <7>;
62                          riscv,ndev = <80>;
63                  };
64
65                  dummy_apb: apb-clock {
66                          compatible = "fixed-clock";
67                          clock-frequency = <50000000>;
68                          clock-output-names = "dummy_apb";
69                          #clock-cells = <0>;
70                  };
71
72                  serial@3fff73000 {
73                          compatible = "snps,dw-apb-uart";
74                          reg = <0x3 0xfff73000 0x0 0x400>;
75                          interrupt-parent = <&intc>;
76                          interrupts = <23>;
77                          clocks = <&dummy_apb>;
78                          clock-names = "baudclk";
79                          reg-shift = <2>;
80                          reg-io-width = <4>;
81                  };
82          };
83
84          chosen {
85                  bootargs = "console=ttyS0,115200 crashkernel=256M-:128M";
86                  linux,initrd-start = <0x0 0x2000000>;
87                  linux,initrd-end = <0x0 0x0>;
88                  stdout-path = "/soc/serial@3fff73000:115200";
89          };
    };
```

以上就是最小系统 dts 配置(最多双核)，需要注意几点：

- cpus -> timebase-frequency 用于设置 cpu 紧耦合 timer 定时器的频率
- riscv,isa = "rv64imafdcvsu" 用于设置 cpu ISA 类型，如果 C910 不配置 vector 模块，请把 v 字母去掉
- intc: interrupt-controller -> compatible = "riscv,plic0" 需要根据 Soc 地址分布，正确设置它的 IO 地址空间
- dummy_apb: apb-clock 是设置提供串口 IP 的频率，这里取 apb 总线频率，串口驱动根据该设置，换算出正确的波特率

设置好以上三个文件，就可以尝试执行 `bash run.sh 127.0.0.1:1025 fpga 1` 了。如果串口没打印，可以参考 *2.1 How to get dmesg without uart* 部分，利用 JTAG 获取 dmesg。

2.5 Q&A收集

Q: 编译 buildroot 下载太慢

A: 一般为网速、翻墙问题，需自行解决。完成第一下载后，会把开源软件包保存在 dl 目录下，可以把 dl 目录放置在公共目录，设置 BR2_DL_DIR= 环境变量到该目录，下次编译 buildroot 就会先从该目录获取源码包，避免重复下载。

Q: readme.txt 的 wget 速度太慢

A: wget 是单线程下载，速度不如浏览器，在 readme.txt -> Gitlab-CI url 章节提供了云端编译镜像的存储位置。在浏览器打开后，点击里面的 *Download artifacts archive* 一次性下载全部 artifacts.zip 包，解压后进入 artifacts 目录，此时粘贴复制 readme.txt 脚本，就不会再wget下载了。（wget 的参数 -nc 即 存在了就不再下载）

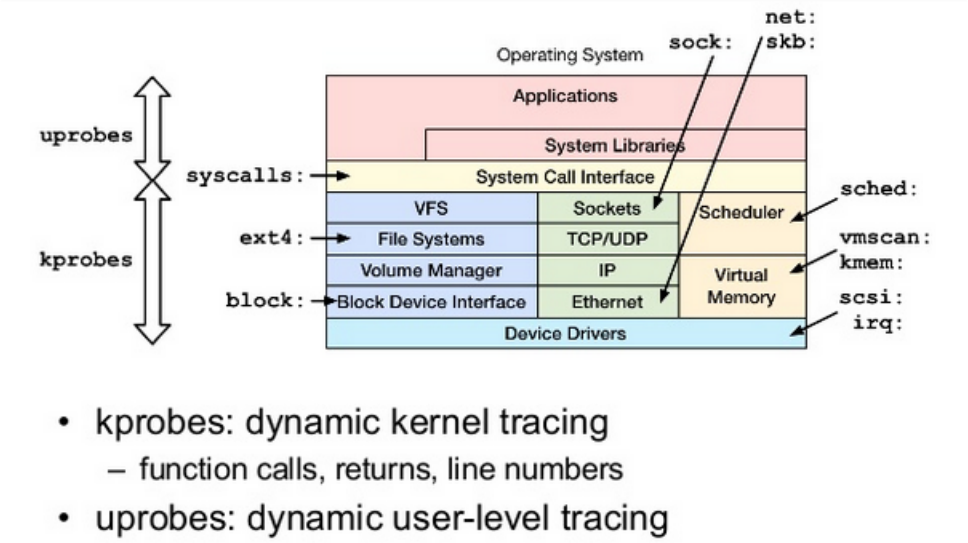
Q: Windows Subsystem for Linux (WSL) ubuntu 环境运行 Qemu 出现: *ERROR: could not open HSTS store at '/home/book/.wget-hsts'* 和 *while loading shared libraries: libbrlapi.so.0.6: cannot open shared object file*

A: chmod 660 ~/.wget-hsts（修改.wget-hsts的权限），Google 搜索 libnettle 6.0 和 libpng12版本的 deb 文件，用 dpkg -i 手动安装。

第3章 调试工具指南

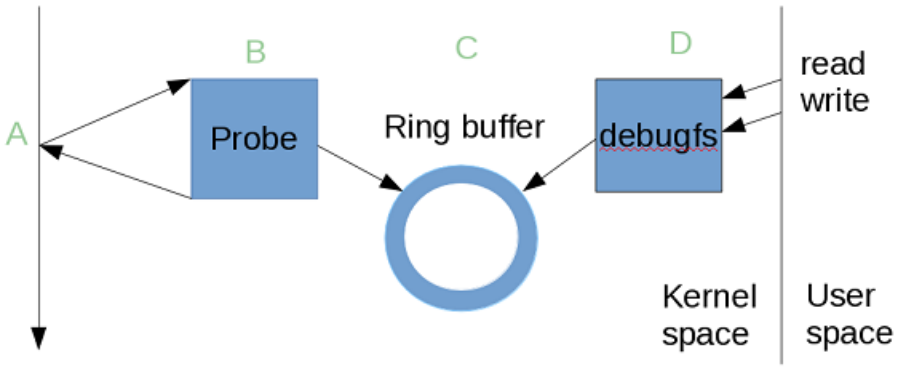
3.1 Linux K/Uprobe 使用指南

经过长期的发展，`kprobes/uprobes` 机制在事件(events)的基础上分别为 内核态 和 用户态 提供了追踪调试的功能，这也构成了 tracepoint 机制的基础。后期很多工具，比如 `perf_events`，`ftrace` 等都是在其基础上演化而来。由 Brendan Gregg 提供的资料来看，`kprobes/uprobes` 在 Linux 动态追踪层面起到了**基石的作用**, 如下所示:



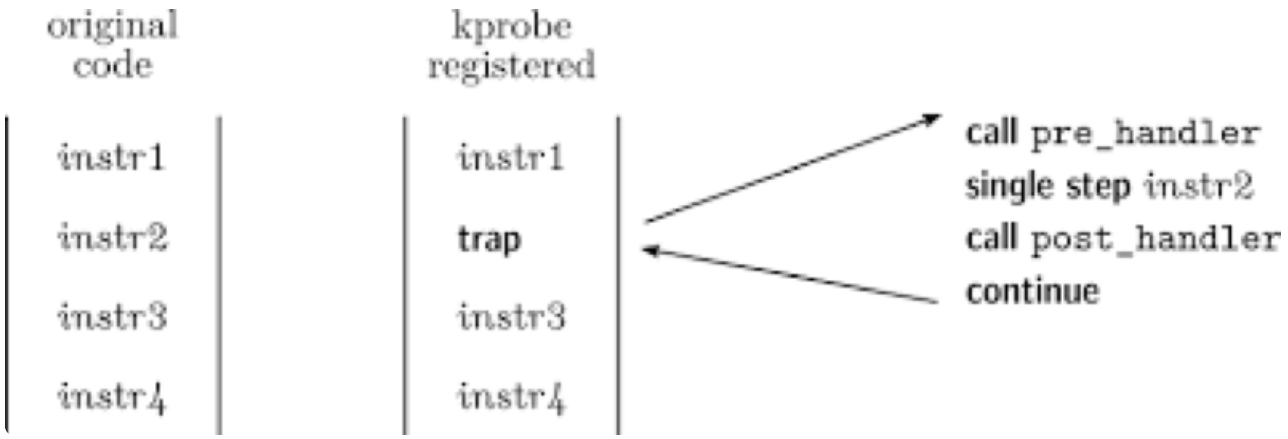
引用: https://blog.arstercz.com/introduction_to_linux_dynamic_tracing/

Kprobe 和 Uprobe 均可以通过 ftrace 的 /sys/debug/tracing interface (基于 debugfs 的用户空间层面的 API) 执行各种跟踪和分析。虽然 ftrace 的内部是复杂的, 不过输出的信息却以简单明了为主，更详细的使用示例可以参考 *Debugging the kernel using Ftrace* 。如下图所示，为 ftrace 的大致原理。



3.1.1 Kprobe 介绍

Kprobe是Linux内核的一个重要特性，是其他的内核调试工具（Perf, systemtap）的“基础设施”。同时内核 BPF 也依赖 Kprobe，它利用指令桩原理，截获指令流，并在指令执行前后插入hook函数：



如果需要知道内核函数是否被调用、被调用上下文、入参以及返回值，比较简单的方法是加printk，但是效率低。

利用Kprobe技术，用户可以自定义回调函数，可以在几乎所有的函数中动态插入探测点。

当内核执行流程执行到指定的探测函数时，会调用该回调函数。此时用户即可收集所需的信息了，同时内核最后还会回到原本的正常执行流程。如果用户已经收集足够的信息，不再需要继续探测，同样可以动态的移除探测点。

Kprobes技术包括的2种探测手段分别时kprobe 和 kretprobe:

- kprobe是最基本的探测方式，是实现后两种的基础，它可以在任意的位置放置探测点（就连函数内部的某条指令处也可以），它提供了探测点的调用前、调用后和内存访问出错3种回调方式，分别是pre_handler、post_handler和fault_handler，其中pre_handler函数将在被探测指令被执行前回调，post_handler会在被探测指令执行完毕后回调（注意不是被探测函数），fault_handler会在内存访问出错时被调用。
- retprobe从名字中就可以看出其用途了，它同样基于kprobe实现，用于获取被探测函数的返回值。

基本使用指南

- 开启内核：

Bash 复制代码

```bash
Symbol: FTRACE [=y]
  Type  : boolean
  Prompt: Tracers
  Location:
  (5) -> Kernel hacking
  Defined at kernel/trace/Kconfig:132
  Depends on: TRACING_SUPPORT [=y]

Symbol: KPROBE_EVENT [=y]
  Type  : boolean
  Prompt: Enable kprobes-based dynamic events
  Location:
  -> Kernel hacking
  (1)   -> Tracers (FTRACE [=y])
  Defined at kernel/trace/Kconfig:405
  Depends on: TRACING_SUPPORT [=y] && FTRACE [=y] && KPROBES [=y] && HAVE_REGS_AND_STACK_ACCESS_API [=y]
  Selects: TRACING [=y] && PROBE_EVENTS [=y]

Symbol: HAVE_KPROBES_ON_FTRACE [=y]
  Type  : boolean
  Defined at arch/Kconfig:183
  Selected by: csky [=y]

Symbol: KPROBES_ON_FTRACE [=y]
  Type  : boolean
```

```
Defined at arch/Kconfig:79

Depends on: KPROBES [=y] && HAVE_KPROBES_ON_FTRACE [=y] && DYNAMIC_FTRACE_WITH_REGS [=y]
```

- 终端运行:

首先通过 mount 获得 ftrace debug 接口，然后通过 kprobe_evets 注册你需要 probe 的内核函数，在 tracing/events/kprobes// 下可以控制该 kprobe 函数的 开启和关闭

Shell 复制代码

```shell
# mount -t debugfs nodev /sys/kernel/debug/

# echo 'p:myprobe _do_fork dfd=%a0 filename=%a1 flags=%a2 mode=+4($stack)' > /sys/kernel/debug/tracing/kprobe_events
# echo 'r:myretprobe _do_fork $retval' >> /sys/kernel/debug/tracing/kprobe_events

# echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
# echo 1 > /sys/kernel/debug/tracing/events/kprobes/myretprobe/enable

# cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 8/8 #P:1
#
# _-----=> irqs-off
# / _----=> need-resched
# | / _---=> hardirq/softirq
# || / _--=> preempt-depth
# ||| / delay
# TASK-PID CPU# |||| TIMESTAMP FUNCTION
# | | | |||| | |
swapper/0-1 [000] dn.. 2.488544: Unknown type 599
swapper/0-1 [000] dn.. 2.489270: Unknown type 600
sh-121 [000] d... 408.113780: myprobe: (_do_fork+0x0/0x3ac) dfd=0xbe485f20 filename=0x0 flags=0x0 mode=0x0
sh-121 [000] d... 408.117058: myretprobe: (sys_clone+0xa6/0xac <- _do_fork) arg1=0x82
sh-121 [000] d... 409.816850: myprobe: (_do_fork+0x0/0x3ac) dfd=0xbe485f20 filename=0x0 flags=0x0 mode=0x0
sh-121 [000] dn.. 409.817539: myretprobe: (sys_clone+0xa6/0xac <- _do_fork) arg1=0x83
sh-121 [000] d... 411.202079: myprobe: (_do_fork+0x0/0x3ac) dfd=0xbe485f20 filename=0x0 flags=0x0 mode=0x0
sh-121 [000] d... 411.202750: myretprobe: (sys_clone+0xa6/0xac <- _do_fork) arg1=0x84
```

社区kprobe文档非常完善:

https://www.kernel.org/doc/Documentation/kprobes.txt

https://lwn.net/Articles/410200/

3.1.2 Trace-cmd 和 kernelshark:

Trace-cmd 是一个基于 ftrace 的用户态前端命令行工具，它的仓库在:

git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git

很多发行版，都有 trace-cmd 的包，完善的 man 手册。具体使用参考:
https://lwn.net/Articles/410200/

3.1.3 Uprobe

3.1.3.1 ftrace 基本使用说明

先准备一个应用程序，测试代码:

Shell 复制代码

```shell
#include <stdio.h>
#include <unistd.h>

static void
print_curr_state_one(void)
{
printf("This is the print current state one function\n");
}


static void
print_curr_state_two(void)
{
printf("This is the print current state two function\n");
}


int main() {
  while(1) {
    print_curr_state_one();
    sleep(1);
    print_curr_state_two();
          }
}
```

编译并获取可执行文件和反汇编 (9 系列请使用 riscv-linux-gcc):

Shell 复制代码

```shell
→  csky-linux-gcc test.c -o test
→  linux-next git:(linux-next-ftrace-kprobe-uprobe-simlutate-insn) ✗ ../artifacts/output_860_next/images/host/bin/csky-linux-objdump -S test

00008518 <print_curr_state_one>:
8518:       1422            subi            sp, sp, 8
851a:       dd0e2000        st.w            r8, (sp, 0)
851e:       ddee2001        st.w            r15, (sp, 0x4)
8522:       6e3b            mov             r8, sp
8524:       1006            lrw             r0, 0x8698      // 853c <print_curr_state_one+0x24>
8526:       eae00007        jsri            0x0     // from address pool at 0x8540
852a:       6c00            or              r0, r0
852c:       6fa3            mov             sp, r8
852e:       d9ee2001        ld.w            r15, (sp, 0x4)
8532:       d90e2000        ld.w            r8, (sp, 0)
8536:       1402            addi            sp, sp, 8
8538:       783c            rts
853a:       0000            .short  0x0000
853c:       00008698        .long   0x00008698
8540:       00000000        .long   0x00000000
00008544 <print_curr_state_two>:
8544:       1422            subi            sp, sp, 8
8546:       dd0e2000        st.w            r8, (sp, 0)
854a:       ddee2001        st.w            r15, (sp, 0x4)
854e:       6e3b            mov             r8, sp
8550:       100d            lrw             r0, 0x86c8      // 8584 <main+0x1c>
8552:       eae0000e        jsri            0x0     // from address pool at 0x8588
8556:       6c00            or              r0, r0
8558:       6fa3            mov             sp, r8
855a:       d9ee2001        ld.w            r15, (sp, 0x4)
855e:       d90e2000        ld.w            r8, (sp, 0)
8562:       1402            addi            sp, sp, 8
8564:       783c            rts
...
00008568 <main>:
8568:       1422            subi            sp, sp, 8
856a:       dd0e2000        st.w            r8, (sp, 0)
856e:       ddee2001        st.w            r15, (sp, 0x4)
8572:       6e3b            mov             r8, sp
8574:       e3ffffd2        bsr             0x8518  // 8518 <print_curr_state_one>
```

```
8578:       3001            movi            r0, 1
857a:       eae00006        jsri            0x0     // from address pool at 0x8590
857e:       e3ffffe3        bsr             0x8544  // 8544 <print_curr_state_two>
8582:       07f9            br              0x8574  // 8574 <main+0xc>

➜  linux-next git:(linux-next-ftrace-kprobe-uprobe-simlutate-insn) ✗ ../artifacts/output_860_next/images/host/bin/csky-linux-readelf -S test
There are 28 section headers, starting at offset 0x1cd8:
Section Headers:
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .interp           PROGBITS        00008134 000134 00000d 00   A  0   0  1
[ 2] .note.ABI-tag     NOTE            00008144 000144 000020 00   A  0   0  4
[ 3] .hash             HASH            00008164 000164 00003c 04   A  4   0  4
[ 4] .dynsym           DYNSYM          000081a0 0001a0 0000a0 10   A  5   1  4
[ 5] .dynstr           STRTAB          00008240 000240 000098 00   A  0   0  1
[ 6] .gnu.version      VERSYM          000082d8 0002d8 000014 02   A  4   0  2
[ 7] .gnu.version_r    VERNEED         000082ec 0002ec 000020 00   A  5   1  4
[ 8] .rela.dyn         RELA            0000830c 00030c 00009c 0c   A  4   0  4
[ 9] .init             PROGBITS        000083b0 0003b0 000030 00  AX  0   0 16
[10] .text             PROGBITS        000083e0 0003e0 000282 00  AX  0   0 16
```

根据程序反汇编编插 uprobe 桩

```shell
echo 'p:enter_current_state_one /root/test:0x518 arg0=%a0 lr=%lr' >> /sys/kernel/debug/tracing/uprobe_events
echo 'r:exit_current_state_one /root/test:0x518 arg0=%a0' >> /sys/kernel/debug/tracing/uprobe_events
echo 'p:enter_current_state_two /root/test:0x544 arg0=%a0 lr=%lr' >> /sys/kernel/debug/tracing/uprobe_events
echo 'r:exit_current_state_two /root/test:0x544 arg0=%a0' >> /sys/kernel/debug/tracing/uprobe_events
echo 1 > /sys/kernel/debug/tracing/events/uprobes/enable
cat /sys/kernel/debug/tracing/trace
```

用户态任意位置都可以设置 uprobe 桩点

3.1.3.2 Perf probe 使用说明

除了 ftrace，我们还可以使用 Perf probe -x 动态跟踪点是一种可观察性功能，可实现以下功能：甚至不需要重新编译就可以插装源代码的任意行。 有了程序源代码的副本，跟踪点可以在运行时放置在任何地方，并且每个变量的值都可以转储时间执行通过跟踪点。 这是一项非常强大的技术，用于检测其他人编写的复杂系统或代码。

```plain text
perf probe -x /lib/libc.so.6 memcpy
perf record -e probe_libc:memcpy -aR ls
perf report
```

社区有大量 Uprobe 应用案例，和文档:

http://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html

https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt

https://www.kernel.org/doc/html/latest/trace/uprobetracer.html

3.2 Perf使用指南

3.2.1 Perf 简介

- Perf是一系列强大的性能分析工具集合。

- 在Linux 2.6.31版本引入，至今tool/perf目录拥有1万多个提交，是内核开发中最活跃的几个领域之一。
- Perf最初只负责处理系统性能事件，随着版本迭代和演进也引入了诸如probe，tracepotint，auxtrace，bpf等等各具特色的子工具。
- 通过perf可以使用一到两行命令就完成像程序热点采样，接口调用分析，阻塞分析这些以往需要插入大量分析代码才能完成的事情。
- 借助于内核日渐健全的tracepoint，perf拥有了一千多个linux内核预插桩点，可以对系统中调度，内存，文件系统，网络等各方面进行分析
- 围绕perf和系统性能事件，也有不少像perf-tool，火焰图，热点图，vtune等第三方功能扩展。

本说明主要针对**perf stat/record/report，硬件PMU，火焰图**几个部分重点进行介绍

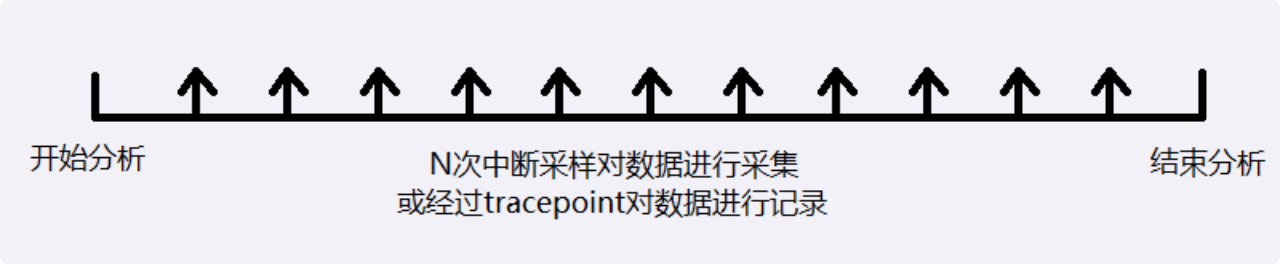
3.2.2 Perf 分析流程

Perf 进行性能分析的方式通常有两种：

- 使用perf stat等命令对特定的事件计数器进行计算，并在程序结束后打印数值
- 使用perf record等命令以若干的事件为触发间隔对系统进行采样，将数据保存至perf.data文件以供后续分析

```
# perf stat ls
messages

 Performance counter stats for 'ls':

             42.83 msec task-clock                #    0.735 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
                59      page-faults               #    0.001 M/sec
           2563283      cycles                    #    0.060 GHz
            714777      instructions              #    0.28  insn per cycle
            109487      conditionl-branch-instructions #    2.556 M/sec
             13290      conditional-branch-misspredict #   12.14% of all branches

       0.058298533 seconds time elapsed

       0.012837000 seconds user
       0.051349000 seconds sys
```

```
# perf record ls
messages       perf.data      perf.data.old
[ perf record: woken up 1 times to write data ]
[ perf record: Captured and wrote 0.011 MB perf.data (260 samples) ]
#
```



3.2.3 Perf 事件支持

通过Perf list可以查看当前支持的所有事件包含硬件事件，软件事件，硬件cache事件，PMU事件以及预设Tracepoint事件

```
# perf list

List of pre-defined events (to be used in -e):

  L1-icache-access                                   [Hardware event]
  L1-icache-misses                                   [Hardware event]
  conditional-branch-instructions                    [Hardware event]
  conditional-branch-misspredict                     [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  d-utlb-misses                                      [Hardware event]
  i-utlb-misses                                      [Hardware event]
  indirect-branch-instructions                       [Hardware event]
  indirect-branch-mispredict                         [Hardware event]
  instructions                                       [Hardware event]
  jtlb-misses                                        [Hardware event]
  lsu-cross-4k-stall-counter                         [Hardware event]
  lsu-other-stall-counter                            [Hardware event]
  lsu-speculation-fail                               [Hardware event]
  lsu-sq-data-discard-counter                        [Hardware event]
  lsu-sq-discard-counter                             [Hardware event]
  rf-instruction-counter                             [Hardware event]
  rf-launch-fail-counter                             [Hardware event]
  rf-reg-launch-fail-counter                         [Hardware event]
  store-instructions                                 [Hardware event]

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-store-misses                             [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
```

3.2.4 使用 Perf 的准备工作

T-HEAD 平台支持通过buildroot来快速完成整个Linux系统的搭建，所以要在T-HEAD平台上体验perf相关功能需要通过 https://github.com/c-sky/buildroot/releases 获取最新的buildroot使用源代码编译或使用Quick Start中的命令下载预编译的镜像直接执行，具体可参照 buildroot用户手册，perf功能在 T-HEAD配置中默认开启，启动后可以输入perf命令确认环境

```
Starting network: OK
processor       : 0
C-SKY CPU model : ck810f
product info[0] : 0x0504000c
product info[1] : 0x10000000
product info[2] : 0x20000000
product info[3] : 0x30000000
hint (CPU funcs): 0x00000000
ccr  (L1C & MMU): 0x00000001
ccr2 (L2C)      : 0x00000000

arch-version : e68b1635ac2711d5fc939bce66318aa118c9484a

Skip the ci test

Welcome to Buildroot
buildroot login: root
# perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
```

3.2.5 使用火焰图分析函数热点

以callchain_test为例如，这里将介绍如何使用perf命令和火焰图分析一个程序的函数热点

- 首先运行: perf record –g callchain_test 对 callchain_test程序的热点和调用栈进行采样
- 运行perf report可以直接观测函数热点，如果第一步中未使能 –g 选项则只显示热点不显示函数调用关系，可以看到右图中热点集中在 test_4 函数，函数调用入口为 test_1
- 当函数热点较为分散时可以通过火焰图更直观的看到函数调用关系

```
perf report
To display the perf.data header info, please use --header/--header-only options.


Total Lost Samples: 0

Samples: 154  of event 'cpu-clock'
Event count (approx.): 38500000

Overhead  Command          Shared Object      Symbol
........  ...............  ..................  ......................................

  55.19%  callchain_test   callchain_test     [.] test_4
   1.95%  callchain_test   [kernel.kallsyms]  [k] __softirqentry_text_start
   1.95%  callchain_test   [kernel.kallsyms]  [k] flush_tlb_one
   1.95%  callchain_test   [kernel.kallsyms]  [k] flush_tlb_range
   1.30%  callchain_test   [kernel.kallsyms]  [k] flush_tlb_mm
   1.30%  callchain_test   [kernel.kallsyms]  [k] free_hot_cold_page
   1.30%  callchain_test   [kernel.kallsyms]  [k] page_add_file_rmap
   1.30%  callchain_test   [kernel.kallsyms]  [k] path_openat
   1.30%  callchain_test   [kernel.kallsyms]  [k] sys_mmap_pgoff
   1.30%  callchain_test   ld-2.28.9000.so    [.] $t
   1.30%  callchain_test   libc-2.28.9000.so  [.] __cxa_atexit
   1.30%  perf             [kernel.kallsyms]  [k] perf_event_exec
   0.65%  callchain_test   [kernel.kallsyms]  [k] __d_lookup_rcu
   0.65%  callchain_test   [kernel.kallsyms]  [k] __fdget_raw
   0.65%  callchain_test   [kernel.kallsyms]  [k] __fput
   0.65%  callchain_test   [kernel.kallsyms]  [k] __kunmap_atomic
```

```
perf report
To display the perf.data header info, please use --header/--header-only options.


Total Lost Samples: 0

Samples: 164  of event 'cpu-clock'
Event count (approx.): 41000000

Children      Self  Command          Shared Object      Symbol
........  ........  ...............  ..................  ....................................

  62.80%     0.00%  callchain_test   libc-2.28.9000.so  [.] __libc_start_main
          |
          ---__libc_start_main
             |
             |--59.76%--main
             |          test_1
             |          test_2
             |          test_3
             |          test_4
             |          |
             |           --0.61%--ret_from_exception
             |                    schedule
             |                    __schedule
             |                    finish_task_switch
             |
             |--2.44%--0x3132c
             |          |
```

3.3 KASAN使用指南

3.3.1 简介

KernelAddressSANitizer（KASAN）是动态内存错误检测器。它提供了一种快速而全面的解决方案，以查找无用后使用和越界错误。

KASAN使用编译时工具来检查每个内存访问，因此您将需要GCC 4.9.2或更高版本。需要GCC 5.0或更高版本才能检测对堆栈或全局变量的越界访问。

3.3.2 使用方法

在内核中开启 KASAN:

Plain Text 复制代码

```
CONFIG_KASAN = y
```

并在CONFIG_KASAN_OUTLINE和CONFIG_KASAN_INLINE之间选择。 Outline和inline是编译器检测类型。 前者产生较小的二进制文件，而后者则快1.1–2倍。 内联检测需要GCC 5.0或更高版本。

KASAN可与SLUB和SLAB内存分配器一起使用。 为了更好地检测错误和更好地报告，请启用CONFIG_STACKTRACE。

要禁用特定文件或目录的检测，请在相应的内核Makefile中添加类似于以下内容的行：

- 对于单个文件 (譬如: main.o)

Plain Text 复制代码

```
KASAN_SANITIZE_main.o := n
```

- 对于一个目录中的所有文件

Plain Text 复制代码

```
KASAN_SANITIZE := n
```

3.3.3 出错报告

典型的越界访问报告如下所示：

Shell 复制代码

```shell
==================================================================
BUG: AddressSanitizer: out of bounds access in kmalloc_oob_right+0x65/0x75 [test_kasan] at addr ffff8800693bc5d3
Write of size 1 by task modprobe/1689
==================================================================
BUG kmalloc-128 (Not tainted): kasan error
------------------------------------------------------------------------------

Disabling lock debugging due to kernel taint
INFO: Allocated in kmalloc_oob_right+0x3d/0x75 [test_kasan] age=0 cpu=0 pid=1689
__slab_alloc+0x4b4/0x4f0
kmem_cache_alloc_trace+0x10b/0x190
kmalloc_oob_right+0x3d/0x75 [test_kasan]
init_module+0x9/0x47 [test_kasan]
do_one_initcall+0x99/0x200
load_module+0x2cb3/0x3b20
SyS_finit_module+0x76/0x80
system_call_fastpath+0x12/0x17
INFO: Slab 0xffffea0001a4ef00 objects=17 used=7 fp=0xffff8800693bd728 flags=0x100000000004080
INFO: Object 0xffff8800693bc558 @offset=1368 fp=0xffff8800693bc720

Bytes b4 ffff8800693bc548: 00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a  ........ZZZZZZZZ
Object ffff8800693bc558: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc568: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc578: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc588: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc598: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc5a8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc5b8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc5c8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
a5 kkkkkkkkkkkkkkk.

```

```
Redzone ffff8800693bc5d8: cc cc cc cc cc cc cc cc
     ........
Padding ffff8800693bc718: 5a 5a 5a 5a 5a 5a 5a 5a
     ZZZZZZZZ
CPU: 0 PID: 1689 Comm: modprobe Tainted: G    B          3.18.0-rc1-mm
1+ #98
ffff8800693bc000 0000000000000000 ffff8800693bc558 ffff88006923bb78
ffffffff81cc68ae 00000000000000f3 ffff88006d407600 ffff88006923bba8
ffffffff811fd848 ffff88006d407600 ffffea0001a4ef00 ffff8800693bc558
Call Trace:
[<ffffffff81cc68ae>] dump_stack+0x46/0x58
[<ffffffff811fd848>] print_trailer+0xf8/0x160
[<ffffffffa00026a7>] ? kmem_cache_oob+0xc3/0xc3 [test_kasan]
[<ffffffff811ff0f5>] object_err+0x35/0x40
[<ffffffffa0002065>] ? kmalloc_oob_right+0x65/0x75 [test_kasan]
[<ffffffff8120b9fa>] kasan_report_error+0x38a/0x3f0
[<ffffffff8120a79f>] ? kasan_poison_shadow+0x2f/0x40
[<ffffffff8120b344>] ? kasan_unpoison_shadow+0x14/0x40
[<ffffffff8120a79f>] ? kasan_poison_shadow+0x2f/0x40
[<ffffffffa00026a7>] ? kmem_cache_oob+0xc3/0xc3 [test_kasan]
[<ffffffff8120a995>] __asan_store1+0x75/0xb0
[<ffffffffa0002601>] ? kmem_cache_oob+0x1d/0xc3 [test_kasan]
[<ffffffffa0002065>] ? kmalloc_oob_right+0x65/0x75 [test_kasan]
[<ffffffffa0002065>] kmalloc_oob_right+0x65/0x75 [test_kasan]
[<ffffffffa00026b0>] init_module+0x9/0x47 [test_kasan]
[<ffffffff810002d9>] do_one_initcall+0x99/0x200
[<ffffffff811e4e5c>] ? __vunmap+0xec/0x160
[<ffffffff81114f63>] load_module+0x2cb3/0x3b20
[<ffffffff8110fd70>] ? m_show+0x240/0x240
[<ffffffff81115f06>] SyS_finit_module+0x76/0x80
[…] system_call_fastpath+0x12/0x17
Memory state around the buggy address:
ffff8800693bc300: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
ffff8800693bc380: fc fc 00 00 00 00 00 00 00 00 00 00 00 00 00 fc
ffff8800693bc400: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
ffff8800693bc480: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
ffff8800693bc500: fc fc fc fc fc fc fc fc fc fc fc 00 00 00 00 00
>ffff8800693bc580: 00 00 00 00 00 00 00 00 00 00 03 fc fc fc fc fc
^
ffff8800693bc600: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
ffff8800693bc680: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
ffff8800693bc700: fc fc fc fc fb fb fb fb fb fb fb fb fb fb fb fb
ffff8800693bc780: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
ffff8800693bc800: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
==================================================================
```

该报告的标题描述了发生了哪种错误以及导致这种错误的访问方式。后面是对访问的slub对象的描述（有关详细信息，请参见 Documentation/vm/slub.txt 中的“ SLUB调试输出”部分）以及对访问的内存页面的描述。

在最后一部分中，报告显示访问地址周围的内存状态。阅读此部分需要对KASAN的工作方式有一些了解。

存储器的每8个对齐字节的状态被编码在一个影子字节中。这8个字节可以访问，部分访问，释放或为redzone。对于每个影子字节，我们使用以下编码：0表示相应存储区的所有8个字节均可访问； N（1 <= N <= 7）表示前N个字节是可访问的，而其他（8–N）个字节则不可访问；任何负值表示整个8字节字不可访问。我们使用不同的负值来区分不同类型的不可访问的内存，例如Redzone或释放的内存（请参见mm / kasan / kasan.h）。

在报告上方的箭头指向影子字节03，这意味着被访问的地址可以部分访问。

本文参考：https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html

3.4 Kgdb使用指南

3.4.1 介绍

内核有两个不同的调试器前端（kdb和kgdb），它们连接到调试核心。如果在编译和运行时正确配置了内核，则可以使用两个调试器前端中的任何一个并在它们之间动态转换。

Kdb是简单化的shell风格的界面，可以在带有键盘或串行控制台的系统控制台上使用。您可以使用它来检查内存，寄存器，进程列表，dmesg，甚至设置断点以停在特定位置。尽管您可以设置断点并执行一些基本的内核运行控制，但Kdb并不是源代码级调试器。 Kdb主要旨在进行一些分析，以帮助开发或诊断内核问题。如果代码是使用CONFIG_KALLSYMS构建的，则可以按名称访问内核内置文件或内核模块中的某些符号。

Kgdb旨在用作Linux内核的源代码级调试器。它与gdb一起用于调试Linux内核。期望gdb可用于“侵入”内核以检查内存，变量并查看调用堆栈信息，类似于应用程序开发人员使用gdb调试应用程序的方式。可以在内核代码中放置断点并执行一些有限的执行步骤。

使用kgdb需要两台机器。其中一台机器是开发机器，另一台是目标机器。要调试的内核在目标计算机上运行。开发机器针对包含符号的vmlinux文件运行gdb实例（不是启动映像，如bzImage，zImage，uImage...）。在gdb中，开发人员指定连接参数并连接到kgdb。开发人员与gdb建立的连接类型取决于在测试计算机内核中编译为内置模块或可加载内核模块的kgdb I / O模块的可用性。

3.4.2 编译内核

在 menuconfig 中开启 CONFIG_KGDB

Shell 复制代码

```shell
Symbol: KGDB [=y]
Type  : bool
```

3.5 Lockdep使用指南

3.5.1 简介

验证程序操作的基本对象是锁的类型。

一类锁是一组在锁规则上逻辑上相同的锁，即使锁可能具有多个（可能是数万个）实例化。 例如，inode结构中的锁是一个类，而每个inode具有该锁类的自己的实例化。

验证器跟踪锁类的“状态”，并跟踪不同锁类之间的依赖关系。 验证器保持状态和依赖关系正确的滚动证明。

锁实例化不同，锁类本身永远不会消失：在启动后首次使用锁类时，它将被注册，并且该锁类的所有后续使用都将附加到该锁类上

3.5.2 使用介绍

在 Kernel hacking -> Lock Debuging 中开启 LOCK_DEP 测试，(注意不要开启 LOCK_STAT)

```
.config - Linux/riscv 5.8.0-rc7 Kernel Configuration
→ Kernel hacking → Lock Debugging (spinlocks, mutexes, etc...)
                    Lock Debugging (spinlocks, mutexes, etc...)
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are
  hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                        [*] Lock debugging: prove locking correctness
                        [*]   Enable raw_spinlock - spinlock nesting checks
                        [ ] Lock usage statistics
                        -*- RT Mutex debugging, deadlock detection
                        -*- Spinlock and rw-lock debugging: basic checks
                        -*- Mutex debugging: basic checks
                        -*- Wait/wound mutex debugging: Slowpath testing
                        -*- RW Semaphore debugging: basic checks
                        -*- Lock debugging: detect incorrect freeing of live locks
                        [*] Lock dependency engine debugging
                        [*] Sleep inside atomic section checking
                        [*] Locking API boot-time self-tests
                        <*> torture tests for locking
                        <*> Wait/wound mutex selftests
```

运行后，会启动自动测试

Plain Text 复制代码

```
[    0.000424] Lock dependency validator: Copyright (c) 2006 Red Har
[    0.000443] ... MAX_LOCKDEP_SUBCLASSES:  8
[    0.000460] ... MAX_LOCK_DEPTH:          48
[    0.000476] ... MAX_LOCKDEP_KEYS:        8192
[    0.000492] ... CLASSHASH_SIZE:          4096
[    0.000508] ... MAX_LOCKDEP_ENTRIES:     32768
[    0.000524] ... MAX_LOCKDEP_CHAINS:      65536
[    0.000540] ... CHAINHASH_SIZE:          32768
[    0.000556]  memory used by lock dependency info: 6301 kB
[    0.000572]  memory used for stack traces: 4224 kB
[    0.000588]  per task-struct memory footprint: 1920 bytes
[    0.000604] ------------------------
[    0.000620] | Locking API testsuite:
[    0.000636] ----------------------------------------------------------
[    0.000651]                                 | spin |wlock |rloc|
[    0.000667]   ----------------------------------------------------
[    0.000698]                     A-A deadlock:  ok  |  ok  |  ok|
[    0.009065]                 A-B-B-A deadlock:  ok  |  ok  |  ok|
[    0.018270]             A-B-B-C-C-A deadlock:  ok  |  ok  |  ok|
[    0.028441]             A-B-C-A-B-C deadlock:  ok  |  ok  |  ok|
[    0.038546]         A-B-B-C-C-D-D-A deadlock:  ok  |  ok  |  ok|
[    0.049592]         A-B-C-D-B-D-D-A deadlock:  ok  |  ok  |  ok|
[    0.060594]         A-B-C-D-B-C-D-A deadlock:  ok  |  ok  |  ok|
[    0.071571]                    double unlock:  ok  |  ok  |  ok|
[    0.079825]                  initialize held:  ok  |  ok  |  ok|
[    0.087588]   ----------------------------------------------------
[    0.087606]              recursive read-lock:             |  ok|
[    0.090094]           recursive read-lock #2:             |  ok|
[    0.092472]            mixed read-write-lock:             |  ok|
[    0.094807]            mixed write-read-lock:             |  ok|
[    0.097203]   mixed read-lock/lock-write ABBA:             |  ok|
[    0.099780]    mixed read-lock/lock-read ABBA:             |FAIL|
[    0.102444]  mixed write-lock/lock-write ABBA:             |  ok|
[    0.105050]   ----------------------------------------------------
[    0.105079]      hard-irqs-on + irq-safe-A/12:  ok  |  ok  |  ok|
[    0.108638]      soft-irqs-on + irq-safe-A/12:  ok  |  ok  |  ok|
[    0.112183]      hard-irqs-on + irq-safe-A/21:  ok  |  ok  |  ok|
[    0.115749]      soft-irqs-on + irq-safe-A/21:  ok  |  ok  |  ok|
[    0.119282]        sirq-safe-A => hirqs-on/12:  ok  |  ok  |  ok|
[    0.122826]        sirq-safe-A => hirqs-on/21:  ok  |  ok  |  ok|
[    0.126384]          hard-safe-A + irqs-on/12:  ok  |  ok  |  ok|
[    0.129957]          soft-safe-A + irqs-on/12:  ok  |  ok  |  ok|
[    0.133488]          hard-safe-A + irqs-on/21:  ok  |  ok  |  ok|
[    0.137056]          soft-safe-A + irqs-on/21:  ok  |  ok  |  ok|
[    0.140635]     hard-safe-A + unsafe-B #1/123:  ok  |  ok  |  ok|
```

```
[    0.144589]     soft-safe-A + unsafe-B #1/123:  ok  |  ok  |  ok|
[    0.148522]     hard-safe-A + unsafe-B #1/132:  ok  |  ok  |  ok|
[    0.152483]     soft-safe-A + unsafe-B #1/132:  ok  |  ok  |  ok|
[    0.156465]     hard-safe-A + unsafe-B #1/213:  ok  |  ok  |  ok|
[    0.160438]     soft-safe-A + unsafe-B #1/213:  ok  |  ok  |  ok|
[    0.164415]     hard-safe-A + unsafe-B #1/231:  ok  |  ok  |  ok|
[    0.168325]     soft-safe-A + unsafe-B #1/231:  ok  |  ok  |  ok|
```

第4章 展望未来

更新后的玄铁处理器Linux有如下5大亮点

4.1 极简开发，轻松上手

对于嵌入式开发者来说，能有易于上手的开发环境，简单操作的系统是能够节省很多时间的。**玄铁处理器 linux 开发平台是基于 buildroot.org 构建的，帮助开发者快速构建玄铁 CPU Linux 学习和实验环境，快速集成开源组件。**

Buildroot 是一款简单，高效，易于使用的开源嵌入式linux 交叉编译开发平台（https://buildroot.org），每一款 CPU 提供了最小系统参考配置，通过

https://github.com/c-sky/buildroot/releases 按季度进行发布，同时结合 gitlab 开源 CI/CD (持续集成/持续发布) 系统，保障发布给开源社区代码正确性。

玄铁 linux 开发平台融入代码仓库的动态 readme.txt 设计，将上手指引精确到每一个 config，逐行精简命令，一目了然（每一个步骤不超过 10 条命令，支持 'ctrl c' + 'ctrl v' 秒贴执行，极速上手）。

演示环境--快速上手玄铁linux 开发平台

利用云上 CI/CD 系统，预先编译完成各个配置，并把结果存在云上，供开发者直接下载快速体验 linux 环境。其中 hw 目录下，存放着玄铁系列 CPU 最小系统硬件平台，包含 fpga, asic。

配合玄铁 CPU JTAG调试器，提供清晰简洁的运行脚本，不浪费第一次上手玄铁 CPU 用户的一分一秒。

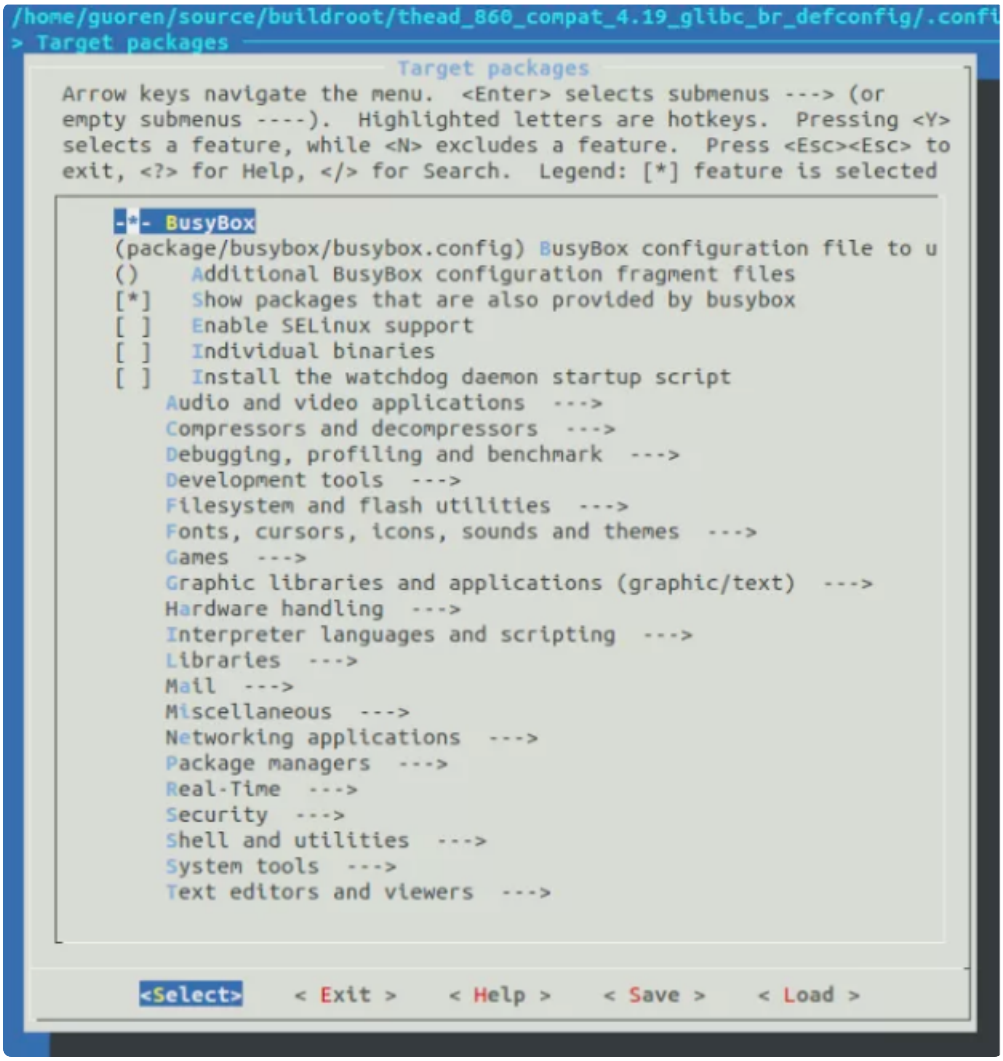
(具体请参考 readme.txt -> 'Quick Start for hw run' 章节)

4.2 软件丰富，定制灵活

通过开发平台，可获得:

- linux 最小系统样例
- Bootloader 和 Opensbi 样例
- 基于最小系统硬件，调试环境 (JTAG + DRAM + 网络)
- 基于 QEMU 的配套验证环境
- 丰富的诊断工具 (gdb, perf PMU state/record, ftrace, kprobe/uprobe, tracecmd + kernelshard, bpf 等)
-

一般来说，对于一个新的 CPU 体系架构，构建一套最小系统，要从基础编译器，内核，最小文件系统开始准备，费时耗力。**玄铁 linux 开发平台通过一个配置，将工具链，内核，根文件系统的软件包集成在一起，一条简洁的命令就可以让开发者从源码级构建整个 linux 系统，非常方便！**



配置图例

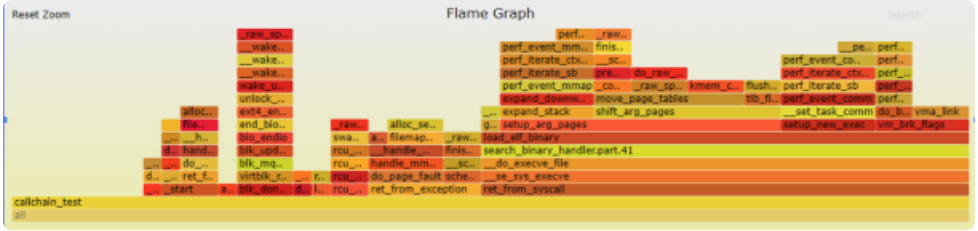
4.3 易于诊断，高效定位

玄铁处理器linux 诊断性工具众多，包含如下工具:

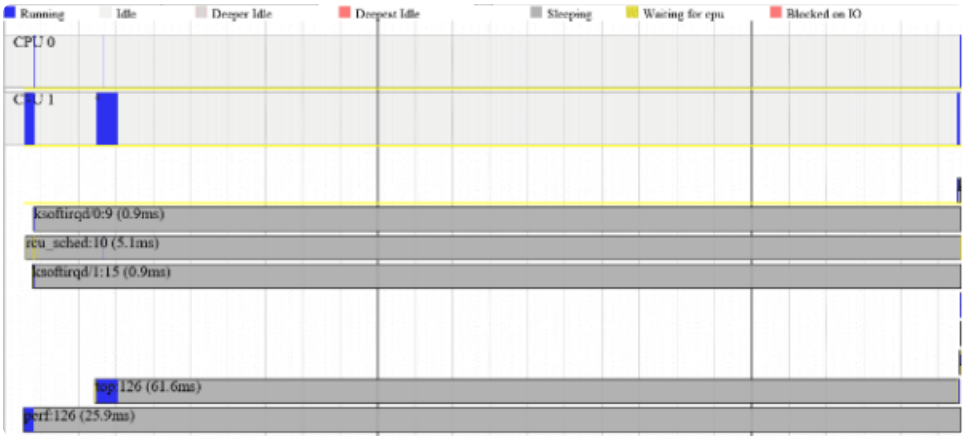
- 以 Perf, ftrace, K/Uprobe, stap, BPF tools 组合形成的 linux 性能分析，动态追踪，系统观测工具
- 以 GDB/KGDB/KDUMP 为代表的 交互式，在/离线调试工具
- 以 top/free/slabtop/iostat/iotop/netstat/sar 为代表的 linux 系统监控工具

在 v1.0.9 版本中，实现了玄铁 C800 CPU kprobe/uprobe 指令流插桩功能，即使在没有源码的情况下，也可以通过 perf probe, tracecmd 等前端工具，从容调试，极大提升开发效率。

- 支持 Perf stat/record + callchain (配合图形前端工具可生成火焰图，timechart)



火焰图

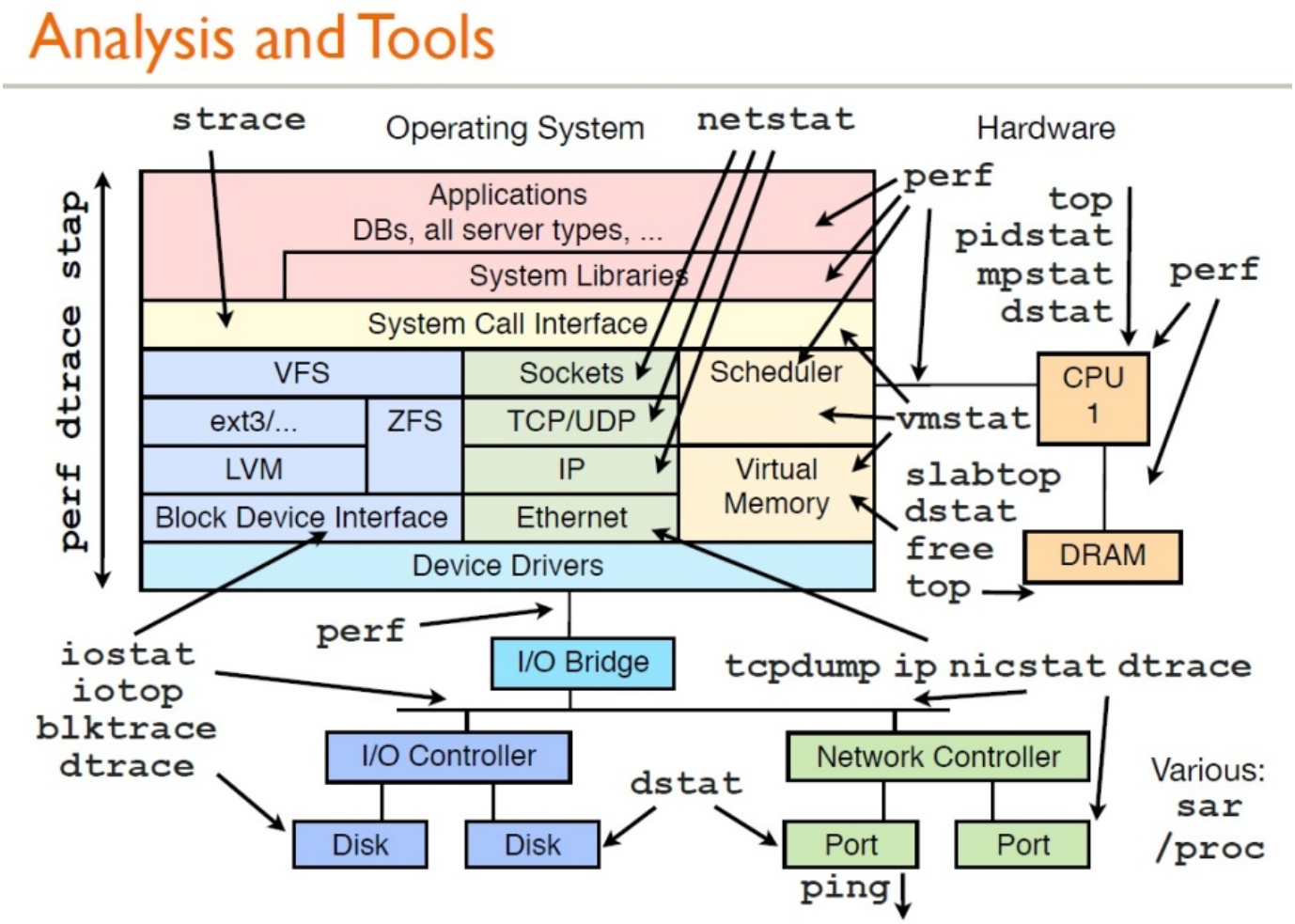


timechart图

- 支持 Perf PMU 硬事件统计 & 采样，覆盖指令数/周期数/CACHE行为/TLB行为/LSU行为 等多个 CPU 内部模块，方便 CPU 评估，分析性能瓶颈，快速定位问题。
- 支持 ftrace 内核追踪框架
- 支持 kprobe/uprobe linux 动态追踪功能

- 支持 gdb，gdbserver，kdump，coredump 交互调试工具
- 支持 lockdep 内核死锁检查
- 支持 gcov 内核代码覆盖率统计
- 支持 strace 系统调用追踪
- 支持 mtrace (glibc) 内存泄漏分析

未来会持续完善 Linux 诊断性工具集，提供丰富的 Linux 诊断手段。

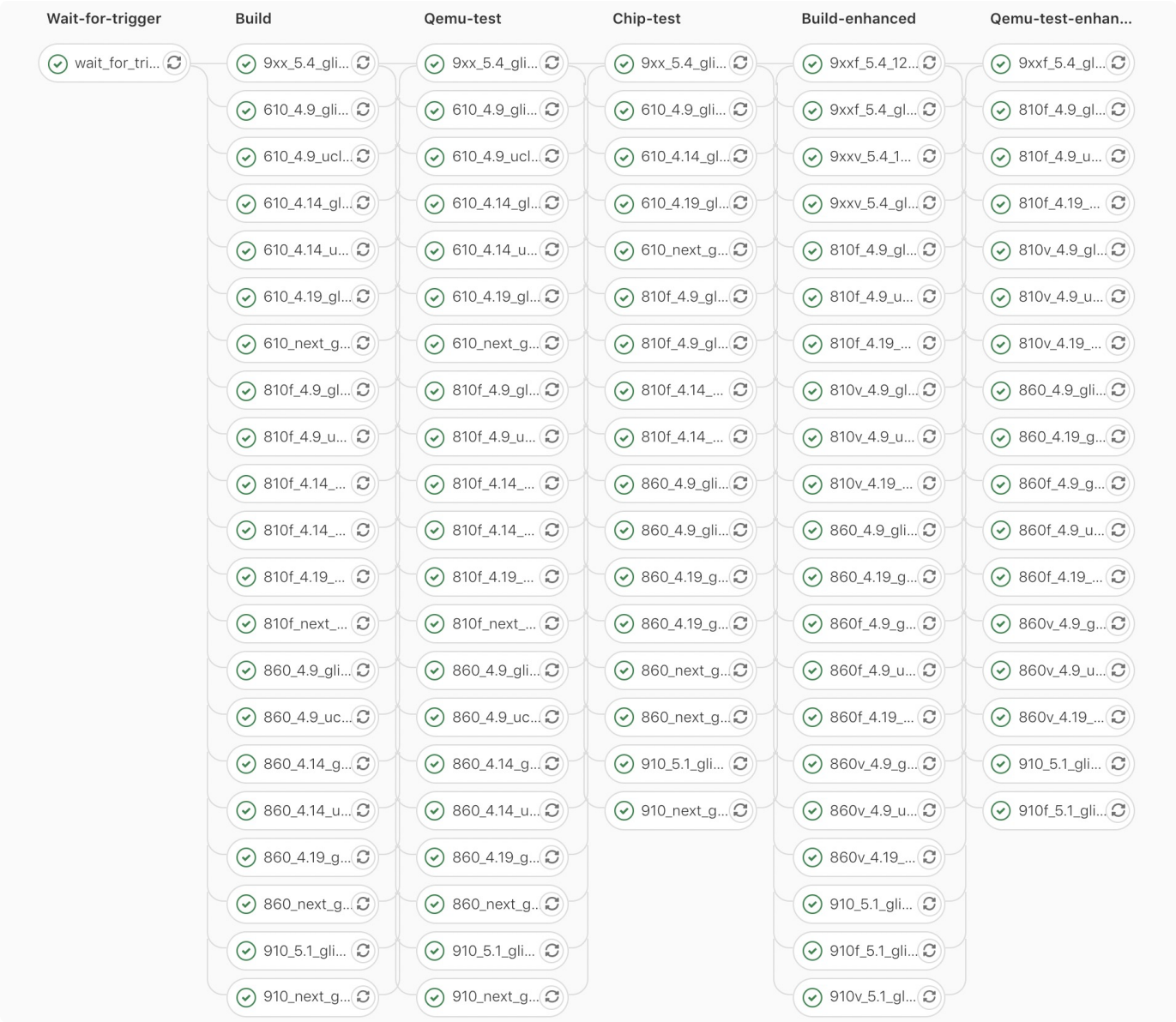


图片来源brendangregg.com

4.4 持续集成，质量保证

玄铁系列 CPU 已被众多开源生态软件支持，为了让用户在使用这些开源软件时，得到更好的保障，平头哥建立持续跟踪系统，确保开源软件在迭代过程中的质量。

在 gitlab 上构建了一个开源开放的 CI/CD 系统。它经历了1年多的运行与完善，累计执行数千次 pipeline 版本迭代，有效保障了开源社区 CPU 后端代码的质量。(https://gitlab.com/c-sky/buildroot/pipelines/charts)



4.5 源于开源，回馈社区

一款优秀的 CPU，离不开社区软件生态的支持，离不开积极参与 kernel.org 社区的开发，开发者也拥有自己的 mailing list:

linux-csky@vger.kernel.org (https://lore.kernel.org/linux-csky/),

并随着 linux 开发节奏，定期向 Linus Torvalds 先生发出来自中国 CPU 架构 的 pull-request