工程实践与科技创新 III-D 作业 4

姓名:侯晟元　　学号:518021910604

内容一要求(满分 20 分)：

1. 通过测试发现虚拟化的性能损耗（与原生性能对比）
2. 通过阅读文献等途径，修改开源软件的相关代码，降低虚拟化的性能损耗。

QEMU-KVM 简介:

KVM 负责 cpu 虚拟化+内存虚拟化，实现了 cpu 和内存的虚拟化，但 kvm 并不能模拟其他设备，还必须有个运行在用户空间的工具才行。KVM 的开发者选择了比较成熟的开源虚拟化软件 QEMU 来作为这个工具，QEMU 模拟 IO 设备（网卡，磁盘等），对其进行了修改，最后形成了 QEMU-KVM。

https://www.cnblogs.com/echo1937/p/7138294.html

一.通过测试发现虚拟化的性能损耗（与原生性能对比）

本次实验我们采用 sysbench 比较原生虚拟机和在 QEMU-KVM 环境下创建的虚拟机(利用加载了 KVM 模块的 QEMU 虚拟化技术)的性能。Sysbench 是一款开源多线程的测试工具，可以执行 CPU，内存，线程，IO，数据库等方面的性能测试。为了更好地控制变量分析实验结果，我们两台虚拟机均采用统一操作系统 Ubuntu18.04，并采用相同的测试指令。

首先在作业二中下载 qemu 源代码(版本 5.1.0)，并编译安装，注意在设置配置文件时需要加上--enable-kvm 命令行选项，使得 QEMU 可以利用 KVM 模块。

```
root@ubuntu:/qemu-5.1.0# ./configure --enable-kvm --enable-debug --enable-vnc --
enable-werror --target-list="x86_64-softmmu"
```

编译(make)安装(make install)后确认安装成功

```
qemu-img version 5.1.0
Copyright (c) 2003-2020 Fabrice Bellard and the QEMU Project developers
root@ubuntu:/qemu-5.1.0#
```

确认 KVM 模块是否可以使用

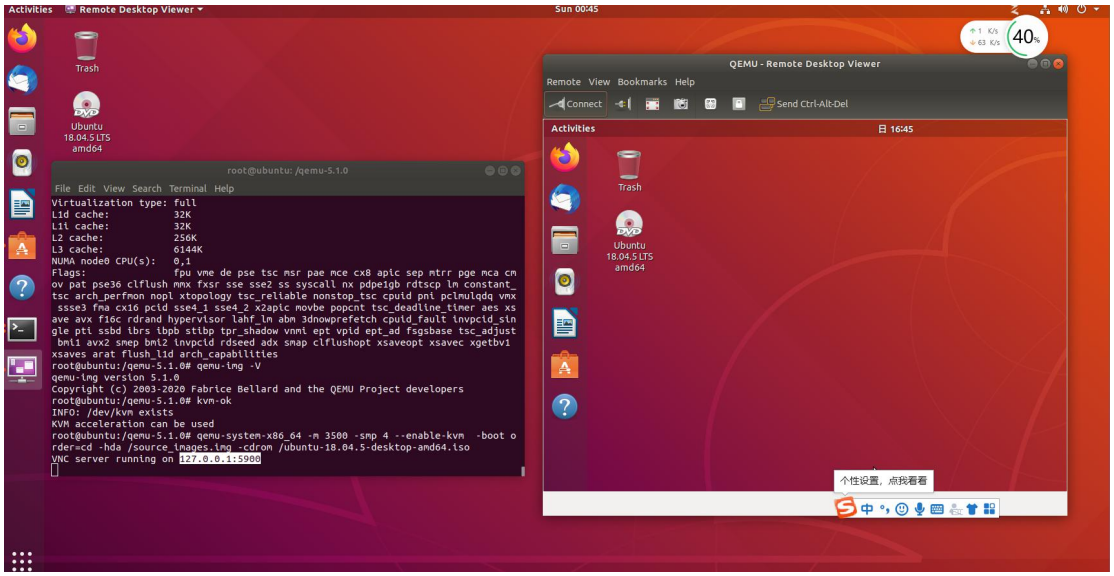
```
root@ubuntu:/qemu-5.1.0# kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
```

利用 qemu_system_x86 指令创建 Ubuntu18.04 四核虚拟机，注意同样要加上--enable-kvm 以设置加载 KVM 模块

```
root@ubuntu:/qemu-5.1.0# qemu-system-x86_64 -m 3500 -smp 4 --enable-kvm  -boot d
 -hda /source_images.img -cdrom /ubuntu-18.04.5-desktop-amd64.iso
```

安装完成后在 VNC 下打开虚拟机效果如下

```
root@ubuntu:/qemu-5.1.0# qemu-system-x86_64 -m 3500 -smp 4 --enable-kvm  -boot o
rder=cd -hda /source_images.img -cdrom /ubuntu-18.04.5-desktop-amd64.iso
VNC server running on 127.0.0.1:5900
```



在原生虚拟机和加载 KVM 模块的 QEMU 虚拟机分别下载安装 sysbench

```
root@skywalker:/# apt get install sysbench
```

```
root@skywalker:/# sysbench --version
sysbench 1.0.11
```

安装完成后，使用 sysbench 的测试指令进行测试

(1)CPU 性能测试

最大质数发生器测试(最大质数不超过 20000，线程数量为 1)

```
root@ubuntu:/qemu-5.1.0# sysbench cpu --cpu-max-prime=20000 run
```

原生虚拟机

```
Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:   393.65

General statistics:
    total time:                          10.0019s
    total number of events:              3938

Latency (ms):
         min:                                    2.22
         avg:                                    2.54
         max:                                    9.25
         95th percentile:                        3.25
         sum:                                 9994.05

Threads fairness:
    events (avg/stddev):           3938.0000/0.00
    execution time (avg/stddev):   9.9941/0.00
```

QEMU-KVM 虚拟机

```
Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:   299.68

General statistics:
    total time:                          10.0015s
    total number of events:              2998

Latency (ms):
         min:                                  2.28
         avg:                                  3.33
         max:                                 21.31
         95th percentile:                      6.09
         sum:                               9973.49

Threads fairness:
    events (avg/stddev):           2998.0000/0.00
    execution time (avg/stddev):   9.9735/0.00
```

比较原生虚拟机和在 QEMU-KVM 环境中安装的虚拟机，可以发现原生虚拟机每秒处理的事件数量(393.65)要超过 QEMU-KVM 虚拟机(299.68)将近 30%。测试总的执行时间上二者几乎没有差异，这是因为原生虚拟机处理的事件量(3938)要比 QEMU-KVM 虚拟机更多(2998)。而在时延方面，QEMU-KVM 虚拟机在各方面的指标(avg=3.33ms,max=21.31ms)都差原生虚拟机(avg=2.54ms,max=9.25ms)不少。

以上比较说明了，在 CPU 性能方面，原生虚拟机要大大优于 QEMU-KVM。例如，原生虚拟机 CPU 速度比 QEMU-KVM 虚拟机快了近 30%，最大时延只有 QEMU-KVM 虚拟机的一半，而平均时延则仅仅大约是 QEMU-KVM 虚拟机的 2/3。但是由于 QEMU-KVM 通过一些技术减少了处理的事件数，抵消了虚拟化性能损耗带来的一些影响。

(2) 内存性能测试

向内存中传输 2G 的数据，每个块的大小为 8K。

```
root@ubuntu:/qemu-5.1.0# sysbench memory --memory-block-size=8k --memory-total-size=2G run
```

原生虚拟机

```
Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 8KiB
  total size: 2048MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 262144 (1214234.75 per second)

2048.00 MiB transferred (9486.21 MiB/sec)


General statistics:
    total time:                          0.2139s
    total number of events:              262144

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                    4.10
         95th percentile:                        0.00
         sum:                                  175.61

Threads fairness:
    events (avg/stddev):           262144.0000/0.00
    execution time (avg/stddev):   0.1756/0.00
```

QEMU-KVM 虚拟机

```
Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 8KiB
  total size: 2048MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 262144 (478681.62 per second)

2048.00 MiB transferred (3739.70 MiB/sec)


General statistics:
    total time:                          0.5433s
    total number of events:              262144

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                   12.87
         95th percentile:                        0.00
         sum:                                  413.41

Threads fairness:
    events (avg/stddev):           262144.0000/0.00
    execution time (avg/stddev):   0.4134/0.00
```

比较原生虚拟机和在 QEMU-KVM 环境中安装的虚拟机的内存性能，可以发现向内存中传输 2G 数据，二者需要相同的操作数，但是原生虚拟机传输速率(9486.21MB/s)几乎是 QEMU-KVM 虚拟机(3739.70MB/s)的 2.5 倍，总的执行时间也不到 QEMU-KVM 虚拟机的一半。时延上二者的传输速度都相当快，以至于平均时延接近为 0，但是原生虚拟机的最大时延(4.10ms)只有 QEMU-KVM 虚拟机(12.87ms)的 1/3。

以上比较说明了，在内存性能上，原生虚拟机同样要大大优于 QEMU-KVM。例如，虽然二者操作数相同，但数据传输速率比 QEMU-KVM 虚拟机快了近 1.5 倍，因此执行时间更短。最大时延原生虚拟机只有 QEMU-KVM 虚拟机的 1/3。但

是在平均时延上，二者都表现出了非常好的性能。Threads fairness 上，二者事件数相同，但是原生虚拟机的平均处理时间性能优于后者不少。

(3)文件 IO 性能测试

1.准备阶段:首先生成需要测试的文件，生成的小文件都存放在当前目录下

```
root@ubuntu:/qemu-5.1.0# sysbench fileio --threads=2 --file-total-size=1G --file-test-mode=rndrw prepare
```

原生虚拟机

```
Creating file test_file.113
Creating file test_file.114
Creating file test_file.115
Creating file test_file.116
Creating file test_file.117
Creating file test_file.118
Creating file test_file.119
Creating file test_file.120
Creating file test_file.121
Creating file test_file.122
Creating file test_file.123
Creating file test_file.124
Creating file test_file.125
Creating file test_file.126
Creating file test_file.127
1073741824 bytes written in 1.91 seconds (537.15 MiB/sec).
```

QEMU-KVM 虚拟机

```
Creating file test_file.106
Creating file test_file.107
Creating file test_file.108
Creating file test_file.109
Creating file test_file.110
Creating file test_file.111
Creating file test_file.112
Creating file test_file.113
Creating file test_file.114
Creating file test_file.115
Creating file test_file.116
Creating file test_file.117
Creating file test_file.118
Creating file test_file.119
Creating file test_file.120
Creating file test_file.121
Creating file test_file.122
Creating file test_file.123
Creating file test_file.124
Creating file test_file.125
Creating file test_file.126
Creating file test_file.127
1073741824 bytes written in 17.15 seconds (59.72 MiB/sec).
root@skywalker:/#
```

从文件生成阶段就可以看出，生成相同数量的文件，原生虚拟机的生成速度(537.15MB/s)要远远快于 QEMU-KVM 虚拟机(59.72MB/s),导致二者在文件 IO 的时间上差距巨大(一个是 1.91s，一个是 17.15s)。这说明在文件的创建操作上，原生虚拟机的性能相比于 QEMU-KVM 虚拟机具有显著的优势。

2.运行阶段(测试随机读写大小总共 1G 的文件，线程数量为 2)

```
root@ubuntu:/qemu-5.1.0# sysbench fileio --threads=2 --file-total-size=1G --file-test-mode=rndrw run
```

原生虚拟机

```
Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Extra file open flags: 0
128 files, 8MiB each
1GiB total file size
Block size 16KiB
Number of IO requests: 0
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync() each 100 requests.
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random r/w test
Initializing worker threads...

Threads started!


File operations:
    reads/s:                      21278.47
    writes/s:                     14185.65
    fsyncs/s:                     45382.07

Throughput:
    read, MiB/s:                  332.48
    written, MiB/s:               221.65

General statistics:
    total time:                          10.0001s
    total number of events:              808596

Latency (ms):
         min:                                    0.00
         avg:                                    0.02
         max:                                    9.03
         95th percentile:                        0.10
         sum:                                19617.63

Threads fairness:
    events (avg/stddev):           404298.0000/22216.00
    execution time (avg/stddev):   9.8088/0.01
```

QEMU-KVM 虚拟机

```
Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Extra file open flags: 0
128 files, 8MiB each
1GiB total file size
Block size 16KiB
Number of IO requests: 0
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync() each 100 requests.
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random r/w test
Initializing worker threads...

Threads started!


File operations:
    reads/s:                      419.84
    writes/s:                     279.90
    fsyncs/s:                     889.67

Throughput:
    read, MiB/s:                  6.56
    written, MiB/s:               4.37

General statistics:
    total time:                          10.0008s
    total number of events:              15900

Latency (ms):
         min:                                    0.00
         avg:                                    1.25
         max:                                   47.61
         95th percentile:                        4.65
         sum:                                19925.03

Threads fairness:
    events (avg/stddev):           7950.0000/25.00
    execution time (avg/stddev):   9.9625/0.00
```

对 1G 数据文件进行随机读写操作时，原生虚拟机读取文件的操作速度(21278.47op/s)和写入操作速度(14185.65op/s)远远要快于 QEMU-KVM 虚拟机读(419.84op/s)写(279.90op/s)，是后者的将近 50 倍。在吞吐量上，原生虚拟机读写吞吐量同样展现出了惊人的效率，也快于后者近 50 倍。但是二者总的执行时间却近似相同，观察可以发现，这是因为原生虚拟机虽然执行操作速度快，吞吐量高，但是需要执行的事件也多，这一点也可以解释为何二者的总时延非常接近，但是平均时延和最大时延还是原生虚拟机要更低。

由此可以说明，在文件和磁盘的读写性能方面。原生虚拟机在总的执行时间和时延上其实和 QEMU-KVM 虚拟机没有太大区别，但是前者的操作数量和事件数要远多于后者，同时读写吞吐量和操作处理速度前者也要远快于后者，这在一定程度上产生了某种平衡的效果。

3.运行测试完毕后需要将临时文件清理回收

```
root@ubuntu:/qemu-5.1.0# sysbench fileio --threads=2 --file-total-size=1G --file-test-mode=rndrw cleanup
sysbench 1.0.11 (using system LuaJIT 2.1.0-beta3)

Removing test files...
```

由于清除操作未显示性能数据，此处不再对原生和 QEMU-KVM 虚拟机进行比较。

(4) 互斥锁性能测试

线程数量为 2，互斥数组的总大小 4096，每个线程互斥锁的数量为 50000，互斥锁外部的空循环数量为 10000，模拟所有线程在同一时刻并发运行。

```
root@ubuntu:/qemu-5.1.0# sysbench mutex --threads=2 --mutex-num=4096 --mutex-locks=50000 --mutex-loops=10000 run
```

原生虚拟机

```
root@ubuntu:/qemu-5.1.0# sysbench mutex --threads=2 --mutex-num=4096 --mutex-locks=50000 --mutex-loops=10000 run
sysbench 1.0.11 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Initializing worker threads...

Threads started!


General statistics:
    total time:                          0.2955s
    total number of events:              2

Latency (ms):
         min:                                  285.87
         avg:                                  288.37
         max:                                  290.87
         95th percentile:                      292.60
         sum:                                  576.74

Threads fairness:
    events (avg/stddev):           1.0000/0.00
    execution time (avg/stddev):   0.2884/0.00
```

QEMU-KVM 虚拟机

```
sysbench 1.0.11 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Initializing worker threads...

Threads started!


General statistics:
    total time:                          0.4752s
    total number of events:              2

Latency (ms):
         min:                                  426.13
         avg:                                  447.68
         max:                                  469.22
         95th percentile:                      467.30
         sum:                                  895.35

Threads fairness:
    events (avg/stddev):           1.0000/0.00
    execution time (avg/stddev):   0.4477/0.02
```

观察可以得到，在互斥锁的性能方面，原生虚拟机执行测试程序的时间(0.2955s)和平均时延(avg=288.37ms)，最大时延(max=290.87ms)都接近于QEMU-KVM 虚拟机(total time=0.4752s,avg=447.68ms,max=469.22ms)的 3/4。这些数据说明了在互斥锁方面，原生虚拟机的线程管理要比 QEMU-KVM 虚拟机的线程管理更高效，因此多线程异步执行的性能更好。

(5) 多线程性能测试

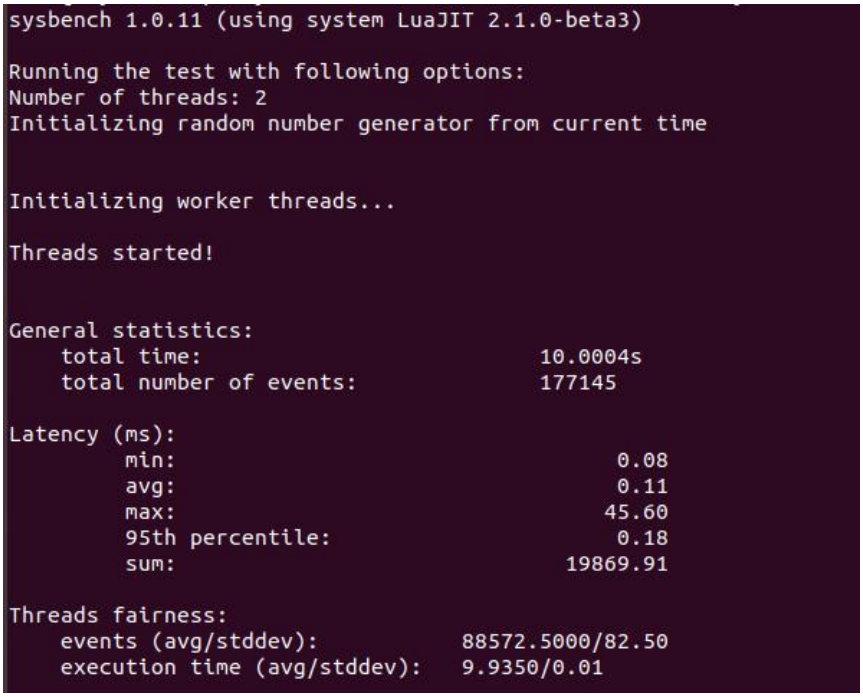
线程数量为 2，每个请求产生 100 个线程，每个线程拥有锁的数量为 4

```
root@ubuntu:/qemu-5.1.0# sysbench threads --threads=2 --thread-yields=100 --thread-locks=4 run
```

原生虚拟机

```
root@ubuntu:/qemu-5.1.0# sysbench threads --threads=2 --thread-yields=100 --thread-locks=4 run
sysbench 1.0.11 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Initializing worker threads...

Threads started!


General statistics:
    total time:                          10.0001s
    total number of events:              365419

Latency (ms):
         min:                                    0.04
         avg:                                    0.05
         max:                                   13.82
         95th percentile:                        0.09
         sum:                                19847.01

Threads fairness:
    events (avg/stddev):           182709.5000/627.50
    execution time (avg/stddev):   9.9235/0.00
```

QEMU-KVM 虚拟机



观察实验结果可以得知，原生虚拟机和 QEMU-KVM 虚拟机运行多线程测试程序的执行时间差别不大，仅有 0.0003s，这同时也体现在了二者总时延上的相近。继续观察可以进一步发现，这种执行时间上的势均力敌仍然是事件数量和时间处理速度的平衡，在最大时延和平均时延上，QEMU-KVM 虚拟机还是要慢原生虚拟机不少，但是由于 QEMU-KVM 虚拟机采用了某些优化改进降低了需要处理的事件数量，使得其整体的性能损耗并不是很大。

总结:观察上述五份实验结果，我们可以发现 QEMU-KVM 虚拟机性能的一些规律特征。

(1) 对于 CPU 性能，互斥锁性能和多线程性能，虽然 QEMU-KVM 虚拟机由于中间各种转换和调用增加了操作执行的开销，导致各种时延(最大时延，平均时延)普遍低于原生虚拟机，但是 QEMU-KVM 虚拟机可以通过某些优化手段降低执行代码需要的事件数量(我个人猜测是对 guest 执行代码在编译不同体系结构指令时进行优化)，这样就在很大程度上降低了虚拟化所带来损耗的影响，加快了运行时间，尽管优化后 QEMU-KVM 虚拟机的各项执行时间仍然要略逊于原生虚拟机。

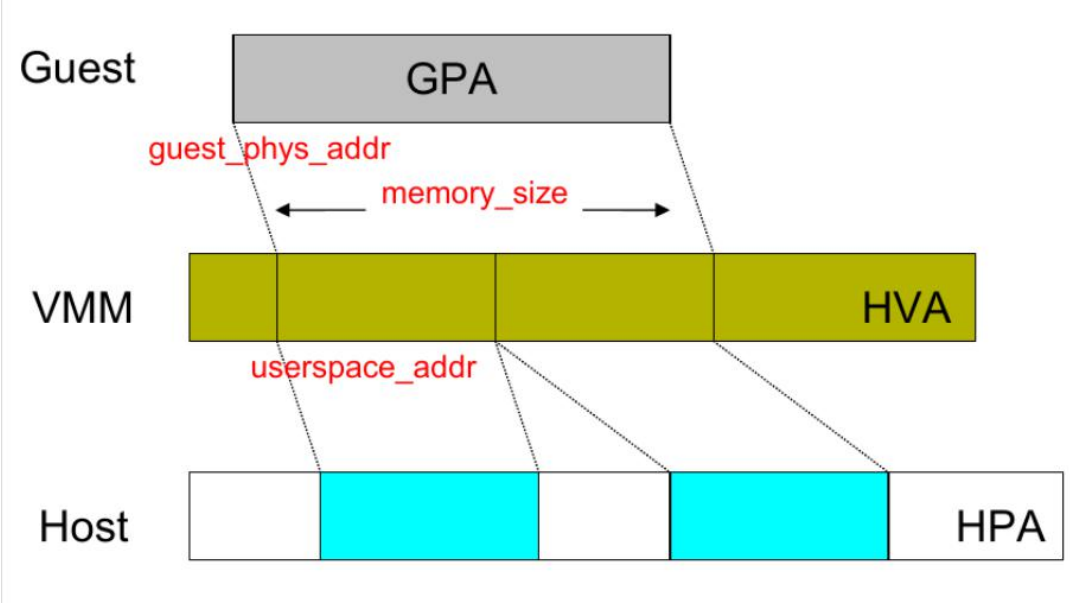
(2)对于内存性能测试和文件 IO 测试，由于涉及到内存的虚拟化和 IO 虚拟化技术，无法直接通过优化代码的方式大幅度降低事件量，因此在基本指令操作开销要更大的情况下，QEMU-KVM 虚拟机难免在总的时间性能上要大幅低于原生的虚拟机。

二. 通过阅读文献等途径，修改开源软件的相关代码，降低虚拟化的性能损耗。

(1) QEMU-KVM 虚拟内存管理简介

QEMU-KVM 的内存虚拟化是由 QEMU 和 KVM 二者共同实现的，其本质上是一个将

Guest 虚拟内存转换成 Host 物理内存的过程。主要有以下步骤
(a)Guest 启动时，由 QEMU 从它的进程地址空间申请内存并分配给 Guest 使用，即内存的申请是在用户空间完成的
(b)通过 KVM 提供的 API，QEMU 将 Guest 内存的地址信息传递并注册到 KVM 中维护，即内存的管理是由内核空间的 KVM 实现的
(c)整个转换过程涉及 GVA,GPA,HVA,HPA 四种地址，Guest 的物理地址空间从 QEMU 的虚拟地址空间中分配
(d)内存虚拟化的关键是在于维护 GPA 到 HVA 的映射关系
具体的示意图可以参考如下



QEMU 在其源代码中定义了几种结构体:
(a)AddressSpace 结构体，表示 Guest 中 CPU/设备看到的内存，是 Guest 中的一段地址空间，x86 体系结构有两种地址空间一种是 address_space_memory，一种是 address_space_io 在 memory.c 中定义。一个 addresspace 一般指向一群具有树结构的 memoryregion,其中 root 指针指向根级 MemoryRegion。
(b)MemoryRegion 结构体，描述 guest memory 中的一段内存区域，作为联系 GPA 和 RAMBlocks(描述真实内存)起到桥梁作用，有 ROM,RAM,MMIO,aliase 等多种类型，其中 aliase 表示 MemoryRegion 中的一段区域。
(c)RAMBlock 结构体，MemoryRegion 结构体描述的是逻辑上一段连续的内存区域，但是 RAMBlock 用来记录实际分配内存地址信息，存储 GPA->HVA 映射关系
QEMU 在用户空间申请内存后，需要将内存信息通过一系列系统调用传入内核空间的 KVM，由 KVM 进行管理，因此 QEMU 也定义了一些用于向 KVM 传递参数的结构体。
(a) KVMSlot, KVM 中内存管理的基本单位，定义在 kvm-all.c 中

```
typedef struct KVMSlot
{
    target_phys_addr_t start_addr; // Guest 物理地址，GPA
    ram_addr_t memory_size;        // 内存大小
    void *ram; // QEMU 用户空间地址，HVA
    int slot;  // Slot 编号
    int flags; // 标志位，例如是否追踪脏页、是否可用等
} KVMSlot;
```

起到的是类似于内存插槽的作用，在 KVMState 的定义中最多支持 32 个 KVMSlot

(b)MemoryListener，用于监控虚拟机的物理地址访问，本质上就是一些回调函数的集合，定义在 memory.c 中。对于每一个 AddressSpace 都有一个 MemoryListener 与其对应，当物理映射 GPA->HVA 改变时，就回调这些函数。

(c)KVMMemoryListener，在 kvm_int.h 中定义，用于将 QEMU 内存拓扑结构动态更新同步至 KVM 中，后续修改就需要用到这个结构体，包含插槽互斥锁，存储监听器，对应 slot，以及编号。

```
typedef struct KVMMemoryListener {
    MemoryListener listener;
    /* Protects the slots and all inside them */
    QemuMutex slots_lock;
    KVMSlot *slots;
    int as_id;
} KVMMemoryListener;
```

QEMU 的一般内存申请流程为:回调函数注册，addresspace 初始化，实际内存的分配。依次调用

configure_accelerator()->kvm_init->kvm_ioctl()->kvm_arch_init()->memory_listener_register()->listener_add_address_space()->kvm_region_add()->kvm_set_user_memory_region()->ioctl(KVM_SET_USER_MEMORY_REGION)

(2) QEMU-KVM 虚拟内存管理 KVM 模块相关源代码修改，修改文件为 qemu-5.1.0/accel/kvm/kvm-all.c 和 qemu-5.1.0/include/sysemu/kvm_int.h

kvm-all.c 文件主要定义了 KVM 虚拟化内存管理的一些函数，其中一些函数需要在访问 slot 之前通过 KVMMemoryListener 中的 slots_lock 上锁。

基本思路(附带一些个人理解): 修改前每一个 KVMMemoryListener 都自带一把锁(slots_lock)各自使用，这种情况下，而当我们想要获取所有 KVMMemoryListener 的所有插槽锁时，必须首先要判断哪些锁已经获取，而哪些没有获取，以避免出现不同 KVMMemoryListener 竞态监听访问同一个 slot 的情况，而这会带来较大的开销。因此我们让所有的 KVMMemoryListener 共用一把锁(定义成全局变量 global_slots_lock())，即同一只能有一个 KVMMemoryListener 访问 slot，保障了安全性，开销也会相应的减小。

以上分析的情况只会在 x86 体系结构中出现，因为 x86 体系结构有两种地址空间:PCI 地址空间和 CPU 地址空间，因此有。而大部分平台都只有一种地址空间(因此只有两种 KVMMemoryListener 一种 KVMMemoryListener)，因此这就意味着对于大部分平台来说，每次访问 slot 的 MemoryListener 都是同一个 MemoryListener，个人感觉此时加锁的必要性不是特别高。

首先修改 qemu-5.1.0/include/sysemu/kvm_int.h，关于 KVMMemoryListener 结构体的定义，删除其中包含的插槽互斥锁，使得插槽锁不再是为每一个 MemoryListener 都分配。

```
typedef struct KVMMemoryListener {
    MemoryListener listener;
    /* Protects the slots and all inside them */
   // QemuMutex slots_lock;
    KVMSlot *slots;
    int as_id;
} KVMMemoryListener;
```

进入 qemu-5.1.0/accel/kvm/kvm-all.c 开头
首先将以下两条宏定义删除

```
#define kvm_slots_lock(kml)      qemu_mutex_lock(&(kml)->slots_lock)
#define kvm_slots_unlock(kml)    qemu_mutex_unlock(&(kml)->slots_lock)
```

将插槽互斥锁定义为所有 KVMMemoryListener 共享的全局变量，这时上锁和解锁的函数参数就不应该再从结构体中获取，而是直接引用全局变量。

```
//#define kvm_slots_lock(kml)      qemu_mutex_lock(&(kml)->slots_lock)
//#define kvm_slots_unlock(kml)    qemu_mutex_unlock(&(kml)->slots_lock)

static QemuMutex global_slots_lock;

#define kvm_slots_lock()      qemu_mutex_lock(&global_slots_lock)
#define kvm_slots_unlock()    qemu_mutex_unlock(&global_slots_lock)
```

接下来寻找所有含有加锁解锁的函数，在对应地方修改成新定义的加锁解锁形式(打注释的部分为删去部分，注释下面的加锁解锁函数即为新增添的)。

```
bool kvm_has_free_slot(MachineState *ms)
{
    KVMState *s = KVM_STATE(ms->accelerator);
    bool result;
    KVMMemoryListener *kml = &s->memory_listener;

    //kvm_slots_lock(kml);
    kvm_slots_lock();
    result = !!kvm_get_free_slot(kml);
    //kvm_slots_unlock(kml);
    kvm_slots_unlock();

    return result;
}
```

```
int kvm_physical_memory_addr_from_host(KVMState *s, void *ram,
                                       hwaddr *phys_addr)
{
    KVMMemoryListener *kml = &s->memory_listener;
    int i, ret = 0;

    //kvm_slots_lock(kml);
    kvm_slots_lock();
    for (i = 0; i < s->nr_slots; i++) {
        KVMSlot *mem = &kml->slots[i];

        if (ram >= mem->ram && ram < mem->ram + mem->memory_size) {
            *phys_addr = mem->start_addr + (ram - mem->ram);
            ret = 1;
            break;
        }
    }
    //kvm_slots_unlock(kml);
    kvm_slots_unlock();
    return ret;
}
```

```
static int kvm_section_update_flags(KVMMemoryListener *kml,
                                    MemoryRegionSection *section)
{
    hwaddr start_addr, size, slot_size;
    KVMSlot *mem;
    int ret = 0;

    size = kvm_align_section(section, &start_addr);
    if (!size) {
        return 0;
    }

    //kvm_slots_lock(kml);
    kvm_slots_lock();
    while (size && !ret) {
        slot_size = MIN(kvm_max_slot_size, size);
        mem = kvm_lookup_matching_slot(kml, start_addr, slot_size);
        if (!mem) {
            /* We don't have a slot if we want to trap every access. */
            goto out;
        }

        ret = kvm_slot_update_flags(kml, mem, section->mr);
        start_addr += slot_size;
        size -= slot_size;
    }

out:
    //kvm_slots_unlock(kml);
    kvm_slots_unlock();
    return ret;
}
```

```
if (!size) {
    /* Nothing more we can do... */
    return ret;
}

//kvm_slots_lock(kml);
kvm_slots_lock();
for (i = 0; i < s->nr_slots; i++) {
    mem = &kml->slots[i];
    /* Discard slots that are empty or do not overlap the section */
    if (!mem->memory_size ||
        mem->start_addr > start + size - 1 ||
        start > mem->start_addr + mem->memory_size - 1) {
        continue;
    }

    if (start >= mem->start_addr) {
        /* The slot starts before section or is aligned to it.  */
        offset = start - mem->start_addr;
        count = MIN(mem->memory_size - offset, size);
    } else {
        /* The slot starts after section.  */
        offset = 0;
        count = MIN(mem->memory_size, size - (mem->start_addr - start));
    }
    ret = kvm_log_clear_one_slot(mem, kml->as_id, offset, count);
    if (ret < 0) {
        break;
    }
}

//kvm_slots_unlock(kml);
kvm_slots_unlock();
return ret;
```

```
    /* use aligned delta to align the ram address */
    ram = memory_region_get_ram_ptr(mr) + section->offset_within_region +
          (start_addr - section->offset_within_address_space);

    //kvm_slots_lock(kml);
    kvm_slots_lock();
    if (!add) {
        do {
            slot_size = MIN(kvm_max_slot_size, size);
            mem = kvm_lookup_matching_slot(kml, start_addr, slot_size);
            if (!mem) {
                goto out;
            }
            if (mem->flags & KVM_MEM_LOG_DIRTY_PAGES) {
                kvm_physical_sync_dirty_bitmap(kml, section);
            }
```

```
        err = kvm_set_user_memory_region(kml, mem, true);
        if (err) {
            fprintf(stderr, "%s: error registering slot: %s\n", __func__,
                    strerror(-err));
            abort();
        }
        start_addr += slot_size;
        ram += slot_size;
        size -= slot_size;
    } while (size);

out:
    //kvm_slots_unlock(kml);
    kvm_slots_unlock();
}
```

```
static void kvm_log_sync(MemoryListener *listener,
                         MemoryRegionSection *section)
{
    KVMMemoryListener *kml = container_of(listener, KVMMemoryListener, listener);
    int r;

    //kvm_slots_lock(kml);
    kvm_slots_lock();
    r = kvm_physical_sync_dirty_bitmap(kml, section);
    //kvm_slots_unlock(kml);
    kvm_slots_unlock();
    if (r < 0) {
        abort();
    }
}
```

在 kvm_memory_register 函数中，由于该函数在 QEMU 最先将内存分配信息注册进入 KVM 模块时调用，因此需要进行锁的初始化操作，此处仅仅是将结构体的属性传递直接变为全局变量的引用传递。

```
void kvm_memory_listener_register(KVMState *s, KVMMemoryListener *kml,
                                  AddressSpace *as, int as_id)
{
    int i;

    //qemu_mutex_init(&kml->slots_lock);
    qemu_mutex_init(&global_slots_lock);
    kml->slots = g_malloc0(s->nr_slots * sizeof(KVMSlot));
    kml->as_id = as_id;

    for (i = 0; i < s->nr_slots; i++) {
        kml->slots[i].slot = i;
    }

    kml->listener.region_add = kvm_region_add;
    kml->listener.region_del = kvm_region_del;
    kml->listener.log_start = kvm_log_start;
    kml->listener.log_stop = kvm_log_stop;
    kml->listener.log_sync = kvm_log_sync;
    kml->listener.log_clear = kvm_log_clear;
    kml->listener.priority = 10;

                                                    1351,5        41%
```

三. 修改后的 QEMU-KVM 虚拟机性能测试

安装完成后，使用 sysbench 的测试指令进行测试，此处为了更好地比较我们仍采用相同指令。

(1)CPU 性能对比

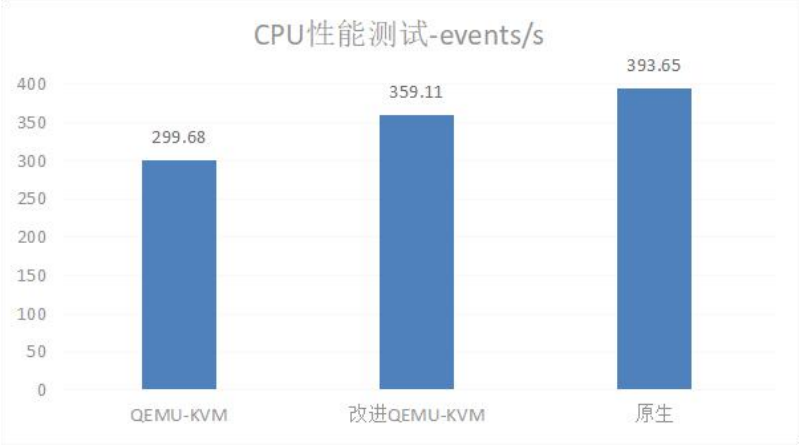
最大质数发生器测试(最大质数不超过 20000，线程数量为 1)

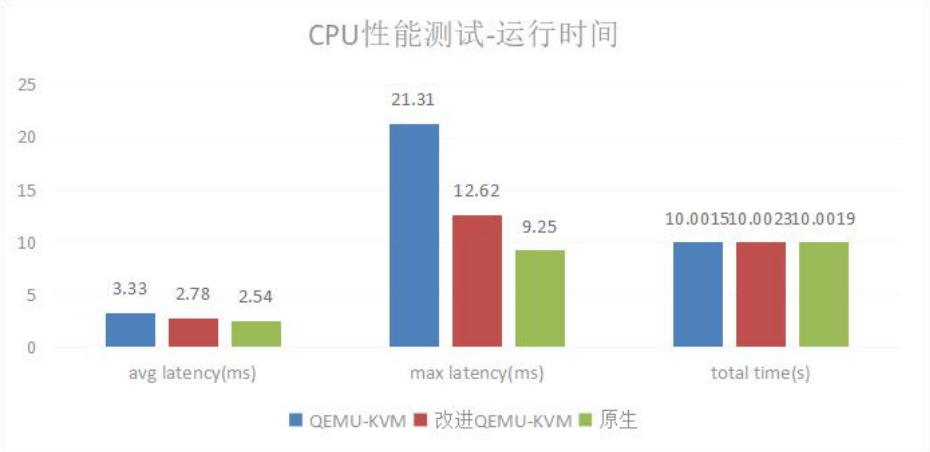
```
root@ubuntu:/qemu-5.1.0# sysbench cpu --cpu-max-prime=20000 run
```

QEMU-KVM 虚拟机得到的执行结果如下

```
Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:   359.11

General statistics:
    total time:                          10.0023s
    total number of events:              3593

Latency (ms):
         min:                                  2.25
         avg:                                  2.78
         max:                                 12.62
         95th percentile:                      3.68
         sum:                               9990.25

Threads fairness:
    events (avg/stddev):           3593.0000/0.00
    execution time (avg/stddev):   9.9902/0.00
```

关于 CPU 性能，原生虚拟机，修改前 QEMU-KVM 虚拟机，修改后 QEMU-KVM 虚拟机的性能比较如下图所示。





观察上述两个图表可以看出，修改后的 QEMU-KVM 在性能上要优于修改前不少，每秒执行的事件数量虽然仍然劣于原生虚拟机的 393.65/s，但已经从修改前的 299.68/s 上升至 359.11/s，同时在平均时延和最大时延上，修改后的 QEMU-KVM

已经大幅降低，但是总的执行时间基本变化幅度不大。总的来说，修改后的 QEMU-KVM 在 CPU 性能方面的虚拟化损耗还是大大降低了。

(2) 内存性能对比

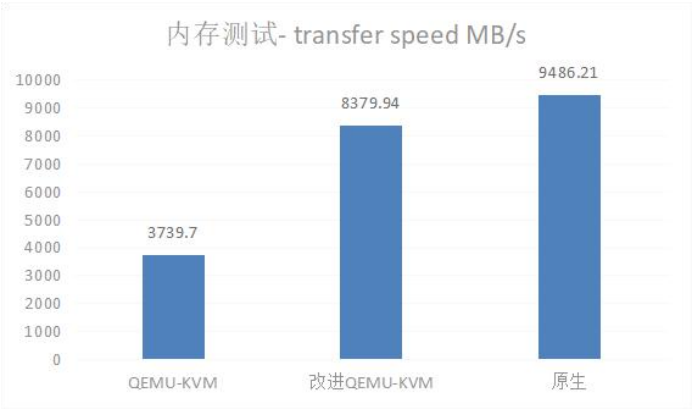
向内存中传输 2G 的数据，每个块的大小为 8K。

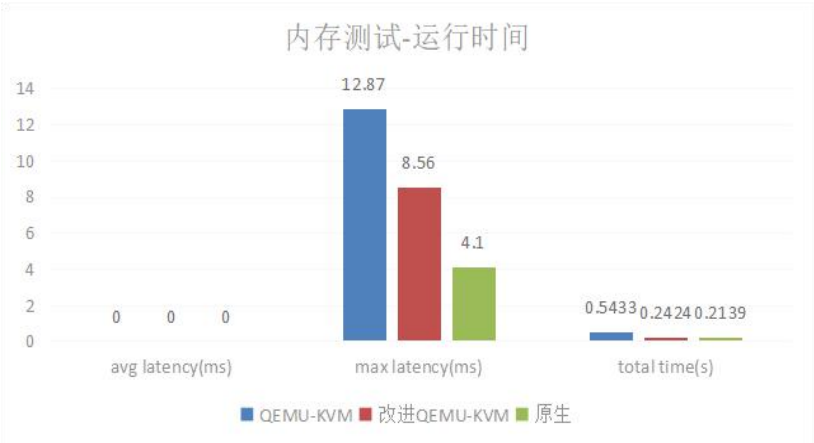
```
root@ubuntu:/qemu-5.1.0# sysbench memory --memory-block-size=8k --memory-total-size=2G run
```

QEMU-KVM 虚拟机得到的执行结果如下

```
Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 8KiB
  total size: 2048MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 262144 (1072632.32 per second)

2048.00 MiB transferred (8379.94 MiB/sec)


General statistics:
    total time:                          0.2424s
    total number of events:              262144

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                    8.56
         95th percentile:                        0.00
         sum:                                  193.58

Threads fairness:
    events (avg/stddev):           262144.0000/0.00
    execution time (avg/stddev):   0.1936/0.00
```

关于内存性能，原生虚拟机，修改前 QEMU-KVM 虚拟机，修改后 QEMU-KVM 虚拟机的性能比较如下图所示。





观察可以发现，由于内存操作耗时量较低，修改后的 QEMU-KVM 在平均时延上

体现的效果不是很明显，但是在最大时延上改进后的 QEMU-KVM 和改进前相比有了显著的下降，传输速度有了显著地提升，均正在接近原生水平，二者所带来的的效果就是总的执行时间上改进后的 QEMU-KVM 已经很接近于原生虚拟机，大大降低了虚拟机损耗。

(3) 文件 IO 性能对比

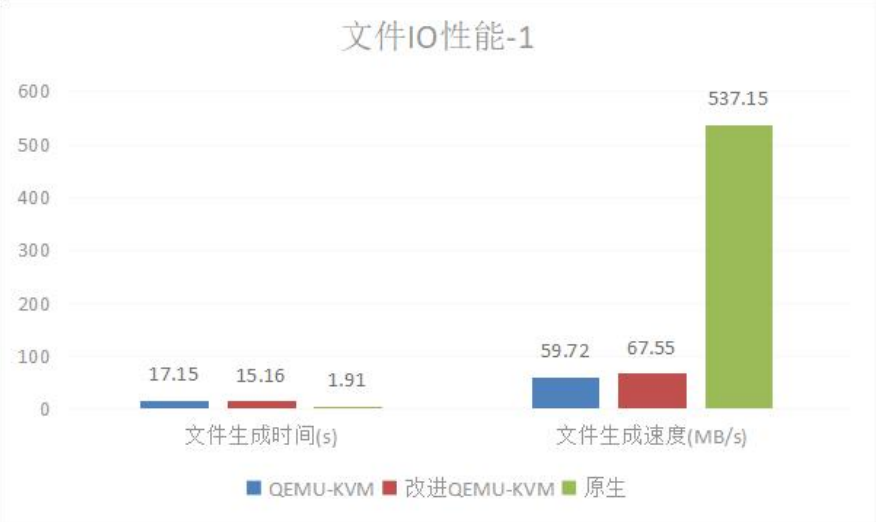
1.准备阶段:首先生成需要测试的文件，生成的小文件都存放在当前目录下

```
root@ubuntu:/qemu-5.1.0# sysbench fileio --threads=2 --file-total-size=1G --file-test-mode=rndrw prepare
```

QEMU-KVM 虚拟机得到的执行结果如下

```
Creating file test_file.119
Creating file test_file.120
Creating file test_file.121
Creating file test_file.122
Creating file test_file.123
Creating file test_file.124
Creating file test_file.125
Creating file test_file.126
Creating file test_file.127
1073741824 bytes written in 15.16 seconds (67.55 MiB/sec).
```

关于文件生成性能，原生虚拟机，修改前 QEMU-KVM 虚拟机，修改后 QEMU-KVM 虚拟机的性能比较如下图所示。



观察实验结果可以发现，修改后的 QEMU-KVM 的文件生成速度仍然要远远低于原生虚拟机，文件生成时间也要比虚拟机要高很多，但是和修改前相比已经有了一些下降。这可能主要是因为文件生成需要较多的磁盘 io 操作，仅仅修改虚拟内存管理模块的互斥锁并不能从根本上消除虚拟化的损耗，但是虚拟化损耗还是有较大的降低。

2.运行阶段(测试随机读写大小总共 1G 的文件，线程数量为 2)

```
root@ubuntu:/qemu-5.1.0# sysbench fileio --threads=2 --file-total-size=1G --file-test-mode=rndrw run
```

QEMU-KVM 虚拟机得到的执行结果如下

```
Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Extra file open flags: 0
128 files, 8MiB each
1GiB total file size                                    ts.
Block size 16KiB
Number of IO requests: 0
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync() each 100 requests.
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random r/w test
Initializing worker threads...

Threads started!


File operations:
    reads/s:                      497.80
    writes/s:                     331.86
    fsyncs/s:                     1051.67

Throughput:
    read, MiB/s:                  7.78    0.0015s
    written, MiB/s: events:       5.19

General statistics:
    total time:                          10.0015s
    total number of events:              18821

Latency (ms):
         min:                                    0.00
         avg:                                    1.06
         max:                                   25.45
         95th percentile:                        3.68
```

```
Threads fairness:
    events (avg/stddev):           9410.5000/75.50
    execution time (avg/stddev):   9.9806/0.00
```

关于文件随机读写性能，原生虚拟机，修改前 QEMU-KVM 虚拟机，修改后 QEMU-KVM 虚拟机的性能比较如下图所示。

观察实验结果可以发现，修改后的 QEMU-KVM 的度写吞吐量，执行总时间和文件生成速度方面的性能仍然要远远差于原生虚拟机，但是和修改前相比已经有了一些略微的改善。这可能主要还是因为上一页分析中所讲述的原因。总的来说，虚拟化损耗还是有所下降。

3.运行测试完毕后需要将临时文件清理回收

```
root@ubuntu:/qemu-5.1.0# sysbench fileio --threads=2 --file-total-size=1G --file-test-mode=rndrw cleanup
sysbench 1.0.11 (using system LuaJIT 2.1.0-beta3)

Removing test files...
```

(4) 互斥锁性能对比

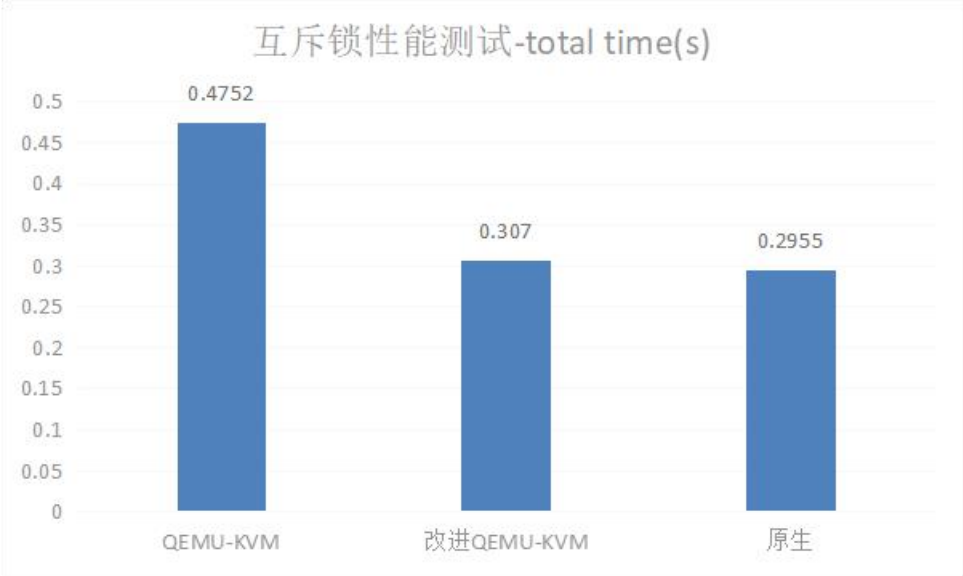
线程数量为 2，互斥数组的总大小 4096，每个线程互斥锁的数量为 50000，互斥锁外部的空循环数量为 10000，模拟所有线程在同一时刻并发运行。

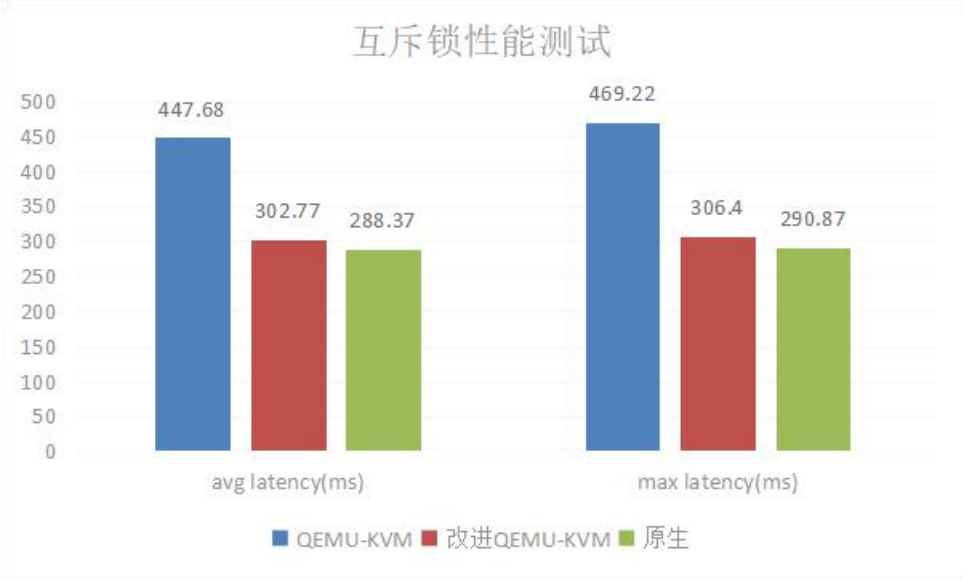
```
root@ubuntu:/qemu-5.1.0# sysbench mutex --threads=2 --mutex-num=4096 --mutex-locks=50000 --mutex-loops=10000 run
```

QEMU-KVM 虚拟机得到的执行结果如下

```
Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Initializing worker threads...

Threads started!


General statistics:
    total time:                          0.3070s
    total number of events:              2

Latency (ms):
         min:                                  299.14
         avg:                                  302.77
         max:                                  306.40
         95th percentile:                      308.84
         sum:                                  605.55

Threads fairness:
    events (avg/stddev):           1.0000/0.00
    execution time (avg/stddev):   0.3028/0.00
```

关于互斥锁性能，原生虚拟机，修改前 QEMU-KVM 虚拟机，修改后 QEMU-KVM 虚拟机的性能比较如下图所示。





改进后的虚拟机的性能开销下降明显，在平均时延，最大时延和执行总时间上，改进后的 QEMU-KVM 虚拟机均大幅降低，同时正在接近原生水平。这些说明我们达到了预期的一些效果。

(6) POSIX 线程性能对比

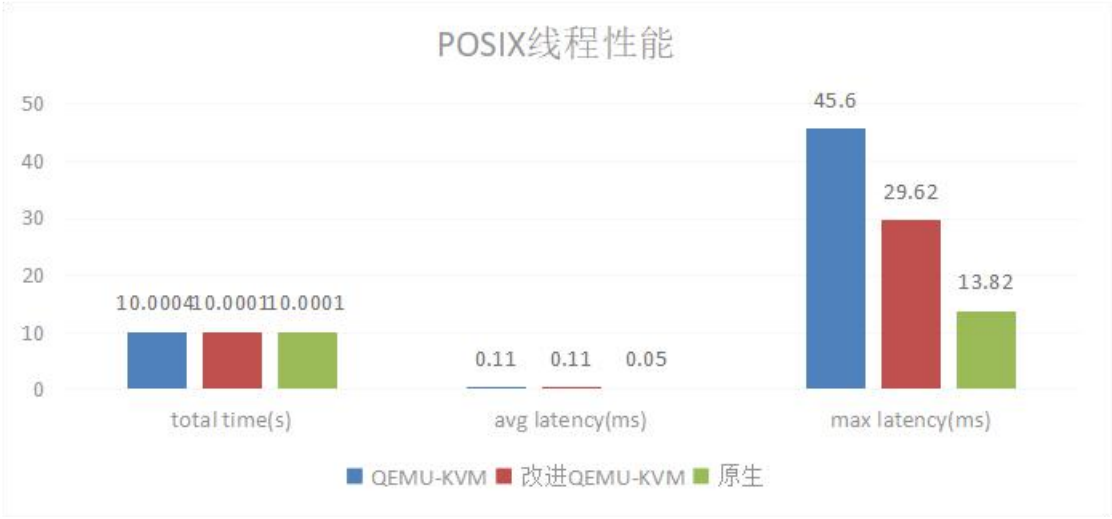
线程数量为 2，每个请求产生 100 个线程，每个线程拥有锁的数量为 4

```
root@ubuntu:/qemu-5.1.0# sysbench threads --threads=2 --thread-yields=100 --thread-locks=4 run
```

QEMU-KVM 虚拟机得到的执行结果如下

```
Running the test with following options:
Number of threads: 2
Initializing random number generator from current time


Initializing worker threads...

Threads started!


General statistics:
    total time:                          10.0001s
    total number of events:              178115

Latency (ms):
         min:                                    0.08
         avg:                                    0.11
         max:                                   29.62
         95th percentile:                        0.17
         sum:                                19899.49

Threads fairness:
    events (avg/stddev):           89057.5000/1347.50
    execution time (avg/stddev):   9.9497/0.00
```

关于多线程性能，原生虚拟机，修改前 QEMU-KVM 虚拟机，修改后 QEMU-KVM 虚拟机的性能比较如下图所示。



观察实验结果可以看到，由于不涉及到较为复杂的 IO 操作，QEMU 对 guest 代码进行优化后降低事件数量，以减少总的执行时间，使得本身 QEMU-KVM 在多线程方面的虚拟化损耗就不大，因此修改后的 QEMU-KVM 在总的执行时间和平均时延上效果不是很明显，但还是在最大时延上取得了较大的改进。

综上所述，我们的工作在降低虚拟化损耗上取得了较为明显的进展。

四. 实验感悟

本次实验极大的增进了我对 QEMU-KVM 内存虚拟化管理模块的了解。一开始对于内容一我感到一头雾水，后来通过上网搜索资料和比对老旧版本的 QEMU 和新版本 QEMU 的一些区别找到了一些灵感，当然薛春宇同学的优秀作业也给了我一定的启发，让我能够从某一功能模块的互斥锁这一比较小的角度去切入。感谢助教在我实验过程中问题的悉心解答和管海兵老师对我们的一些指导性意见。当然，我感觉自己对虚拟内存管理这一块知识和互斥锁的使用仅仅是了解了凤毛麟角，包括写在实验报告中的许多也仅仅是个人的理解，可能难免会有一些错误，希望后续能有机会继续深入学习。