鲲鹏Hyper Tuner性能分析工具实验指导

实验目标与基本要求

本实验基于鲲鹏云服务器部署并熟悉性能分析工具Hyper Tuner。通过此次实验，能够掌握：
① 使用鲲鹏性能分析工具Hyper Tuner创建系统性能分析以及函数分析任务
② 使用鲲鹏的NEON指令来提升矩阵乘法执行效率

实验步骤

1. 登录服务器

输入下列命令，登录服务器：

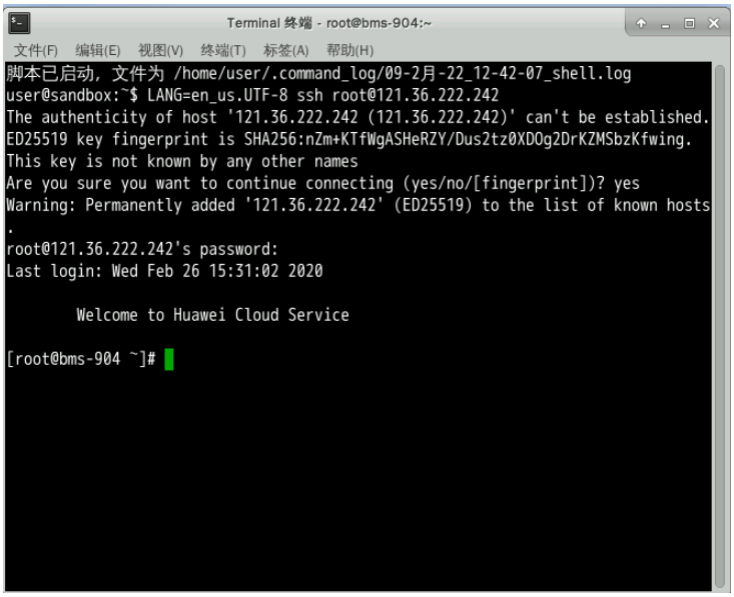
```
LANG=en_us.UTF-8 ssh root@EIP
```

【注意】需将EIP换成分配的服务器公网IP地址；敲回车后：

输入yes；

输入密码：（输入密码时，命令行窗口不会显示密码，输完之后直接回车）；

登录成功后如下图所示：



2 工具安装

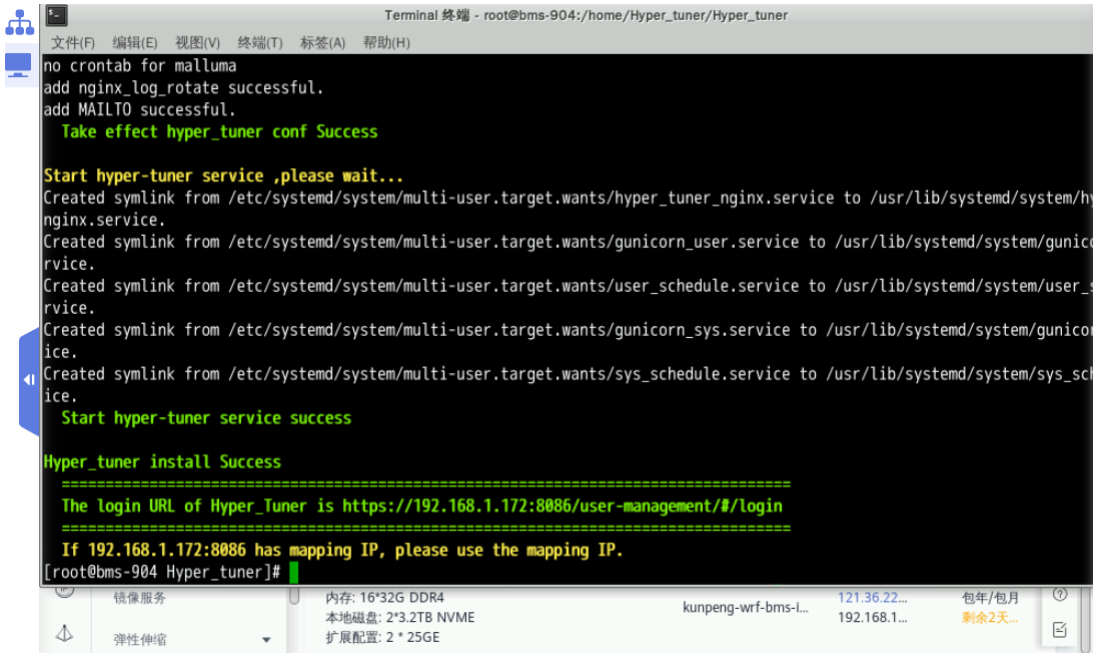
2.1 安装依赖工具

```
#需要依次输入下列命令来安装依赖工具并激活
#1. 设置SSH超时断开时间，防止服务器断连
sed -i '112a ClientAliveInterval 600\nClientAliveCountMax 10'
/etc/ssh/sshd_config && systemctl restart sshd
#2. 安装centos-release-scl
yum install centos-release-scl -y
#3. 安装devtoolset
yum install devtoolset-7-gcc* -y
#4. 激活对应的devtoolset
scl enable devtoolset-7 bash
#5. 安装jdk 11版本并在home目录下重命名为jdk文件夹【需等待约3分钟】
cd /home && wget
https://mirrors.huaweicloud.com/kunpeng/archive/compiler/bisheng_jdk/bisheng-jdk-
11.0.9-linux-aarch64.tar.gz && tar -zxvf bisheng-jdk-11.0.9-linux-aarch64.tar.gz
&& mv bisheng-jdk-11.0.9 jdk
```

2.2 安装Hyper Tuner

```
#需要依次输入下列命令来安装工具
#1. 下载软件包“Hyper-Tuner-2.2.T3.tar.gz”安装在“/home”的根目录下，并解压安装包
cd /home && wget
https://mirrors.huaweicloud.com/kunpeng/archive/Tuning_kit/Packages/Hyper-Tuner-
2.2.T3.tar.gz && tar -zxvf Hyper-Tuner-2.2.T3.tar.gz
#2. 安装系统性能优化工具，默认安装在“/opt”目录下
cd /home/Hyper_tuner && tar -zxvf Hyper-Tuner-2.2.T3.tar.gz && cd
/home/Hyper_tuner/Hyper_tuner && ./hyper_tuner_install.sh -a -i -ip=SIP -
jh=/home/jdk
#[注意] 将上述命令中的SIP替换为私有IP地址
```

中途出现输入提示，**输入 Y**，稍等片刻安装后，运行结果如下图所示：

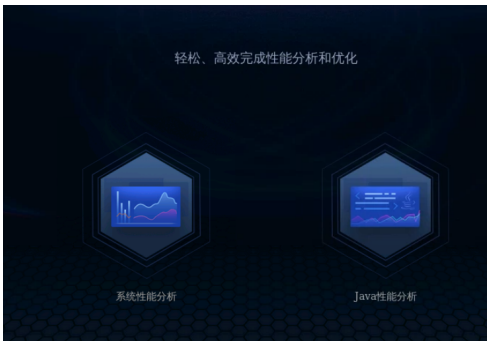


2.3 登录Hyper Tuner

1. 打开火狐浏览器新建标签页，访问地址 https://弹性公网IP:8086/user-management/#/login（例如：https://121.36.222.242:8086/user-management/#/login），如果显示连接不安全，选择高级，确认添加为例外即可显示正常网页；如下图所示：

2. 首次登录需要创建管理员密码，默认用户名为 **tunadmin**，密码为 **tunadmin12#$**

3. 登录成功后，单击“系统性能分析”，进入系统性能分析操作界面，仔细阅读免责声明后，勾选阅读并同意声明内容，即可进行下一步操作，如下图所示：



3. 一维矩阵运算热点函数检测优化

此实验进行矩阵乘法计算，首先测试用普通for loop方式进行一维矩阵乘法计算的性能，然后考虑到矩阵乘法可以拆分并行计算，且并行计算分支相对独立，于是使用鲲鹏的NEON指令来进行计算，检测优化后的执行效率。

3.1 编译运行“矩阵内存访问”代码

步骤1 下载代码

在终端中执行以下命令，创建/home/testdemo目录，进入/home/testdemo目录并下载需要测试的程序：

```
mkdir /opt/testdemo && cd /opt/testdemo && wget https://sandbox-experiment-resource-north-4.obs.cn-north-4.myhuaweicloud.com/hot-spot-function/multiply.tar.gz && tar -zxvf multiply.tar.gz
```

步骤2 修改程序

修改multiply.c和multiply_simd.c中N的定义，修改为130000000：

修改前：

```
#define N    1000000000
#define SEED    0x1234
```

修改后：

```
#define N    130000000
#define SEED    0x1234
```

步骤3 编译程序

执行如下命令，进入multiply文件夹，编译multiply.c并赋予执行文件所有用户只读、只写、可执行权限。

```
cd /opt/testdemo/multiply && gcc -g multiply.c -o multiply && chmod -R 777
/opt/testdemo/multiply
```

```
[root@bms-5447 testdemo]# cd /opt/testdemo/multiply && gcc -g multiply.c -o mul
iply && chmod -R 777 /opt/testdemo/multiply
[root@bms-5447 multiply]#
```

步骤4 执行程序

执行如下命令，将multiply测试程序绑定CPU核启动（当前程序绑核到CPU 1，循环运行multiply程序200次），使用后台启动脚本，程序运行的输出（标准输出（1））将会保存到multiply.out文件，错误信息（2）会重定向到multiply.out文件：

```
cd /opt/testdemo/multiply && nohup bash multiply_start.sh >>multiply.out 2>&1 &
```

```
[root@bms-5447 multiply]# cd /opt/testdemo/multiply && nohup bash multiply_start
.sh >>multiply.out 2>&1 &
[1] 29202
[root@bms-5447 multiply]#
```

执行命令：jobs -l，如果当前进程是running状态，如下图所示，则说明程序在运行状态，可以进行下面的步骤。

```
jobs -l
```

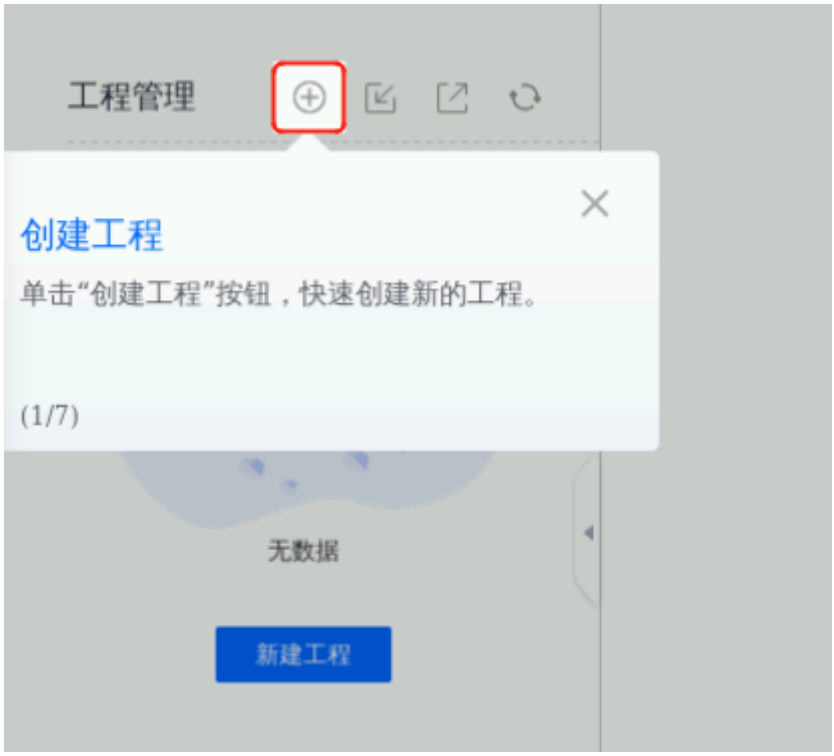
```
[root@bms-8045 multiply]# jobs -l
[1]+ 22109 Running                 cd /opt/testdemo/multiply && nohup bash multi
ply_start.sh >> multiply.out 2>&1 &
```

注：执行以下3.2、3.3的性能采集时，需要在此multiply程序运行过程中采集，若multiply程序运行结束，3.2和3.3操作将无法采集，此时可以重新执行cd /opt/testdemo/multiply && nohup bash multiply_start.sh >>multiply.out 2>&1 & 运行multiply程序，再执行3.2和3.3章节的性能采集。

3.2 系统性能全景分析

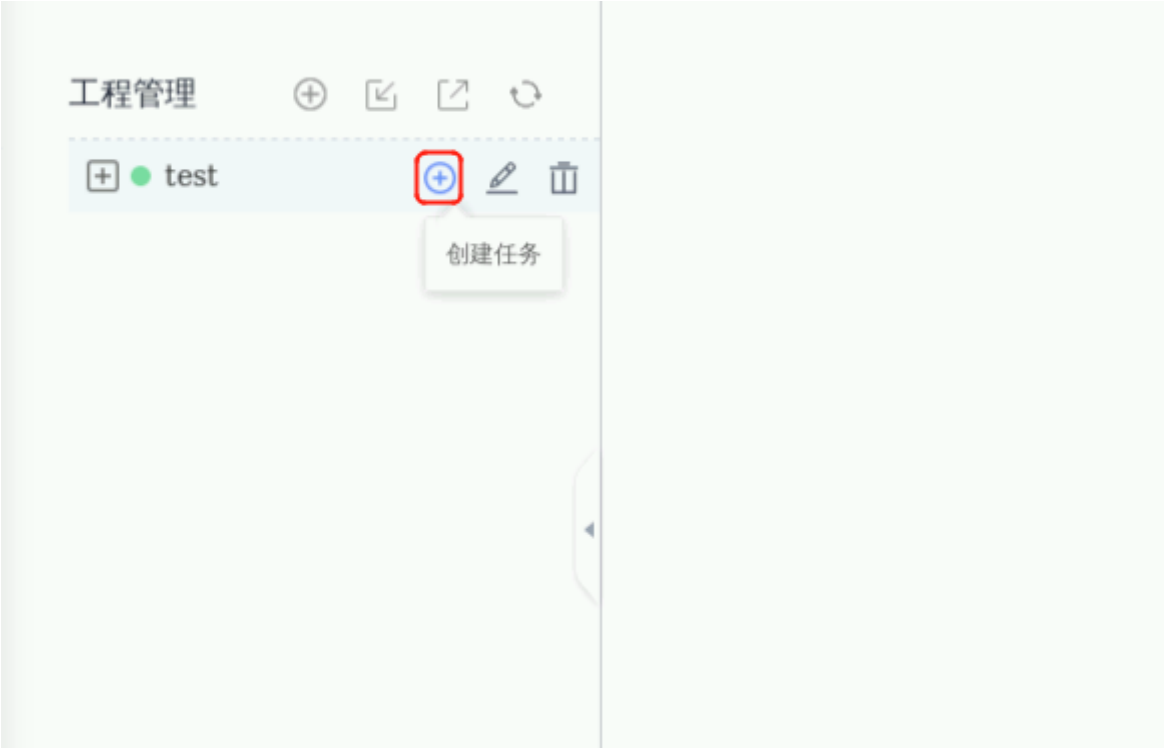
步骤1 创建工程

在程序运行的过程当中，我们利用工具分析当前程序。回到浏览器“系统性能分析”操作界面，根据提示，单击工程管理旁边“+”按钮创建工程，输入工程名称，如test，勾选节点，当前为此云服务器节点，单击“确认”创建工程，如下图所示：

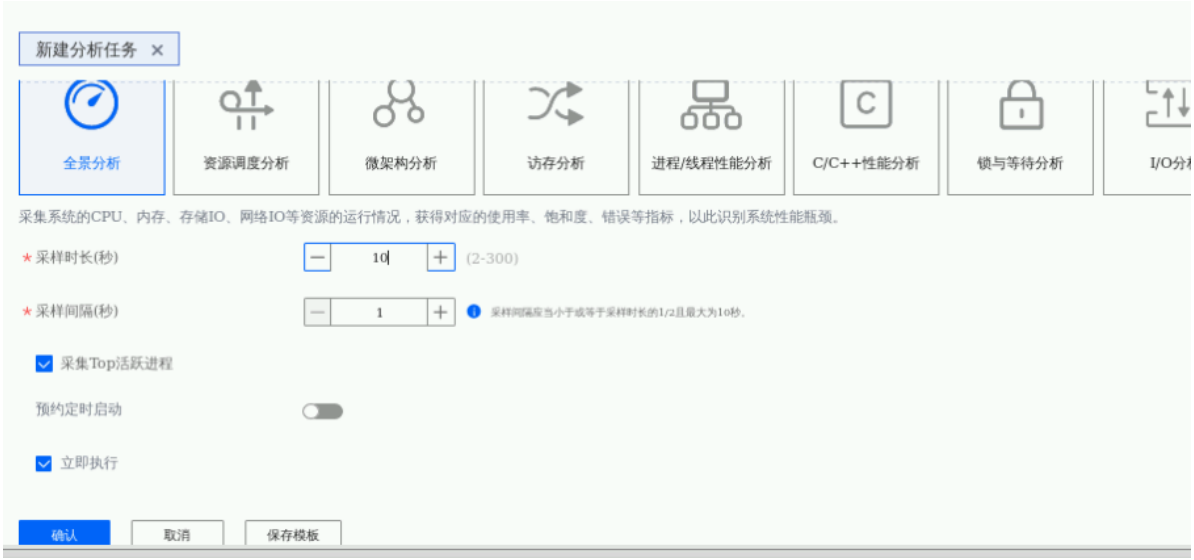


步骤2 创建任务

将光标放到创建好的工程“test”上，会出现按钮，单击“+”创建任务，如下图所示：



弹出如下界面，填写如下：
①任务名称：自定义工程名称multiply_quanjing，
②选择分析对象：系统；
③分析类型：全景分析；
④采样时长：10秒；
⑤采样间隔：1秒，
点击“确认”执行



步骤3 查看采集分析结果

执行完毕后，显示全景分析的结果，如下图所示。
①总览，点击“检测到CPU利用率高”显示优化建议；
②性能，在CPU利用率的图表中，可以看到top5的CPU核在采集时间内的利用率变化，点击右上角的按钮，可以看到在采集时间内的各项数据的平均值。

由此可见，当前CPU核1的使用率（“性能”页签下%CPU的数值）接近100%，并且绝大部分消耗在用户态。说明该程序全部消耗在用户态计算，没有其他IO或中断操作。

3.3 进程/线程性能分析

步骤1 创建任务

参照2.3.2步骤2创建进程/线程性能分析任务，如下图所示。
①任务名称：multiply_process
②分析对象：系统
③分析类型：进程/线程性能分析
④采样时长：10秒
⑤采样间隔：1秒，
⑥采样类型：全部勾选
⑦采集线程信息：打开
点击“确认”执行。

步骤2 查看采集分析结果

执行完毕后，显示进程/线程性能分析的结果，如下图所示。

PID/TID	%user	%system	%wait	%CPU	Command
PID 1	0.09	0.45	0	0.55	systemd
TID 1	0.18	0.45	0	0.64	|__systemd
PID 11	0	0.09	0	0.09	rcu_sched
PID 183	0	0	0.09	0	ksoftirqd/34
PID 208	0	0	0.09	0	ksoftirqd/39
PID 228	0	0	0.09	0	ksoftirqd/43
PID 263	0	0	0.09	0	ksoftirqd/50
PID 268	0	0	0.09	0	ksoftirqd/51
PID 273	0	0	0.64	0	ksoftirqd/52
PID 298	0	0.09	0.18	0.09	ksoftirqd/57
PID 313	0	0	0.35	0	ksoftirqd/60
PID 575	0	0	0.09	0	ksoftirqd/112
PID 584	0	0.09	0	0.09	migration/114
PID 585	0	0	0.09	0	ksoftirqd/114
PID 672	0	0.94	0	0.94	khugepaged
PID 1318	0	0.09	0	0.09	jbd2/dm-0-8
PID 1439	0	0.09	0	0.09	systemd-journal
PID 3967	0.55	0.09	0	0.64	dbus-daemon

默认排序是按照PID/TID升序排列，以此观察哪个进程造成了CPU消耗，点击%CPU右侧降序排列按钮，得到如下图。可以看到multiply程序在消耗大量的CPU，同时全部消耗在用户态中，由此我们可以推测很可能是自身代码实现算法差的问题。

PID/TID	%user	%system	%wait	%CPU	Command
PID 27254	97.67	1.57	0	98.96	multiply
PID 27301	5.62	22.83	0	28.45	pidstat
PID 27292	5.86	18.41	0	24.27	pidstat
PID 27302	4.33	19.21	0	23.54	pidstat
PID 27303	6.06	15.12	0	21.18	pidstat
PID 27295	0	4.02	0	4.02	sadc
PID 8843	0.31	2.11	0	2.43	irqbalance

3.4 程序性能分析

步骤1 编译程序

```
cd /opt/testdemo/multiply && gcc -g -O2 -o multiply multiply.c && chmod -R 777
/opt/testdemo/multiply
```

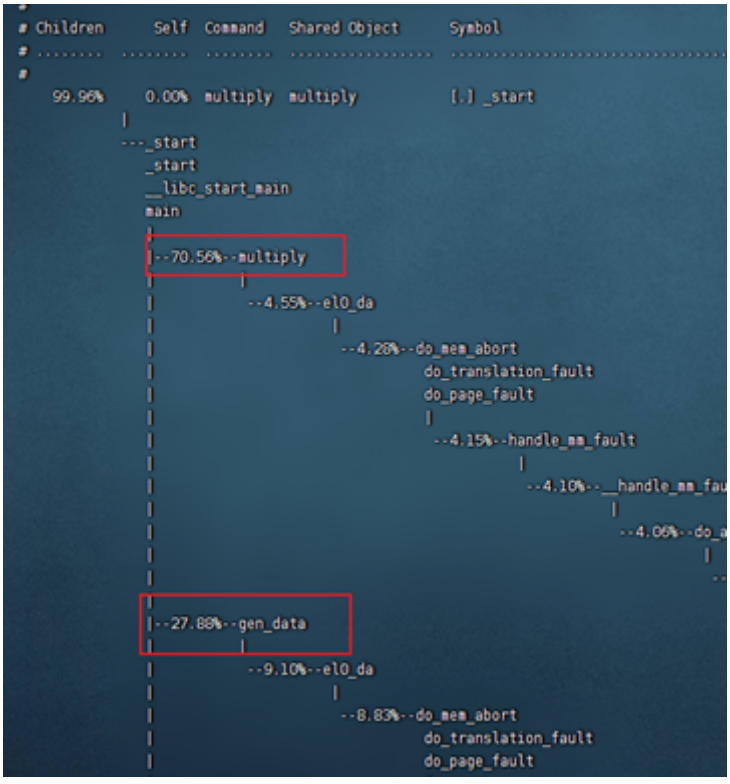
步骤2 查看multiply.c程序中乘法函数multiply消耗时间

```
cd /opt/testdemo/multiply/ && ./multiply
```

```
[root@ecs-f230 test_wnc]# ./multiply
217156.000000, 217156.000000, 217156.000000, 217156.000000
Execution time = 359.610 ms
```

步骤3 采集热点函数占用率

```
#1. 启动perf record 采集数据
cd /opt/testdemo/multiply/ && sudo perf record --call-graph dwarf ./multiply -d 1
-b
#2. 解析perf.data的内容
cd /opt/testdemo/multiply/ && sudo perf report -i perf.data > perf_multiply.txt
#3. 查看main函数和子函数的CPU平均占用率
cd /opt/testdemo/multiply/ && less perf_multiply.txt
```



可以看到 multiply函数占用70%的CPU，首先考虑优化multiply函数。

3.5 NEON指令优化代码

考虑到矩阵乘法可以拆分并行计算，且并行计算分支相对独立，可以使用鲲鹏的NEON指令来提升执行效率。NEON指令通过将对单个数据的操作扩展为对寄存器，也即同一类型元素矢量的操作，从而大大减少了操作次数，以此来提升执行效率。

3.6 优化后程序性能分析

步骤1 编译程序

```
cd /opt/testdemo/multiply &&  gcc -g -O2 -o multiply_simd multiply_simd.c &&
chmod -R 777 /opt/testdemo/multiply
```

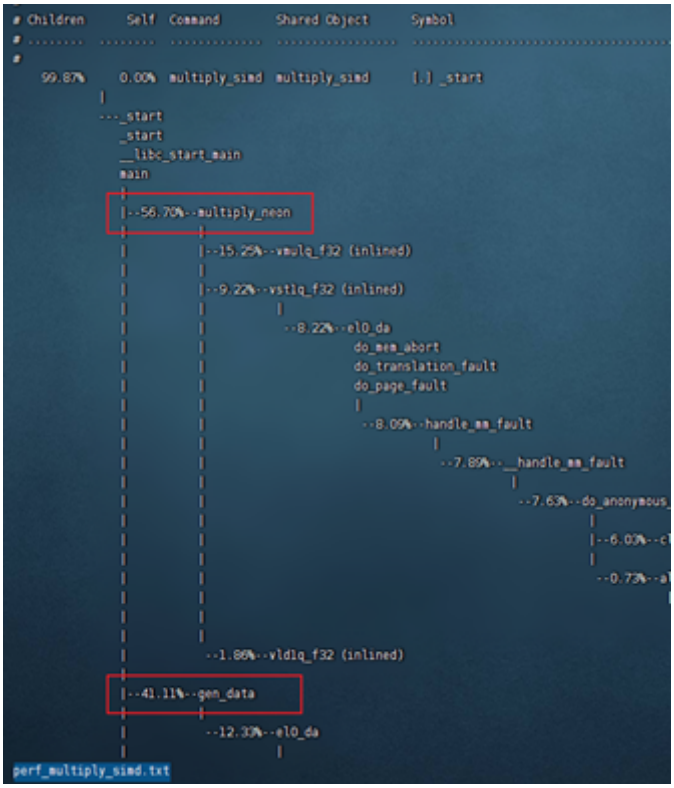
步骤2 查看multiply_simd.c程序中乘法函数multiply_neon消耗时间

```
cd /opt/testdemo/multiply/ && ./multiply_simd
```

```
[root@ecs-f230 test_wnc]# ./multiply_simd
217156.000000, 217156.000000, 217156.000000, 217156.000000
Execution time = 186.679 ms
```

步骤3 采集热点函数占用率

```
#1. 启动perf record 采集数据
cd /opt/testdemo/multiply/ && sudo perf record --call-graph dwarf ./multiply_simd
-d 1 -b
#2. 解析perf.data的内容
cd /opt/testdemo/multiply/ && sudo perf report -i perf.data >
perf_multiply_simd.txt
#3.查看main函数和子函数的CPU平均占用率
cd /opt/testdemo/multiply/ && less perf_multiply_simd.txt
```



对比优化前的multiply函数和优化后的multiply_simd函数，我们可以发现无论实在运行时间还是CPU占用率上，multiply_simd函数都有明显的下降，说明优化有效。

3.7 结束程序

在终端中执行如下代码：
①通过jobs -l查看multiply_simd程序运行pid（如果程序尚未结束，可以看到后台运行的程序；如果没有输出，说明进程已经结束，不需要进行此步操作）。
②通过kill -9杀死进程。替换<multiply_simd程序pid>为刚查到的multiply_simd的pid。

```
jobs -l
kill -9 <multiply_simd程序pid>
```

4.实验总结

通过将矩阵乘法计算方式从for loop方式优化为使用鲲鹏的NEON指令来进行计算，函数指令数大幅减少，执行效率得到了提升。至此实验已全部完成。