RISC-V Summit China

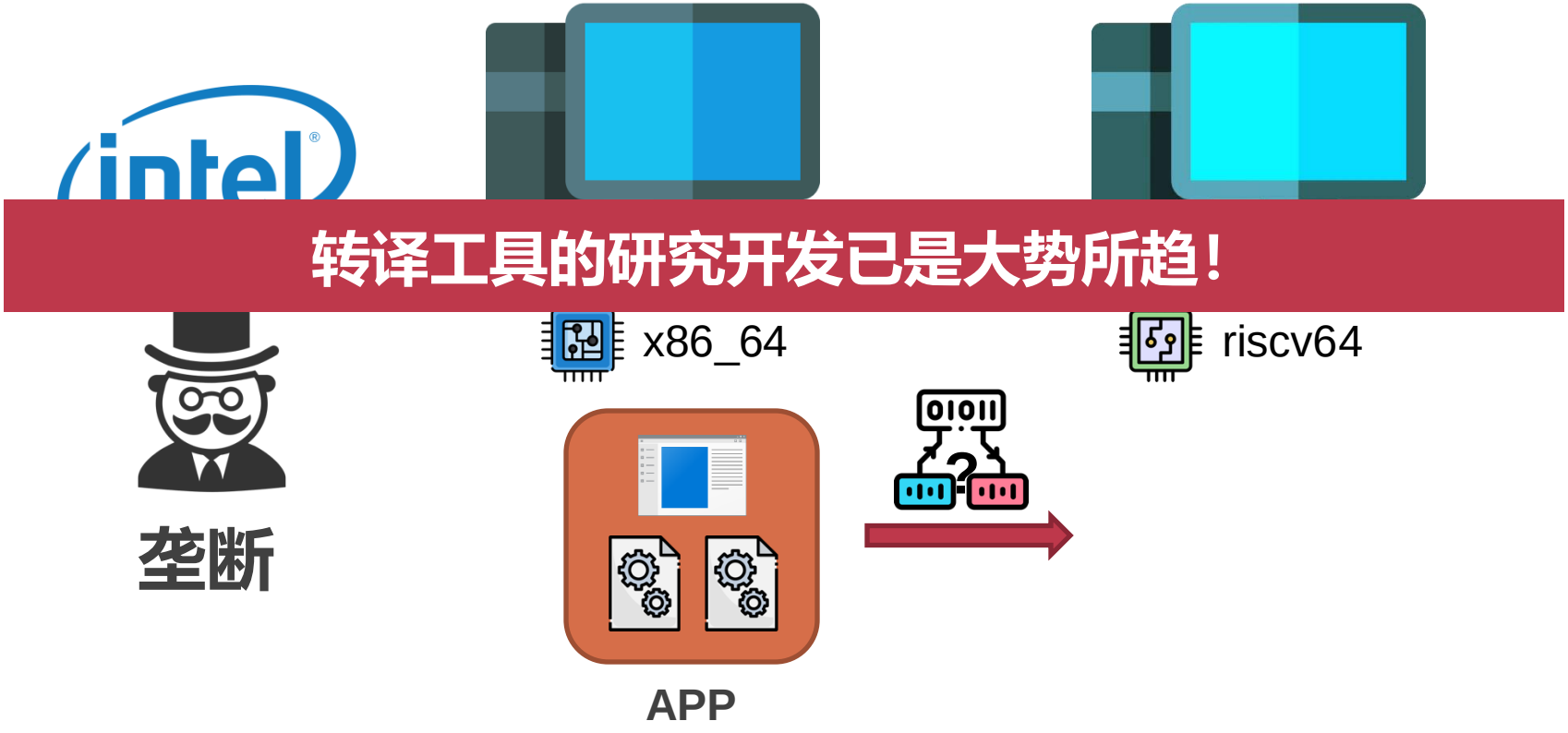
QEMU 调用本机动态库加速转译

上海交通大学 章子杨

转译工具的需求与现状

研究背景

转译工具的需求



现有的转译工具

Apple Rosetta 2

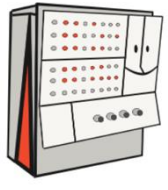
Windows On Arm

Huawei ExaGear

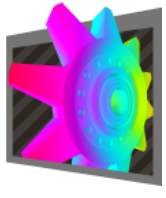
统统闭源

QEMU

Box64



Blinkenlights

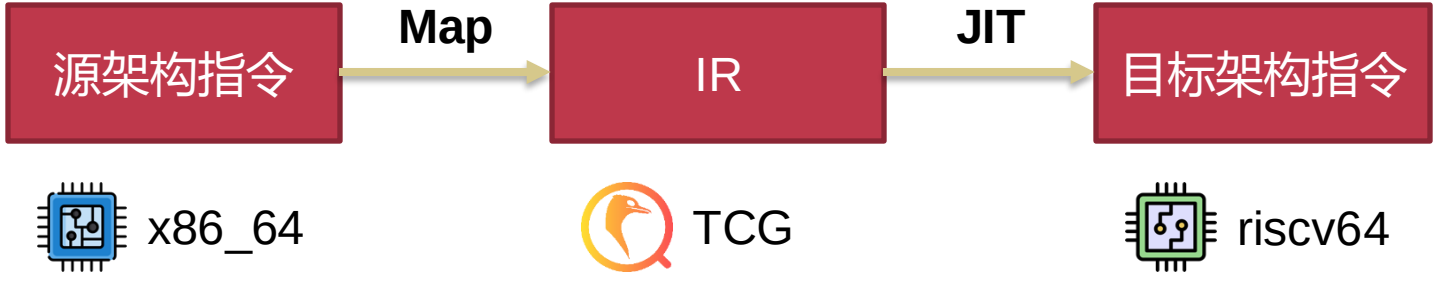


Fex-Emu

各有所长

研究现状

- 动态翻译具有更高的灵活性与用户体验，是主流选择。
- 动态二进制翻译器，备受关注的指标主要有执行效率、兼容性。
- 在现有的二进制翻译相关的学术研究与工程实现中，大部分优化措施，都集中在优化翻译规则上。



探索新的转译流程

以 x86_64 翻译到 riscv64 为例

现状是，有些库是无法模拟的，如包含 CUDA 指令的动态库。
还有很多库是开源的，如 OpenGL、SDL，每个平台都能编译。

x86_64 源程序

完全模拟

痴于在“翻译规则”之
窠臼中上下求索

x86_64 源程序

部分模拟

将“物尽其用”思想
贯彻到底

使用本机动态库加速的基本方案（以 Box64 为例）

基本方案

本机优势

例：做一个矩阵乘法，如果“乘法”代码来自 Native 动态库，显然速度更快，因为充分发挥了本机的向量化优势，也没有转译开销。

效率优势：执行相同的逻辑，本地机器码直接执行比转译快得多

兼容性优势：支持 CUDA 指令、驱动指令等

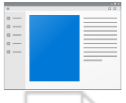
```
    int B[100 * 100];
    int C[100 * 100];

    // 初始化 A、B...

    matrix_mul(A, B, Result: C, Width: 100, Height: 100);

    printf(format: "%d\n", C[0]);

    return 0;
}
```

libmatrix.so

利用本地动态库的可行性

```c
main.c > ...
#include <stdio.h>

void matrix_mul(int A[], int B[], int Result[], int Width, int Height);

int main(int argc, char *argv[]) {
    int A[100 * 100];
    int B[100 * 100];
    int C[100 * 100];

    // 初始化 A、B...

    matrix_mul(A, B, Result: C, Width: 100, Height: 100);

    printf(format: "%d\n", C[0]);

    return 0;
}
```

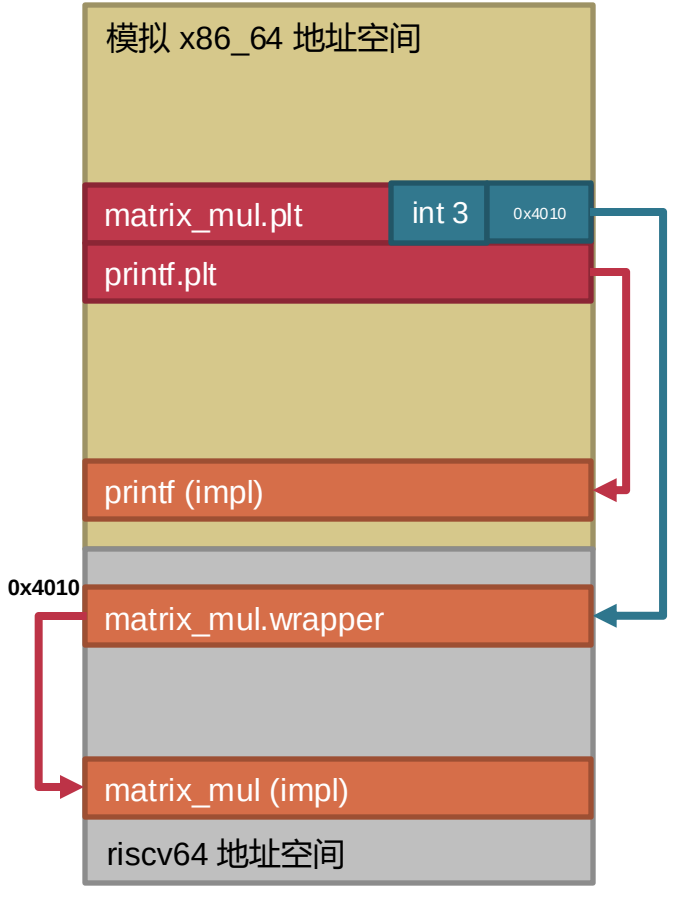
以 Box64 为例



main

libmatrix.so
libc.so

libmatrix.so



使用本机动态库加速的核心问题（以 Box64 为例）

核心问题

核心问题

- **1. 访存**
  - 对于同一个内存地址，Guest 与 Host 访问必须能获取到完全相同的数据

核心问题 - 访存

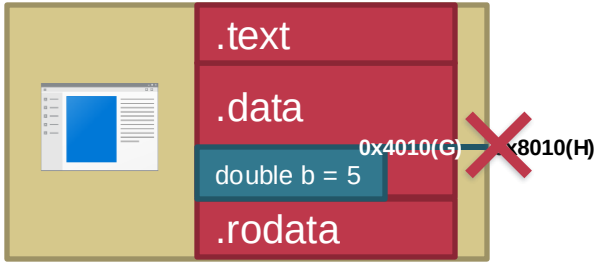
- 条件 1：Guest 的地址空间映射，不存在偏移量。

```c
#include <stdio.h>
#include <stdint.h>

struct sample {
    uint64_t a[2];
    double b;
    char c;
    int i;
};

void print_double(struct sample *s) {
    printf(format: "b: %f\n", s->b);
}
```

```
00000000000004c0 <print_double>:
 4c0:   6910                    ld   a2,16(a0)
 4c2:   00000597            auipc   a1,0x0
 4c6:   0a658593            addi    a1,a1,166 # 568 <matrix_mul+0x9a>
 4ca:   4505                    li   a0,1
 4cc:   bf91                    j    420 <__printf_chk@plt>
```

sample.so

Native 库访问 Guest 地址，访问到的必须是正确的值。

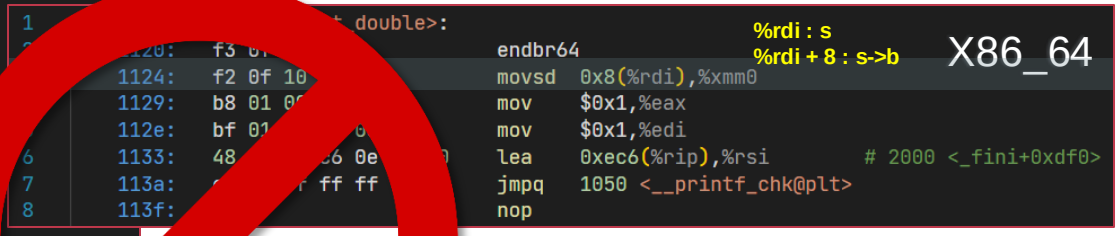


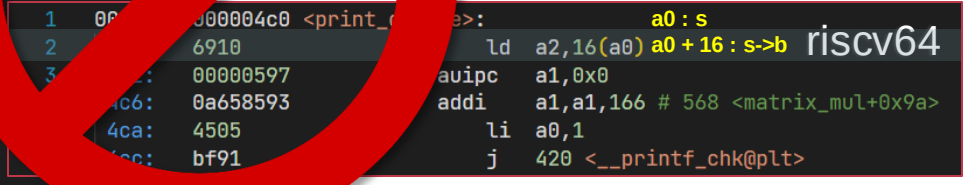

```
> qemu-x86_64 --help | grep GUEST_BASE
-B address            QEMU_GUEST_BASE      set guest_base address to 'address'
```

核心问题 - 访存

- 条件 2：动态库的所有导出变量与函数参数中，不存在字段随架构变化的结构体

```c
#include <stdio.h>
#include <stdint.h>

struct sample {
#ifdef __x86_64
    uint64_t a[1];
#else
    uint64_t a[2];
#endif
    double b;
    char c;
    int i;
};

void print_double(struct sample *s)
    printf(format: "b: %f\n", s->b);
}
```





核心问题

- **1. 访存**
  - 对于同一个内存地址，Guest 与 Host 访问必须能获取到完全相同的数据
- **2. 函数调用**
  - Guest 代码调用 Host 函数时，Host 函数能正确地取出参数，之后 Guest 代码能正确地获取返回值

核心问题 - 函数调用

- 条件 3：调用约定能被正确转换

模拟 x86_64 地址空间

main.c

```c
#include <stdio.h>

void matrix_mul(int A[], int B[], int Result[], int Width, int Height);

int main(int argc, char *argv[]) {
    int A[100 * 100];
    int B[100 * 100];
    int C[100 * 100];

    // 初始化 A、B...

    matrix_mul(A, B, Result: C, Width: 100, Height: 100);

    printf(format: "%d\n", C[0]);

    return 0;
}
```

调用

matrix_mul.plt | int 3 | 0x4010

陷阱

matrix.c

```c
#include <stdio.h>
#include <stdint.h>

__attribute__((visibility("default"))) void matrix_mul(
    int A[], int B[], int Result[], int Width, int Height) {
    for (int i = 0; i < Height; i++) {
        for (int j = 0; j < Width; j++) {
            Result[i * Width + j] = 0;
            for (int k = 0; k < Width; k++) {
                Result[i * Width + j] +=
                    A[i * Width + k] * B[k * Width + j];
            }
        }
    }
}
```

libmatrix.so

转发

box64

```c
typedef union {
    int64_t sq[1];
    uint64_t q[1];
    int32_t sdword[2];
    uint32_t dword[2];
    int16_t sword[4];
    uint16_t word[4];
    int8_t sbyte[8];
    uint8_t byte[8];
} reg64_t;
```

解包

```c
enum {
    _RAX, _RCX, _RDX, _RBX, _RSP, _RBP, _RSI, _RDI,
    _R8, _R9, _R10, _R11, _R12, _R13, _R14, _R15
};

void (*matrix_mul)(int [], int B[], int Result[], int Width, int Height);

void matrix_mul__wrapper(struct X64State *cpu) {
    matrix_mul(
        (int *) cpu->regs[_RDI].q,
        (int *) cpu->regs[_RSI].q,
        (int *) cpu->regs[_RDX].q,
        (int) cpu->regs[_RCX].sdword[0],
        (int) cpu->regs[_R8].sdword[0]
    );
}
```

陷入包装器

matrix_mul.wrapper

riscv64 地址空间

核心问题

- **1. 访存**
  - 对于同一个内存地址，Guest 与 Host 访问必须能获取到完全相同的数据
- **2. 函数调用**
  - Guest 代码调用 Host 函数时，Host 函数能正确地取出参数，之后 Guest 代码能正确地获取返回值
- **3. 函数指针**
  - Guest 代码传递一个 Guest 函数指针给 Host 函数，Host 函数进行回调时，控制流能正确地回到 Guest 中

核心问题 – 函数指针

- 条件 4：函数指针能被正确处理

核心问题

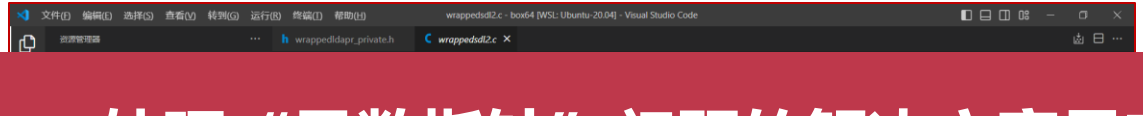
- **1. 访存**
  - 对于同一个内存地址，Guest 与 Host 访问必须能获取到完全相同的数据
- **2. 函数调用**
  - Guest 代码调用 Host 函数时，Host 函数能正确地取出参数，之后 Guest 代码能正确地获取返回值
- **3. 函数指针**
  - Guest 代码传递一个 Guest 函数指针给 Host 函数，Host 函数进行回调时，控制流能正确地回到 Guest 中
- **4. 线程创建与销毁（略）**
- **5. 可变长参数函数（略）**

复杂性与优缺点

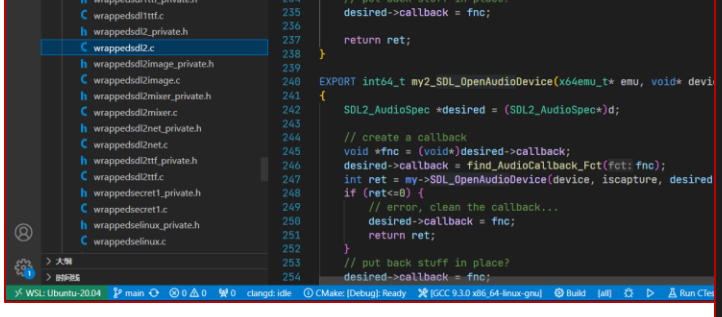
BOX64 的评价

Box64 的复杂性

- Box64 为大量第三方库的各函数都实现了包装器，开发工程规模极大



Box64 处理“函数指针”问题的解决方案是在包装器中临时替换，这一步骤与函数功能存在强耦合，无法被自动化处理。



```
desired->callback = fnc;

return ret;
}

EXPORT int64_t my2_SDL_OpenAudioDevice(x64emu_t* emu, void* devi
{
    SDL2_AudioSpec *desired = (SDL2_AudioSpec*)d;

    // create a callback
    void *fnc = (void*)desired->callback;
    desired->callback = find_AudioCallback_Fct(fct: fnc);
    int ret = my->SDL_OpenAudioDevice(device, iscapture, desired
    if (ret<=0) {
        // error, clean the callback...
        desired->callback = fnc;
        return ret;
    }
    // put back stuff in place?
    desired->callback = fnc;
```

```
    void *fnc = (void*)desired->callback;
    desired->callback = find_AudioCallback_Fct(fct: fnc);
    int ret = my->SDL_OpenAudio(desired, (SDL2_AudioSpec*)o);
    if (ret!=0) {
        // error, clean the callback...
        desired->callback = fnc;
        return ret;
    }
    // put back stuff in place?
    desired->callback = fnc;

    return ret;
}
```

Wrapper 的内容：

1. 调用约定转换
2. 函数指针临时替换

Box64 的优缺点

- **优点：**
  - 开辟了使用 Native 库加速转译的先河
    - 验证了可行性，提供了基本实现方法
    - 设计的 DynaRec （动态重编译）JIT 技术具有较好的性能
- **缺点：**
  - 开发工作量巨大
    - 从零开发动态链接器、转译器
    - **人工实现所有的 Wrapper（方案缺陷导致）**
  - 灵活性极差
    - 开发者只能手动为特定版本的库做适配，无法应对库的更新
  - 故障率极高

使 QEMU 具备调用本机动态库加速的方法

基于 QEMU 的加速方案

实现要素

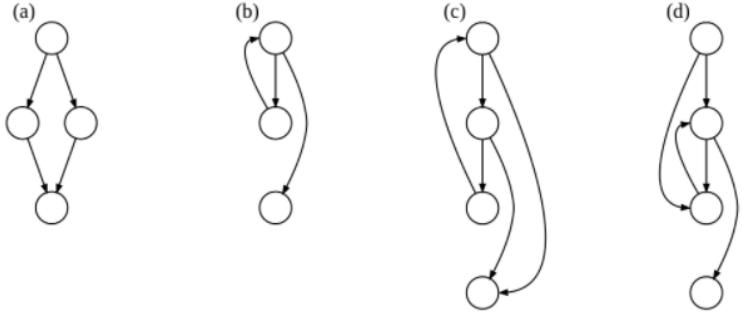
- **物尽其用**
  - **源代码复用：系统库，C/C++ 标准库，开源第三方库的源代码**能被充分利用
  - **框架复用：**不重复造轮子，尽可能复用现有的框架，如 QEMU 与 ld-linux
- **可维护性**
  - **谨慎修改：**如果必须对现有软件的源代码进行修改，那么修改应尽可能少，减少出故障的可能性
- **可自动化**
  - **可批处理：**软件的移植，必须能够使用通用的算法流程批处理实现

自动化方案

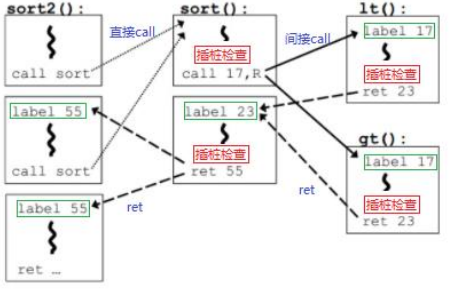
- **我们基于 QEMU 设计的方案：**
  - 修改 QEMU，修改之使之能调用本地动态库
  - 修改动态链接器，使之能正确实现 Guest 库到 Native 库的绑定
  - **修改编译器，使之能为每个函数指针调用生成检查代码**
  - **开发生成器，自动化生成包装器**

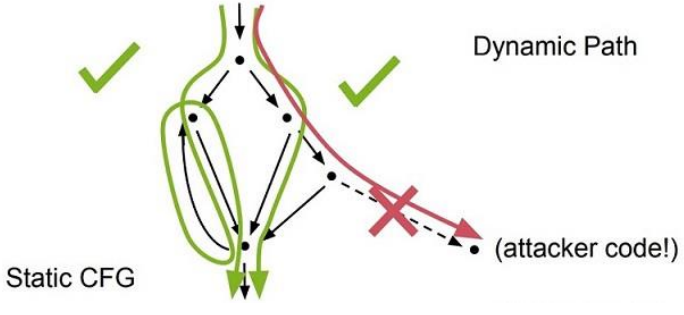
关于 CFI/CFG

- 在跳转代码前插装检查目标地址是否合法




```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```





原始间接jmp指令：

Opcode bytes	Source Instructions		Destination Opcode bytes	Destination Instructions	
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst

ecx保存的目的地址

can be instrumented as (a):

cfi插桩检查，方式(a)：

Opcode bytes	Source Instructions		Destination Opcode bytes	Destination Instructions	
81 39 78 56 34 12	cmp [ecx], 12345678h	; comp ID & dst	78 56 34 12	; data 12345678h	; ID
75 13	jne error_label	; if != fail	8B 44 24 04	mov eax, [esp+4]	; dst
8D 49 04	lea ecx, [ecx+4]	; skip ID at dst	...		
FF E1	jmp ecx	; jump to dst			

插桩检查

插入 `ID` = 12345678h

or, alternatively, instrumented as (b):

cfi插桩检查，方式(b)：

Opcode bytes	Source Instructions		Destination Opcode bytes	Destination Instructions	
B8 77 56 34 12	mov eax, 12345677h	; load ID-1	3E 0F 18 05	prefetchnta	; label
40	inc eax	; add 1 for ID	78 56 34 12	[12345678h]	; ID
39 41 04	cmp [ecx+4], eax	; compare w/dst	8B 44 24 04	mov eax, [esp+4]	; dst
75 13	jne error_label	; if != fail	...		
FF E1	jmp ecx	; jump to label			

插桩检查

插入 `label ID` = prefetchnta 12345678h

自动生成包装器

LLVM Tooling

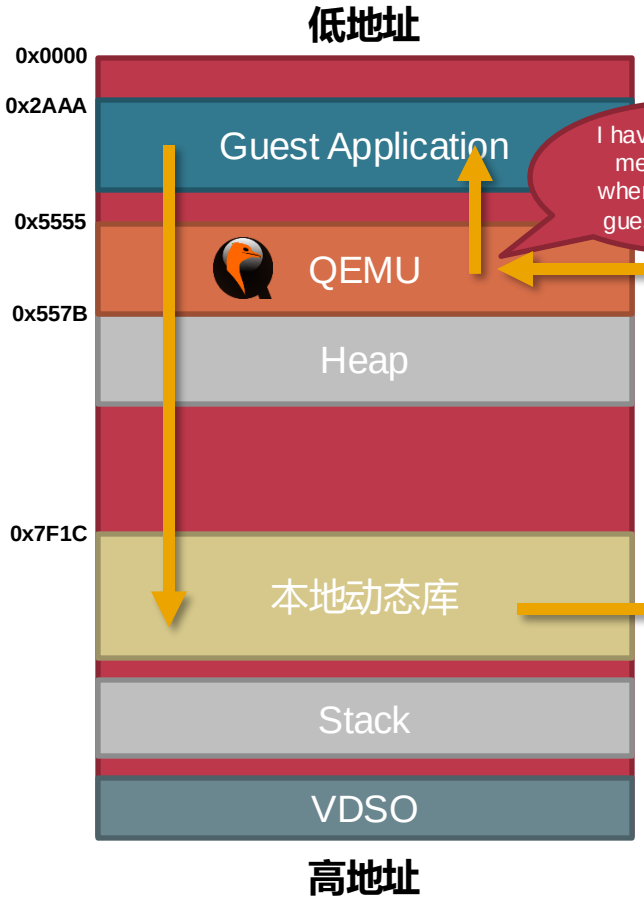
```h
// qsort.h
#ifndef QSORT_H
#define QSORT_H

#include <stddef.h>

typedef int (*Compare)(const void *, const void *);

void qsort(void *arr, size_t num, Compare comp);

#endif // QSORT_H
```


```c
typedef union {
    int64_t sq[1];
    uint64_t q[1];
    int32_t sdword[2];
    uint32_t dword[2];
    int16_t sword[4];
    uint16_t word[4];
    int8_t sbyte[8];
    uint8_t byte[8];
} reg64_t;
```

```c
enum {
    _RAX, _RCX, _RDX, _RBX, _RSP, _RBP, _RSI, _RDI,
    _R8, _R9, _R10, _R11, _R12, _R13, _R14, _R15,
};

typedef int (*Compare)(const void *, const void *);
void (*qsort)(void *arr, size_t num, Compare comp);

void qsort__wrapper(struct X64State *cpu) {
    qsort(
        (int *) cpu->regs[_RDI].q,
        (size_t) cpu->regs[_RSI].q,
        (Compare) cpu->regs[_RDX].q
    );
}
```

```c
// qsort.c
#include <stdio.h>
#include <stdint.h>

typedef int (*Compare)(const void *, const void *);

__attribute__((visibility("default"))) void qsort(
    void *arr, size_t num, Compare comp) {
    void **array = (void **) arr;
    size_t j;
        if (comp(array[j], &array[j + 1]) > 0) {
            //
        }

    // ...
}
```

函数指针的检查



I have exported a method, call it when you meet a guest callback!!!

```
6   #ifdef __cplusplus
7   extern "C" {
8   #endif
9   typedef void (*GuestEntry)(void */*thunk*/, void */*callback*/,
10                             void */*args*/, void */*ret*/);
11  extern GuestEntry GetGuestEntry();
12  extern void *LookUpGuestThunk(const char *sign);
13  #ifdef __cplusplus
14
15  #endif
16
17  static int _check_comp(Compare comp, void *x, void *y) {
18      static void *thunk = LookUpGuestThunk(sign: "int(void*,void*)");
19      static GuestEntry entry = GetGuestEntry();
20      if ((uintptr_t) comp > (uintptr_t) entry) {
21          return comp(x, y);
22      }
23
24      int ret = 0;
25      void *a[] = { [0]=&x, [1]=&y };
26      entry(thunk, (void *) comp, a, &ret);
27      return ret;
28  }
29
30  __attribute__((visibility("default"))) void qsort(
31      void *arr, size_t num, Compare comp) {
32      void **array = (void **) arr;
33      size_t j;
34          if (_check_comp(comp, x: array[j], y: array[j + 1]) > 0) {
35              //
36          }
37
38      // ...
39  }
```

```
if (comp(array[j], &array[j + 1]) > 0) {
```

目前已有的数据

- **测试程序：FFmpeg（可执行文件），libav*.so、libsw*.so 等动态库。**
- **实验：将一首长度为 00:06:31 的 WAV 文件分别重采样为 Wav 和 Mp3 文件，统计三种不同的方式所耗的时间。（MP3 的编解码器是 lame 库）**
  - Native：直接本地执行
  - QEMU：QEMU 对 ffmpeg、libavcodec.so、libswresample.so 等所有二进制文件全部模拟执行
  - QEMU-NC（QEMU Native Compatible）：QEMU 对 ffmpeg 模拟执行，调用 libavcodec.so 等本地动态库中的函数
- **命令行：**
  - ffmpeg –i 挪威的森林.wav –y 挪威的森林_out.wav（读取->写入）
  - ffmpeg –i 挪威的森林.wav –y 挪威的森林_out.mp3（读取->解码->重采样->编码->写入）

x86_64 自己翻译自己

测试环境：Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz

Wav -> Wav	时间 ms	相对 Native
Native	139.8	1
QEMU	1248.8	8.93
QEMU-NC	428.1	3.06

QEMU-NC

本地函数调用次数：776315

参数解包 + 本地库：164.2 ms（占总时间 38.3 %）

Wav -> Mp3	时间 ms	相对 Native
Native	5293.4	1
QEMU	75886.4	14.34
QEMU-NC	5626.7	1.06

QEMU-NC

本地函数调用次数：746639

参数解包 + 本地库：5345.0 ms （占总时间 95.0 %）

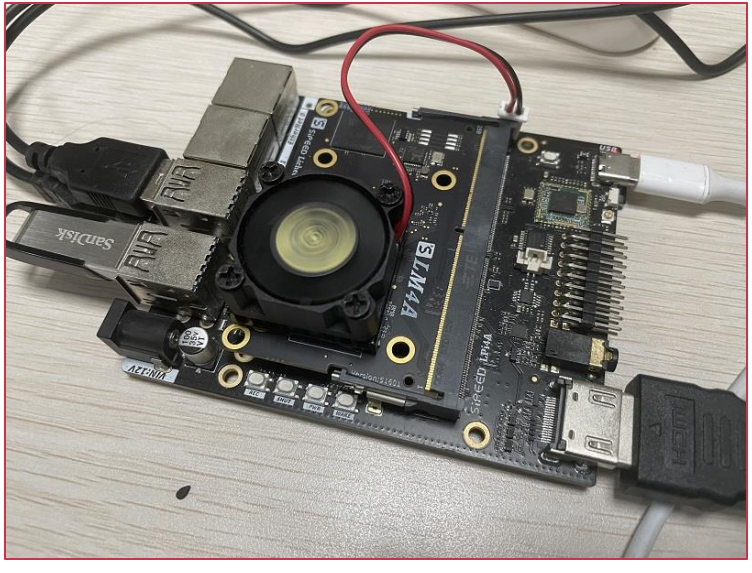
riscv64 开发板转译 x86_64

测试环境：Sipeed LicheePi 4A（矽速科技，荔枝派 4A）1.848GHz

Wav -> Wav	时间 s	相对 Native
Native	1.197	1
QEMU	30.577	25.54
QEMU-NC	4.715	3.94

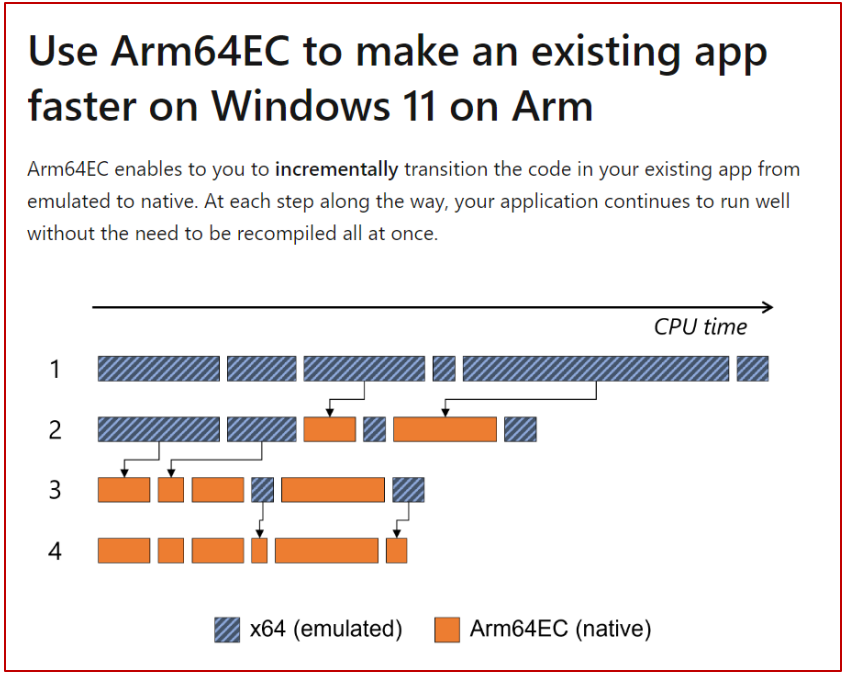
Wav -> Mp3	时间 s	相对 Native
Native	2.964	1
QEMU	约 1940	654.52
QEMU-NC	6.368	2.15

SiPEED 矽速科技



参考资料

https://learn.microsoft.com/en-us/windows/arm/arm64ec



Windows 11 Arm64EC

https://www.qemu.org/docs/master/user

QEMU

https://github.com/ptitSeb/box64

Box64

谢谢大家！

上海交通大学 章子杨