RVOS25 glibc and RVV report

搭建 RISC-V glibc 的调试环境并尝试使用 RVV 优化 memcpy

> TL;DR: 目前已完成的工作总结
>
> 1. 基于 rvv-env 修改了可用于 RISC-V glibc 的调试环境
> 2. 阅读了 RVV 1.0 SPEC
> 3. 对 glibc 中的 `memcpy` 进行了向量化
> 4. 对向量化之后的 `memcpy` 进行了测试

> 本文的原始文件与代码可从 GitHub 仓库 获取.

这篇文章尝试使用 `rvv-env` 作为 RISC-V 下的交叉编译与调试环境，并在其基础上进行自定义，增加必须的工具与符号等，以用来调试 RISC-V glibc。

`rvv-env` 环境搭建

`rvv-env` 提供了容器化的 RISC-V 编译与调试环境，使用 `qemu-user` 在一个容器中（称作 `target`）进行模拟，并在另一个容器（称作 `host`）中通过 GDB 进行远程调试。我对该项目进行了一些修改，在正确放置所有文件后，拉取镜像后应当可以直接调试 glibc.

常规的使用方法及存在的问题

通常来说，不考虑 glibc 及其调试符号的情况下，我们只需要按照 README 中的顺序，安装 `qemu-user-binfmt` 并拉取镜像即可，也就是

1. `sudo apt install qemu-user-binfmt` ，注意如果你在使用 Ubuntu 24.04，你可能需要使用 `qemu-user` 包替换前者。
2. `git clone https://gitlab.com/riseproject/rvv-env.git`
3. `source env.sh`
4. `./oci-pull.sh` 拉取预构建的镜像

之后，通过 `target-run gcc -o test-out -g test.c` 来编译可执行文件，然后首先 `target-gdb test-out` 启动 QEMU 并附加 gdb，之后再 `host-gdb` 连接到远程的 gdb 进行调试。

一般来说这种方法已经足够，但为了调试 glibc，我们还需要两个东西：**对应当前版本的** glibc 的调试符号，以及**对应当前版本的** glibc 源码。这两种方式都可以通过获取对应的包后提取相应的文件，并在容器启动时挂载到对应的位置上来解决。

> 其实最开始我是通过直接在构建镜像时安装这些包解决的，但这完全没有必要 QAQ 构建镜像时会联网拉取预构建的 RISC-V 工具链，很慢，所以我修改了 Dockerfile，现在它会使用本地的文件。当然，这需要你先下载一次并放到 `rvv-env/container/`

获取相关的文件

> tldr: `work/` 下所有需要的文件都已经被打包至仓库的 release 中，下载 `work.tar.xz` 并解压即可，注意路径**不要**出现 `rvv-env/work/work/...`，即多层 `work/` 嵌套。

为了简便起见，我的 host 与 target 的基础镜像都使用 Ubuntu 24.04 Noble，对应的 `libc6` 的包的版本如下，其中 `libc6-dbg` 是调试符号：

```
(py) horizon@horizon-VMware20-1:~/project/rvv-env$
IMAGE_VARIANT_TARGET=target-ubuntu target-run bash
horizon@target-ubuntu:/rvv-env$ apt list | grep libc6

WARNING: apt does not have a stable CLI interface. Use with caution in
scripts.

libc6-dbg-riscv64-cross/now 2.39-0ubuntu8cross1 all [installed,local]
libc6-dbg/now 2.39-0ubuntu8.4 riscv64 [installed,local]
libc6-dev/now 2.39-0ubuntu8.4 riscv64 [installed,local]
libc6-riscv64-cross/now 2.39-0ubuntu8cross1 all [installed,local]
libc6/now 2.39-0ubuntu8.4 riscv64 [installed,local]
```

所需的包可以从 https://packages.ubuntu.com 搜索，点击对应架构的文件列表可以看到包安装后的各个文件的位置。下载 `libc6-dbg` 后，将其中的 `usr/` 提取出，放置在 `rvv-env/work/` 下。`下载 glibc 源码` 后，将 `glibc-2.39/` 同样放置在 `rvv-env/work/` 下。`下载 pwndbg portable` 后，将 `pwndbg/` 同样放置在 `rvv-env/work/` 下。此时你的 `rvv-env/work/` 看起来应该是这样的：

```
(py) horizon@horizon-VMware20-1:~/project/rvv-env$ tree -L 2 work
work
├── glibc-2.39
│   ├── abi-tags
│   ├── aclocal.m4
│   ├── ADVISORIES
│   ├── argp
```

```
│   ├── assert
│   ├── benchtests
│   ├── bits
│   ├── catgets
│   ├── ChangeLog.old
│   ├── config.h.in
│   ├── config.make.in
│   ├── configure
│   ├── configure.ac
│   ├── conform
│   ├── CONTRIBUTED-BY
│   ├── COPYING
│   ├── ... (omitted)
│   ├── wcsmbs
│   └── wctype
├── pwndbg
│   ├── bin
│   ├── exe
│   ├── lib
│   └── share
└── usr
    ├── lib
    └── share

71 directories, 31 files
```

拉取镜像

安装 Docker（建议使用 rootless mode）后，在 `rvv-env` 下 `source env.sh`，再拉取镜像，注意这里拉取的是 `target-ubuntu`：

```
(py) horizon@horizon-VMware20-1:~/project/rvv-env$
IMAGE_VARIANT_TARGET=target-ubuntu ./oci-pull.sh
```

对 `rvv-env` 进行的修改

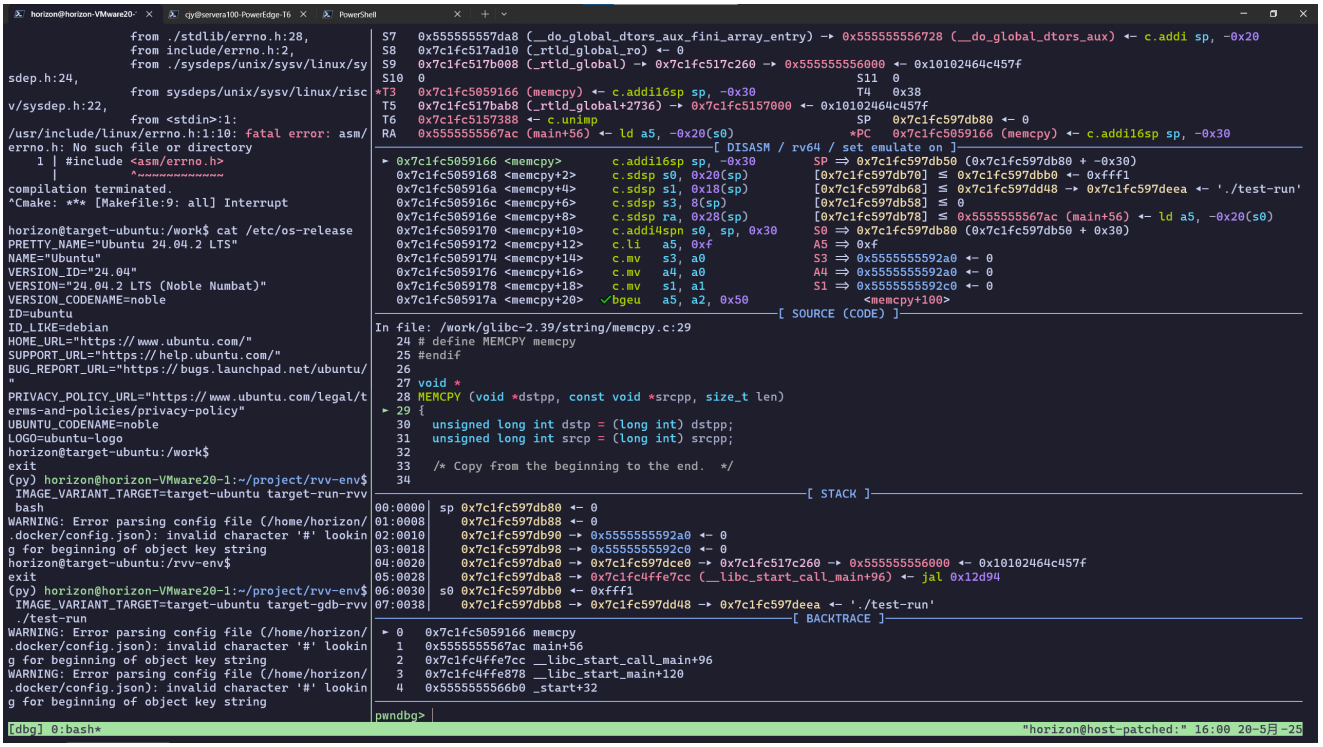
1. 修改了 `env.sh`，现在 `source env.sh` 是幂等的，同时会将 `${WORK_DIR}` 指向 `rvv-env/work/`
2. 修改了 `oci-run.sh`，额外增加了一个 `host-pwndbg` 的命令，用于使用 pwndbg 进行调试
3. 修改了 `oci-run.sh`，默认设置好符号和源码的搜索路径（目前仅对 `host-gdb` 生效；由于 `pwndbg` 本质上是 gdb 的插件，在加载之后就会覆盖配置，难以通过命令行参数指定）

4. 修改了 `container/variants/target-*.env`，现在在构建镜像时会额外增加 glibc 相关的包，不过因为不再需要手动构建镜像所以用不到了

使用 `rvv-env`

基本的使用方法按照 常规的使用方法及存在的问题 即可，修改后的 `rvv-env` 默认会将 `rvv-env/work/` 挂载到容器的 `/work/` 下，同时额外增加了一个 `host-pwndbg` 的命令，用于使用 pwndbg 进行调试，并默认启用 `compact-reg` 来缩小寄存器显示的空间。

右侧是通过 `host-pwndbg` 运行的 `pwndbg`，同时设置了源码与符号的路径，正在观察调试 `memcpy`.



向量化 `memcpy`

> 在完成了之后，才发现于佳耕老师在 25 年 1 月份 提交了 `memcpy` 的向量化补丁，不过 `glibc 2.41.0` 并未包含了向量化的 `memcpy`，而是增加了支持快速未对齐访问的版本...

对常规版本的 `memcpy` 的分析

常规版本的 `memcpy`（`glibc-2.39/string/memcpy.c`）其实原理比较简单，代码如下：

```
void *
MEMCPY (void *dstpp, const void *srcpp, size_t len)
```

```
{
  unsigned long int dstp = (long int) dstpp;
  unsigned long int srcp = (long int) srcpp;

  /* Copy from the beginning to the end.  */

  /* If there not too few bytes to copy, use word copy.  */
  if (len >= OP_T_THRES)
    {
      /* Copy just a few bytes to make DSTP aligned.  */
      len -= (-dstp) % OPSIZ;
      BYTE_COPY_FWD (dstp, srcp, (-dstp) % OPSIZ);

      /* Copy whole pages from SRCP to DSTP by virtual address
manipulation,
     as much as possible.  */

      PAGE_COPY_FWD_MAYBE (dstp, srcp, len, len);

      /* Copy from SRCP to DSTP taking advantage of the known
alignment of
     DSTP.  Number of bytes remaining is put in the third argument,
     i.e. in LEN.  This number may vary from machine to machine.  */

      WORD_COPY_FWD (dstp, srcp, len, len);

      /* Fall out and copy the tail.  */
    }

  /* There are just a few bytes to copy.  Use byte memory operations.
*/
  BYTE_COPY_FWD (dstp, srcp, len);

  return dstpp;
}
```

具体来说，当复制的长度足够（>= 16 bytes）时，首先复制前 `(-dstp) % OPSIZ` 个字节，使得 src 和 dst 与 16B 对齐，之后尝试按照整页复制（不过 RISC-V 上由于 `PAGE_COPY_THRESHOLD` 被定义为 `0`，所以这里并未生效）. 最后，逐字节拷贝直到完成。

RVV 下的实现

基于对原始 `memcpy` 的分析，我们可以使用向量加速复制的过程，即

1. 计算这次向量操作的长度，即 `vsetl` 指令

2. 按块读取内存，保存到向量寄存器，即 `vle` 指令
3. 将向量寄存器的值保存到指定地址，即 `vse` 指令

不过在这之前和之后，我们需要手动处理对齐和尾部剩余的问题，就像原始版本的 `memcpy` 做的那样。核心部分的代码如下，具体的细节参见注释。

> 完整的代码参见 GitHub 上的仓库

```
void* memcpy_rvv_internal(void* dst, const void* src, size_t len) {
    void* original_dst = dst;
    uint8_t* dst_u8 = (uint8_t*)dst;
    const uint8_t* src_u8 = (const uint8_t*)src;
    size_t vlenb;

    // 获取硬件支持的单个向量寄存器的长度，单位字节
    asm volatile("csrr %0, vlenb" : "=r"(vlenb));

    // 如果长度小于16字节，直接使用字节拷贝，类似原始版本的 memcpy
    if (len < 16) {
        for (size_t i = 0; i < len; ++i)
            dst_u8[i] = src_u8[i];
        return original_dst;
    }

    // 确保两个指针都对齐到 8 字节
    size_t head_len = 0;
    if (((uintptr_t)dst_u8 % sizeof(unsigned long)) != 0) {
        head_len = sizeof(unsigned long) - ((uintptr_t)dst_u8 %
sizeof(unsigned long));
        if (head_len > len)
            head_len = len;

        // 处理头部未对齐的字节
        for (size_t i = 0; i < head_len; ++i)
            dst_u8[i] = src_u8[i];
        dst_u8 += head_len;
        src_u8 += head_len;
        len -= head_len;
    }

    // 向量化，启动
    if (len >= sizeof(unsigned long) && (((uintptr_t)src_u8 %
sizeof(unsigned long)) == 0)) {
        unsigned long* dst_ul = (unsigned long*)dst_u8;
        const unsigned long* src_ul = (const unsigned long*)src_u8;
        size_t num_ul = len / sizeof(unsigned long);
```

```
        size_t current_vl_ul;
        if (num_ul > 0) {
            // 64-bit
            if (sizeof(unsigned long) == 8) {
                asm volatile(
                    "1:\n" //循环开始
                    "vsetvli %[vl], %[len_in_elements], e64, m8, ta,
ma\n" //设置vl
                    "beqz %[vl], 2f\n" //如果 vl == 0, 跳到 2, 也就是结束
                    "vle64.v v0, (%[src_ptr])\n" // 从 src 读入数据到 v0
                    "vse64.v v0, (%[dst_ptr])\n" // 从 v0 写入数据到 dst
                    // 计算处理的字节数
                    "slli %[processed_bytes], %[vl], 3\n"
                    "add %[src_ptr], %[src_ptr], %[processed_bytes]\n"
                    "add %[dst_ptr], %[dst_ptr], %[processed_bytes]\n"
                    "sub %[len_in_elements], %[len_in_elements], %
[vl]\n"
                    "j 1b\n"
                    "2:\n"
                    : [vl] "=&r"(current_vl_ul), [len_in_elements]
"+&r"(num_ul),
                    [src_ptr] "+&r"(src_ul), [dst_ptr] "+&r"(dst_ul)
                    : [processed_bytes] "r"(0)
                    : "t0", "t1", "memory", "v0", "v1", "v2", "v3",
"v4", "v5", "v6", "v7");
            } else {
                asm volatile(
                    "1:\n"
                    "vsetvli %[vl], %[len_in_elements], e32, m8, ta,
ma\n"
                    "beqz %[vl], 2f\n"
                    "vle32.v v0, (%[src_ptr])\n"
                    "vse32.v v0, (%[dst_ptr])\n"
                    "slli %[processed_bytes], %[vl], 2\n"
                    "add %[src_ptr], %[src_ptr], %[processed_bytes]\n"
                    "add %[dst_ptr], %[dst_ptr], %[processed_bytes]\n"
                    "sub %[len_in_elements], %[len_in_elements], %
[vl]\n"
                    "j 1b\n"
                    "2:\n"
                    : [vl] "=&r"(current_vl_ul), [len_in_elements]
"+&r"(num_ul),
                    [src_ptr] "+&r"(src_ul), [dst_ptr] "+&r"(dst_ul)
                    : [processed_bytes] "r"(0)
                    : "t0", "t1", "memory", "v0", "v1", "v2", "v3",
"v4", "v5", "v6", "v7");
            }
```

```
            size_t bytes_processed_ul = (len / sizeof(unsigned long) -
num_ul) * sizeof(unsigned long);
            dst_u8 = (uint8_t*)dst_ul;
            src_u8 = (const uint8_t*)src_ul;
            len -= bytes_processed_ul;
        }
    }
    // 如果还有剩余的字节，使用字节向量化拷贝
    if (len > 0) {
        size_t current_vl_u8;
        uint8_t* dst_u8_asm = dst_u8;
        const uint8_t* src_u8_asm = src_u8;
        size_t len_asm = len;
        asm volatile(
            "1:\n"
            "vsetvli %[vl], %[rem_len], e8, m8, ta, ma\n"
            "beqz %[vl], 2f\n"
            "vle8.v v0, (%[src_ptr])\n"
            "vse8.v v0, (%[dst_ptr])\n"
            "add %[src_ptr], %[src_ptr], %[vl]\n"
            "add %[dst_ptr], %[dst_ptr], %[vl]\n"
            "sub %[rem_len], %[rem_len], %[vl]\n"
            "j 1b\n"
            "2:\n"
            : [vl] "=&r"(current_vl_u8), [rem_len] "+&r"(len_asm),
            [src_ptr] "+&r"(src_u8_asm), [dst_ptr] "+&r"(dst_u8_asm)
            :
            : "t0", "memory", "v0", "v1", "v2", "v3", "v4", "v5", "v6",
"v7");
    }
    return original_dst;
}
```

整体的思路和原始版本的 `memcpy` 基本一致。

对向量化后的正确性与性能分析

对两种不同的 `memcpy`，在各种不同的复制大小下进行测试，并使用 `memcmp` 保证正确性，结果如下：

> 编译参数：`-march=rv64gcv -mabi=lp64d -O2`
> 在 `target-run-rvv`（即 qemu）中运行

> 完整代码参见我的 GitHub 仓库

Size (bytes)	memcpy_std (cycles)	memcpy_rvv (cycles)	Speedup (std/rvv)	Correct_std	Correct_rvv
1	114	24	4.75	OK	OK
16	240	86	2.79	OK	OK
31	255	148	1.72	OK	OK
32	267	170	1.57	OK	OK
63	283	317	0.89	OK	OK
64	241	315	0.77	OK	OK
127	307	557	0.55	OK	OK
128	312	794	0.39	OK	OK
255	357	1520	0.23	OK	OK
256	356	1435	0.25	OK	OK
511	429	2751	0.16	OK	OK
512	417	2562	0.16	OK	OK
1023	626	4954	0.13	OK	OK
1024	563	4603	0.12	OK	OK
2048	893	9173	0.10	OK	OK
4095	1565	17987	0.09	OK	OK
4096	1561	17649	0.09	OK	OK
8192	2838	35082	0.08	OK	OK
16384	6515	69516	0.09	OK	OK
32768	14555	139512	0.10	OK	OK
65536	28801	278598	0.10	OK	OK
131072	58267	557906	0.10	OK	OK
262144	118023	1120263	0.11	OK	OK
524288	236935	2239374	0.11	OK	OK
1048576	476893	4481144	0.11	OK	OK
2097152	981736	8958307	0.11	OK	OK
4194304	2006807	18082111	0.11	OK	OK

结果表明，在 qemu 下，向量化对于 `memcpy` 的性能提升并不显著，甚至具有负面作用。使用每次只搬运一字节的向量化版本（即 `e8`）进行对比，结果如下：

Size (bytes)	memcpy_std (cycles)	memcpy_rvv (cycles)	Speedup (std/rvv)	Correct_std	Correct_rvv
1	122	253	0.48	OK	OK
16	300	558	0.54	OK	OK
31	259	864	0.30	OK	OK
32	292	915	0.32	OK	OK
63	334	1599	0.21	OK	OK
64	244	1564	0.16	OK	OK
127	329	2891	0.11	OK	OK
128	283	2887	0.10	OK	OK
255	346	5390	0.06	OK	OK
256	323	5657	0.06	OK	OK
511	433	10962	0.04	OK	OK
512	420	11072	0.04	OK	OK
1023	605	21798	0.03	OK	OK
1024	568	21833	0.03	OK	OK
2048	873	43402	0.02	OK	OK
4095	1555	87718	0.02	OK	OK
4096	1546	86717	0.02	OK	OK
8192	2820	172946	0.02	OK	OK
16384	6406	345852	0.02	OK	OK
32768	14529	691405	0.02	OK	OK
65536	29114	1382283	0.02	OK	OK
131072	58199	2771912	0.02	OK	OK
262144	117404	5551565	0.02	OK	OK
524288	234926	11101120	0.02	OK	OK
1048576	473707	22154676	0.02	OK	OK

可以发现更大的元素宽度对于向量化后的性能是有提升的，这说明向量化后的性能降低可能与 qemu 实现中 `vsetvl`，`vle` 和 `vse` 指令的实现性能有关 -- 模拟后的这些指令的开销可能过大。

总结与结论

这篇文章在 `rvv-env` 的基础上，增加了对 glibc 调试的支持，包括源码与调试符号等，同时增加了 pwndbg 作为调试工具，尽可能做到开箱即用。

另一方面，文章对 `memcpy` 进行了一定程度的向量化，使用 `vse` 与 `vle` 组合实现内存的向量化复制，同时测试了向量化后的性能，尽管由于条件所限在 qemu 下向量化后的性能没有提升，但保证了结果的正确性。

下一步的计划

下一部的计划继续围绕 glibc 及其 `IFUNC` 机制展开。`IFUNC` 机制提供了根据运行时平台的硬件特性选择合适版本的库函数执行的能力，在当前的场景下，可以实现同一份 glibc 二进制在硬件支持 RVV 时使用向量化版本，不支持时回退到标量版本的能力。

1. 继续阅读 RVV Spec 与 IFUNC 相关的文档，了解其用法
2. 阅读 glibc 源码，了解代码结构
3. 为 `memcpy` 添加 IFUNC 支持
4. 继续尝试优化其他 glibc 库函数