

计算机科学与技术学院

## 《计算机系统结构》课程实验

学 号：

班 级：

专 业：计算机科学与技术

学生姓名：

2025 年 月 日

# 实 验 报 告 1

学生姓名：	学 号：	时间：地点：
实验课程名称：计算机系统结构		
一、实验名称：流水线中的相关—顺序查找数组中某一数据 x		
<p>二、实验原理：</p> <p>1、WinDLX 平台与流水线</p> <p>WinDLX 模拟器是一个图形化、交互式的 DLX 流水线模拟器，它采取伪汇编形式编码，模拟流水线的工作方式，能够演示 DLX 流水线是如何工作的。流水线的指令执行分为 5 个阶段：取指、译码、执行、访存、写回。</p> <p>WinDLX 模拟器还提供了对流水线操作的统计功能，便于对流水线进行性能分析。</p> <p>2、流水线中的相关及解决办法</p> <p>（1）结构相关：当某一条机器指令需要访问物理器件时，该器件可能正在被占用，例如连续的两条加法指令都需要用到浮点加法器，就产生结构相关，可以通过增加加法器的方式解决结构相关；</p> <p>（2）数据相关：当某一条指令需要访问某个寄存器时，此时这个寄存器正被另一条指令所使用，从而产生数据相关，可以通过重定向技术解决数据相关；</p> <p>（3）控制相关：当程序执行到某个循环语句时，顺序执行的下一条语句将被跳继续执行循环体的内容，从而产生控制相关，可以通过循环展开解决控制相关。</p>		
<p>三、实验目的：</p> <p>1、加深对流水线理论知识的理解；</p> <p>2、掌握对流水线性能分析的方法，了解影响流水线效率的因素；</p> <p>3、熟悉在 WinDLX 体系结构下的汇编代码编写和优化；</p> <p>4、了解相关的类型及各类相关的解决办法；</p> <p>5、培养运用所学知识解决实际问题的能力。</p>		

#### 四、实验内容：

- 1、根据 WinDLX 模拟器伪汇编指令规则编写顺序查找数组中数字 x 的程序 Findx1.s 和 input.s;
- 2、分别按照不同顺序将 Findx1.s 和 input.s 装入主存，分析输入顺序不同对运行结果产生的影响;
- 3、观察程序中出现的的数据、控制、结构相关，指出程序中出现上述现象的指令组合，并提出解决相关的办法;
- 4、分别考察各类解决的相关办法，分析解决相关后性能的变化。

注意：除解决结构相关，其他情况下加、乘、除运算器都只有一个。

本问题中所有浮点延迟部件设置为：加法：2 个延迟周期；乘法：5 个延迟周期；除法：19 个延迟周期。

#### 五、实验器材（设备、元器件）：

电脑一台

VMware Workstation

虚拟机（Windows7 32 位操作系统）

WinDLX 模拟器

#### 六、实验步骤及操作：

##### 1、初始化 WinDLX 模拟器

(1)为 WinDLX 创建目录，C:\Users \Desktop\WinDLX。将 WinDLX 和 Findx1.s、Findx2.s、input.s 放在这个目录中。

(2)初始化 WinDLX 模拟器：点击 File 菜单中的 Reset all 菜单项，弹出一个“Reset DLX”对话框，点击窗口中的“确定”按钮即可。如图 1 所示。

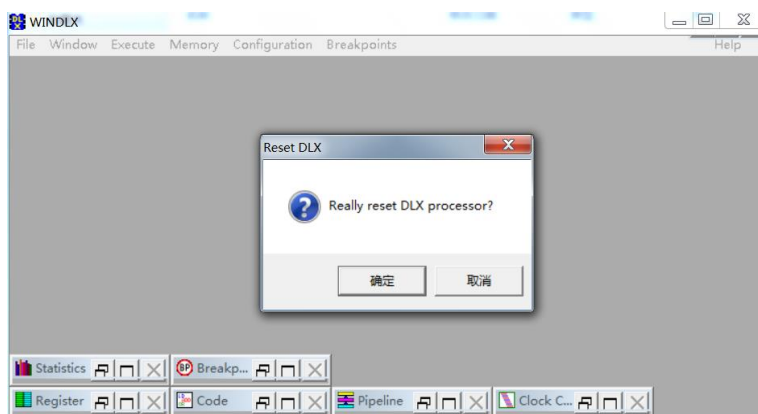


图 1 初始化模拟器界面

## 2、将程序装入 WinDLX 平台

点击 File 菜单中的 Load Code or Data 菜单项，依次双击 Findx1.s 和 input.s。点击 load，将两个程序装入。如图 2 所示。

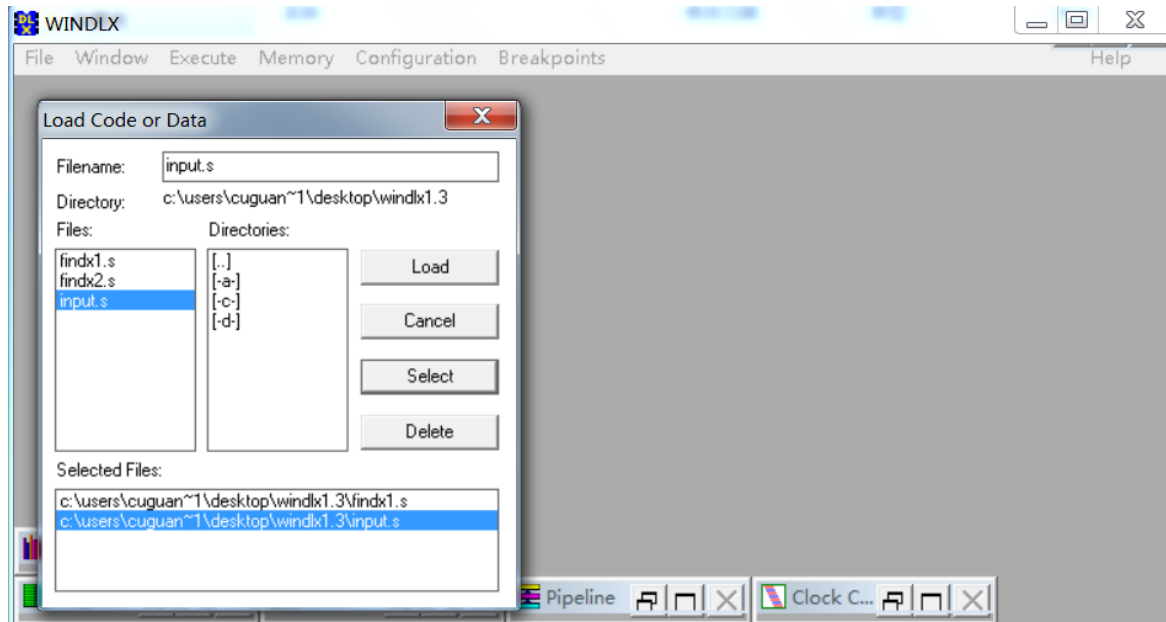


图 2 程序装入界面

## 3、运行程序并观察

进行单步调试，在 WinDLX 模拟器的 6 个子窗口观察程序的执行情况。观察程序运行的总时钟周期，产生的相关种类以及每种相关的数量。

## 4、解决数据相关

勾选 Enable Forwarding，采用重定向技术添加专用数据通路减少数据相关，观察数据相关的数量变化。

## 5、解决结构相关

将 Addition Units 的数目由 1 增加到 2，观察结构相关的数量变化。

## 6、解决控制相关

将 Findx1.s 的循环体展开形成新文件 Findx2.s，采用循环展开的方法减少控制相关，观察控制相关的数量变化。

## 七、实验数据及结果分析：

### 1、程序装入顺序对运行结果的影响

先装入 Findx1.s 再装入 input.s 时，程序能够正确执行；当先装入 input.s 再装入 Findx1.s 时，因为 input.s 的地址高，而程序顺序执行到 input.s 时无法正确的输出，不会出现结果。

### 2、主要代码及说明

```
.data
Prompt:      .asciiz  "input An integer which is array's size value >1 : "
Prompt2:     .asciiz  "input An integer which you want to find  : "
PromptF:     .asciiz  "Not Found  "
PromptLast:  .asciiz  "input an integer : "
PrintfFormat: .asciiz  "The position of the integar is : %d "
.align      2
PrintfPar1:  .word    PromptF
PrintfPar:   .word    PrintfFormat
Printf:      .space   4
PrintfValue: .pace    1024

.text
.global main
main:
    ;输入数组长度 n
    addi    r1,r0,Prompt      ;将 Prompt 字符串首地址放入 r1 寄存器中
    jal     InputUnsigned     ;调用 input 子函数读取一个正整数
    add     r2,r0,r1          ;将 input 函数读取到的数组长度 r1 存放在 r2 和 r7 中
    add     r7,r0,r1
    ;输入要查找的数 x
    addi    r1,r0,Prompt2     ;将 Prompt2 字符串首地址放入 r1 寄存器中
    jal     InputUnsigned     ;调用 input 子函数读取一个正整数
    add     r9,r0,r1          ;将 input 函数读取到的数 x 存放在 r9 中
    addi    r3,r0,0           ;r3 寄存器中数清 0，指向数组中的起始位置，用来遍历
    addi    r10,r0,1          ;立即数 1 写入 r10，表示要找的数 x 在数组中的下标

    ;循环调用 input 函数读数，共读取 n 次
InputArray:
    beqz    r2,ProcessPart    ;判断当前循环是否继续，为 0 时跳出输入循环
    addi    r1,r0,PromptLast  ;将 PromptLast 字符串首地址放入 r1 寄存器中
    jal     InputUnsigned     ;调用 input 子函数读取一个正整数
    sw      PrintfValue(r3),r1 ;将 r1 寄存器中的数保存到 r3 寄存器指向的地址中
    addi    r3,r3,4           ;r3 往后移 4 位,表示指向数组中的下一个位置
    subi    r2,r2,1           ;每次输入成功后 r2 寄存器中的数减 1
    j       InputArray        ;无条件跳转向 InputArray 标识的指令地址
```

ProcessPart:

addi r3,r0,0 ;调用 ProcessPart 函数将 r3 寄存器中数清 0

;在数组中查找数 x

FindLoop:

beqz r7,ENDT ;r7 中所存数为 0 则跳转向 ENDT 所标识的指令地址  
lw r20,PrintfValue(r3) ;将 r3 寄存器所指向地址中所存的数送入 r20 寄存器中  
addi r3,r3,4 ;r3 往后移 4 位,表示指向数组中的下一个位置  
sub r20,r9,r20 ;将 r9 和 r20 的差送入 r20  
beqz r20,END ;r20 中所存数为 0 则跳转向 END 所标识的指令地址  
subi r7,r7,1 ;每次查找后 r7 寄存器中的数减 1  
addi r10,r10,1 ;每次查找后 r10 寄存器中的数加 1  
j FindLoop ;无条件跳转向 FindLoop 标识的指令地址

;当查找完数组仍未找到数 x 时, 进行查找失败的提示

ENDT:

addi r1,r0,PromptF ;将 PromptF 字符串首地址放入 r1 寄存器中  
sw PrintfPar,r1 ;将 r1 寄存器中的数放入 PrintfPar 中  
addi r14,r0,PrintfPar ;输出 PrintfPa 中的内容  
trap 5 ;调用中断, 格式化为标准输出  
j over ;无条件跳转向 over 标识的指令地址

;当在数组中查找到数 x 时, 进行查找成功的提示, 并输出数 x 在数组中的下标

END:

movi2fp f10,r10 ;拷贝 r10 寄存器中一个字长的数放到 f10 寄存器中  
movi2fp f12,r0 ;拷贝 r0 寄存器中一个字长的数放到 f12 寄存器中  
movi2fp f1,r0 ;拷贝 r0 寄存器中一个字长的数放到 f1 寄存器中  
movi2fp f2,r0 ;拷贝 r0 寄存器中一个字长的数放到 f2 寄存器中  
addf f2,f10,f1 ;将 f10 和 f1 的和送入 f2  
addf f10,f1,f12 ;将 f1 和 f12 的和送入 f10  
addf f12,f1,f2 ;将 f1 和 f2 的和送入 f12  
movfp2i r10,f12 ;拷贝 f12 寄存器中一个字长的数放到 r10 寄存器中  
sw Printf,r10 ;将 r10 寄存器中的数放入 Printf 中  
addi r14,r0,PrintfPa ;输出 PrintfPa 中的内容  
trap 5 ;调用中断, 格式化为标准输出  
j over ;无条件跳转向 over 标识的指令地址

over:

trap 0 ;调用系统中断, 0 表示程序执行结束

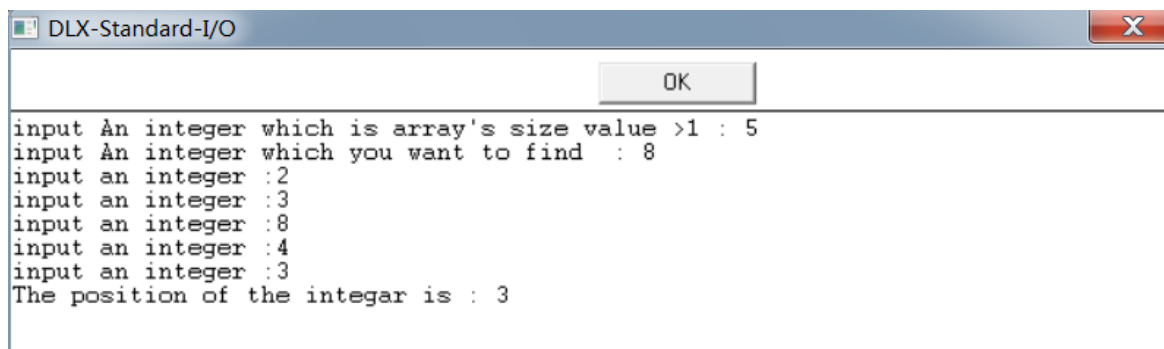
### 3、程序分析及运行结果

#### (1) 测试样例 1:

①按照提示, 先输入一个正整数 n 表示数组的长度。测试用例: n=5

②接下来再按照提示, 输入一个正整数 x 表示所要查找的数字。测试用例: x=8

③最后按照提示，依次输入 n 个整数.测试样例：2，3，8，4，3



```
DLX-Standard-I/O
input An integer which is array's size value >1 : 5
input An integer which you want to find : 8
input an integer :2
input an integer :3
input an integer :8
input an integer :4
input an integer :3
The position of the integer is : 3
```

图 3 运行结果截图 1

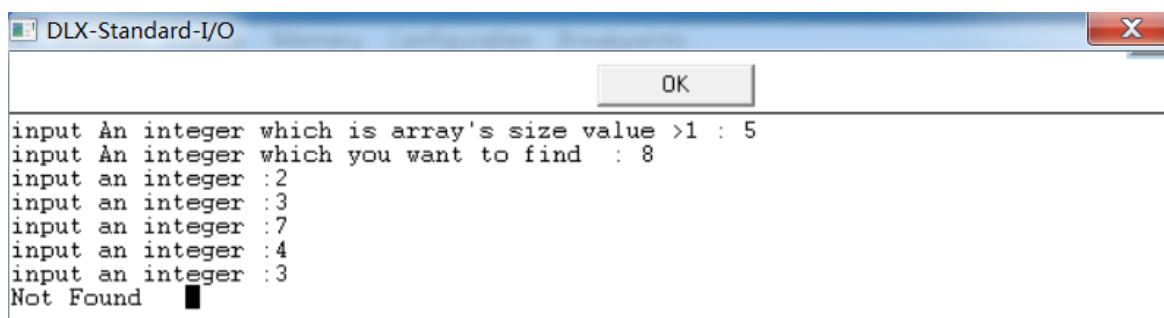
由图 3 知，程序首先提示了数 x 已经被找到，并给出了数 x 在数组中的位置 3，表明程序运行结果正确。

(2) 测试样例 2:

①按照提示，先输入一个正整数 n 表示数组的长度。测试用例：n=5

②接下来再按照提示，输入一个正整数 x 表示所要查找的数字。测试用例：x=8

③最后按照提示，依次输入 n 个整数.测试样例：2，3，7，4，3



```
DLX-Standard-I/O
input An integer which is array's size value >1 : 5
input An integer which you want to find : 8
input an integer :2
input an integer :3
input an integer :7
input an integer :4
input an integer :3
Not Found
```

图 4 运行结果截图 2

由图 4 知，程序提示了并未在数组中找到数 x，容易观察到数组中并无整数 8，表明程序运行结果正确。

(3) 点击 Statistics 窗口，查看程序执行的时钟周期以及数据相关、结构相关、控制相关的发生次数。

如图 5 所示，程序执行共用 478 个时钟周期，数据相关发生 116 次，结构相关发生 1 次，控制相关发生 38 次。

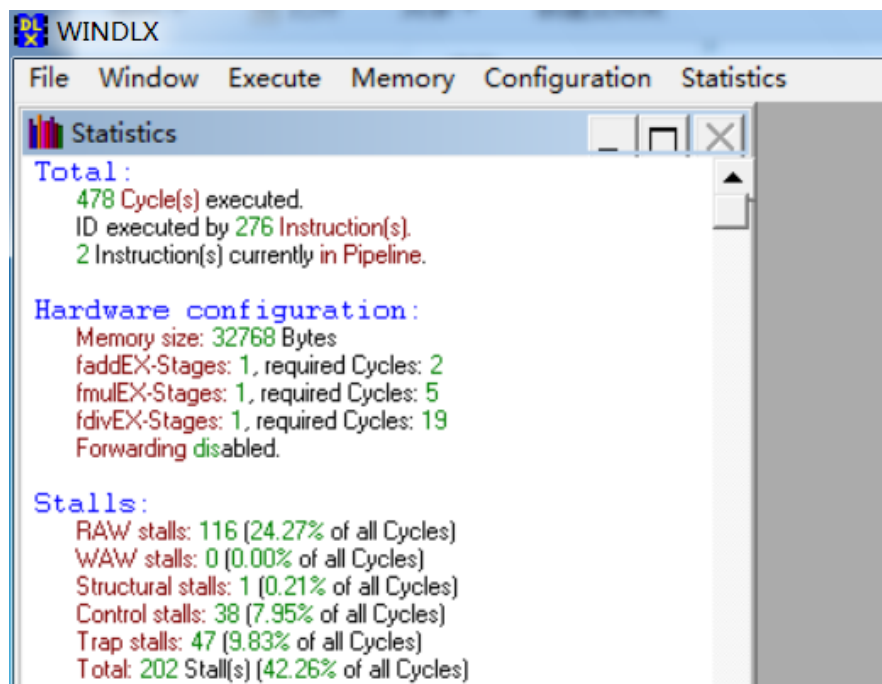


图 5 时钟周期和相关数据截图

#### 4、数据相关及解决

##### (1) 数据相关产生的原因

```

addf      f12,f1,f2      ;将 f1 和 f2 的和送入 f12
movfp2i   r10,f12        ;拷贝 f12 寄存器中一个字长的数放到 r10 寄存器中
sw        Printf,r10      ;将 r10 寄存器中的数放入 Printf 中

```

movfp2i 指令要使用 f12 寄存器的数据，但是上一条指令 addf 刚刚执行完数据还没有更新，产生数据相关，并且之后执行的 sw 指令也需要上一条 movfp2i 指令的中还没更新的寄存器 r10，产生数据相关，如图 6 所示。

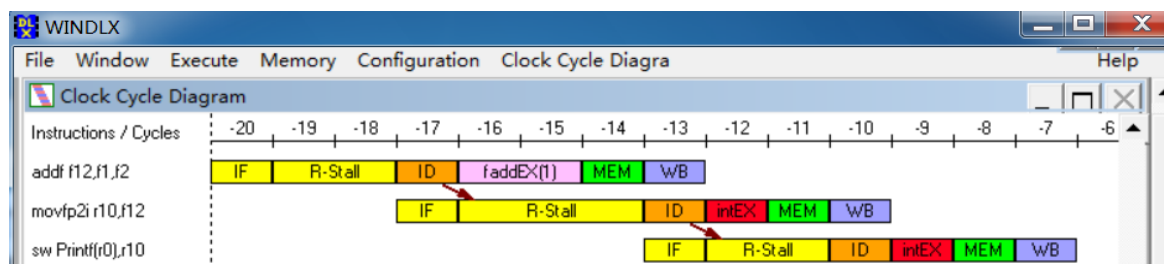


图 6 数据相关截图

##### (2) 数据相关的解决

采用重定向技术，勾选 Configuration 的 Enable Forwarding 选项。在第一条指令结束后直接将寄存器 f12 和寄存器 r10 的内容更新，消除数据相关，如图 7 所示。



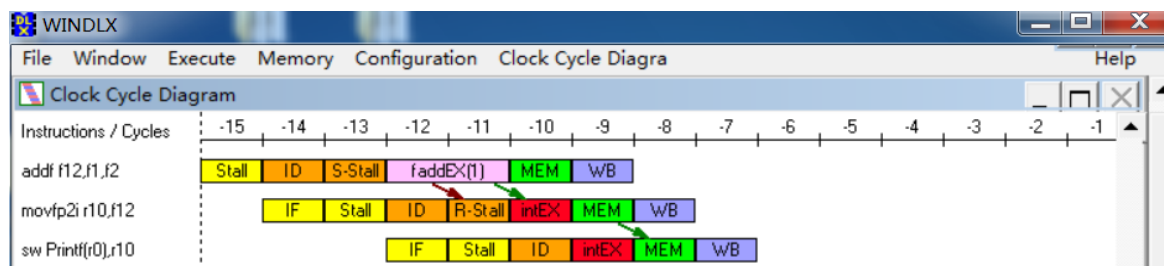


图 7 解决数据相关截图

查看运行结果，数据相关数量降低，数据相关个数为 53，如图 8 所示。

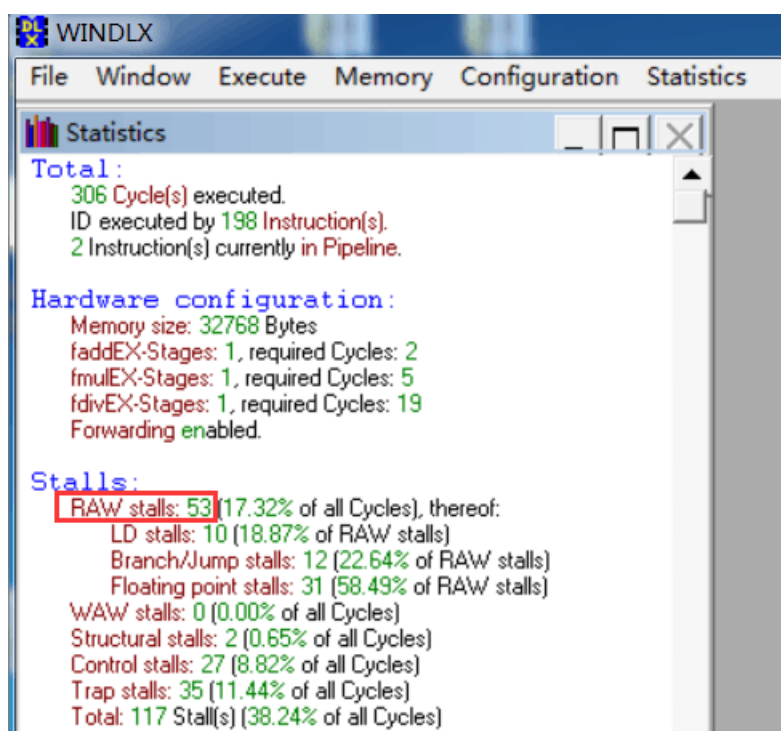


图 8 解决数据相关的数据截图

## 5、控制相关及解决

### (1) 控制相关的产生原因

InputArray:

beqz	r2, ProcessPart	;判断当前循环是否继续，为 0 时跳出输入循环
addi	r1,r0,PromptLast	;将 PromptLast 字符串首地址放入 r1 寄存器中
jal	InputUnsigned	;调用 input 子函数读取一个正整数
sw	PrintfValue(r3),r1	;将 r1 寄存器中的数保存到 r3 寄存器指向的地址中
addi	r3,r3,4	;r3 往后移 4 位,表示指向数组中的下一个位置
subi	r2,r2,1	;每次输入成功后 r2 寄存器中的数减 1
j	InputArray	;无条件跳转向 InputArray 标识的指令地址

在这段程序中，循环体的出现造成了控制相关。

### (2) 控制相关的解决

采用复制循环体内容的方式将循环体的内容展开。可以降低控制相关的个数。如下：

InputArray:

beqz	r2, ProcessPart	;判断当前循环是否继续, 为 0 时跳出输入循环
addi	r1,r0,PromptLast	;将 PromptLast 字符串首地址放入 r1 寄存器中
jal	InputUnsigned	;调用 input 子函数读取一个正整数
sw	PrintfValue(r3),r1	;将 r1 寄存器中的数保存到 r3 寄存器指向的地址中
addi	r3,r3,4	;r3 往后移 4 位,表示指向数组中的下一个位置
subi	r2,r2,1	;每次输入成功后 r2 寄存器中的数减 1
beqz	r2, ProcessPart	;判断当前循环是否继续, 为 0 时跳出输入循环
addi	r1,r0,PromptLast	;将 PromptLast 字符串首地址放入 r1 寄存器中
jal	InputUnsigned	;调用 input 子函数读取一个正整数
sw	PrintfValue(r3),r1	;将 r1 寄存器中的数保存到 r3 寄存器指向的地址中
addi	r3,r3,4	;r3 往后移 4 位,表示指向数组中的下一个位置
subi	r2,r2,1	;每次输入成功后 r2 寄存器中的数减 1
j	InputArray	;无条件跳转向 InputArray 标识的指令地址

重新运行后控制相关数量减少为 35，如图 9 所示。

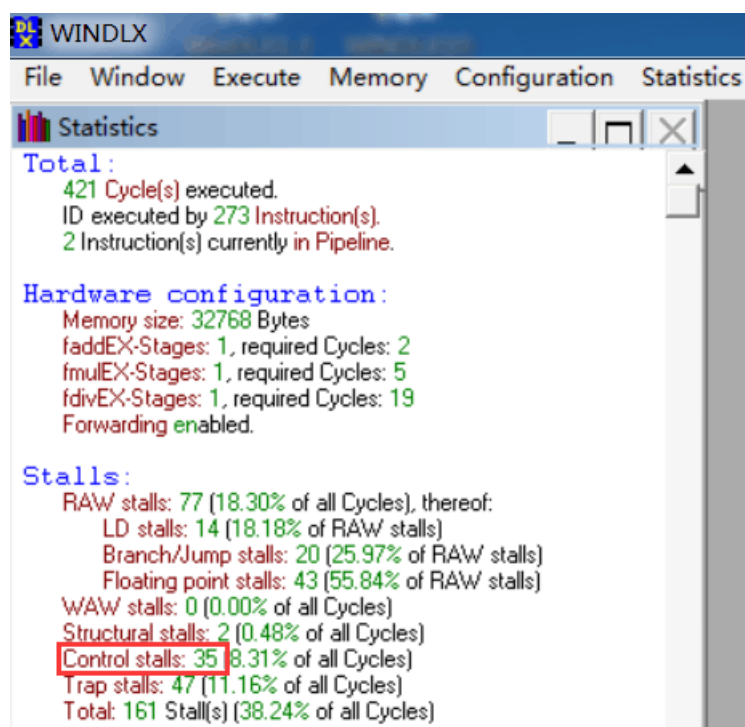


图 9 解决控制相关的数据截图

## 6、结构相关及解决

### (1) 结构相关产生的原因

addf	f2,f10,f1	;把寄存器 f10 和 f1 中的和送入到 f2 中
addf	f10,f1,f12	;把寄存器 f1 和 f12 中的和送入到 f10 中
addf	f12,f1,f2	;把寄存器 f1 和 f2 中的和送入到 f12 中

在这段语句运行时需要连续进行加操作，由于加法器只有一个，产生结构相关。

## (2) 结构相关的解决

添加加法器 Addition Units 的个数，如图 10 所示。

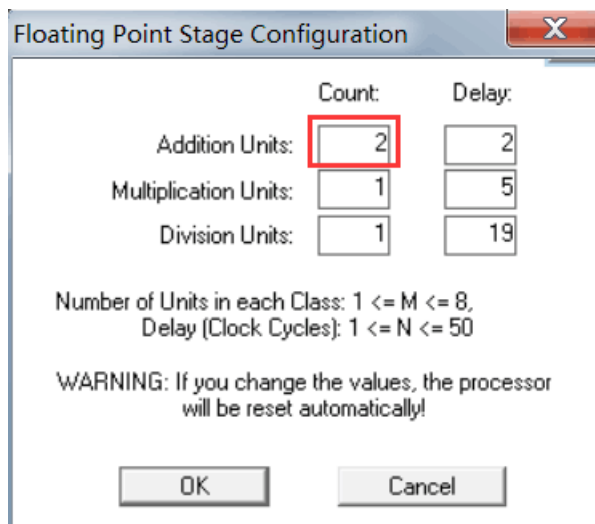


图 10 增加乘法器界面的截图

再次运行程序可以发现结构相关数量降低，降低到 0 个，如图 11 所示。

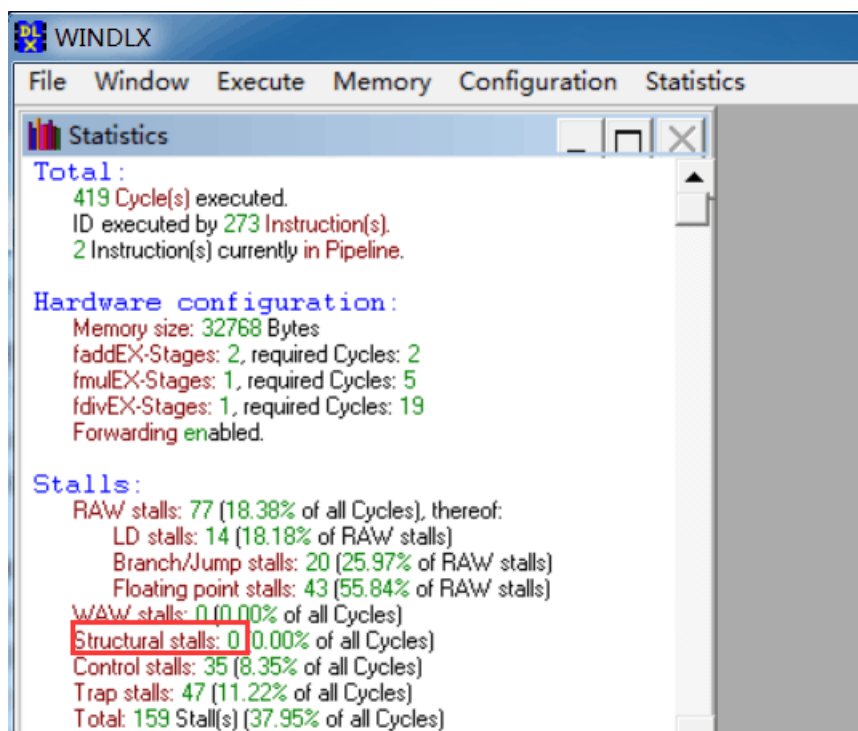


图 11 解决结构相关的数据截图

## 7、程序流程图

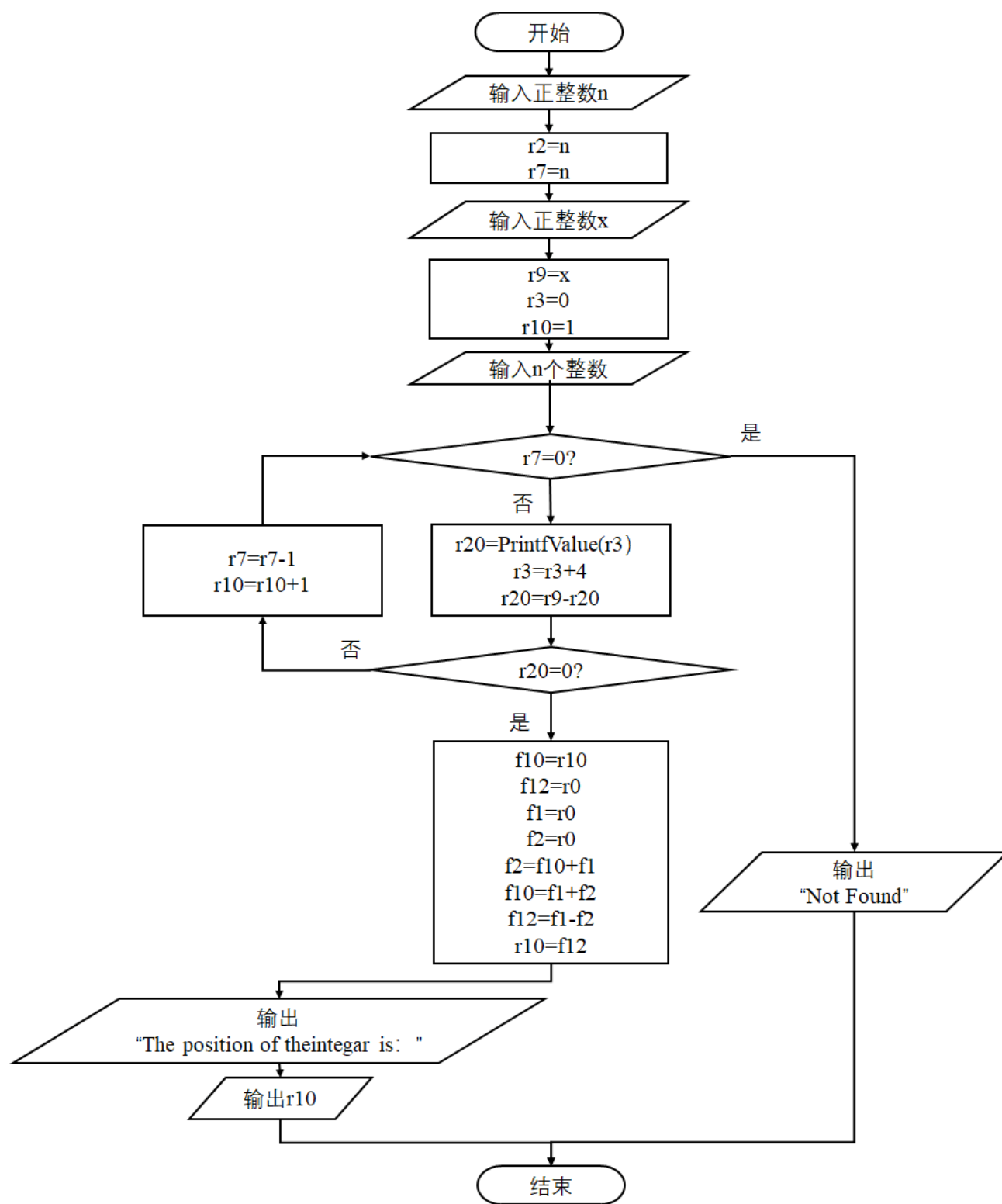


图 11 程序流程图

## 八、实验结论：

- 1、通过采用重定向技术减少了数据相关；
- 2、通过展开循环来减少控制相关；
- 3、通过增加硬件的数目来减少结构相关；
- 4、执行程序的顺序会影响程序执行是否正确，必须先执行源程序，再执行 `input.s`；修改后的程序必须清空之前所有的操作之后再重新运行。

## 九、总结及心得体会：

在具体的实验过程中，我发现曾经学习到的汇编语言指令与实验中的 **DLX** 指令相似，因此在实际学习的过程中上手较快，通过这段经历我既巩固了汇编语言程序设计基础，也学习了 **DLX** 汇编语言程序这门新的程序设计语言，在减少程序运行过程中出现的三种相关的过程中，加深了我对于数据相关、控制相关、结构相关的理解。在帮助其它同学的过程中，我发现部分同学的编程思维还停留在如 C, python 等高级程序语言的层面上，存在着将 `add`, `addi`, `addd` 等指令混用的情况，并没有意识到 **DLX** 指令集属于汇编语言的层级，即没有充分地理解计算机系统按照计算机语言的角度所划分的多级层次结构的这一特点。

## 报告评分：

指导教师签字：

# 实 验 报 告 2

学生姓名:	学 号:	时间: 地点:
实验课程名称: 计算机系统结构		
一、实验名称: 鲲鹏 Hyper Tuner 性能分析工具实验		
<p>二、实验原理:</p> <p>1、鲲鹏性能分析工具 Hyper Tuner</p> <p>Hyper Tuner 即性能分析工具, 支持鲲鹏平台上的系统性能分析、Java 性能分析和系统诊断, 提供系统全景及常见应用场景下的性能采集和分析功能, 并基于调优专家系统给出优化建议。同时提供调优助手, 指导用户快速调优系统性能。鲲鹏性能分析工具支持 IDE 插件(vs Code、IntelliJ)和浏览器两种工作模式, 分别同性能分析 Server 一起完成性能分析和优化等任务。</p> <p>2、矩阵乘法优化方法</p> <p>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。</p>		
<p>三、实验目的:</p> <p>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</p>		

#### 四、实验内容：

- 1、准备实验环境，可以使用华为云官方提供的沙箱实验室，也可以使用学校提供的华为云弹性服务器，根据实验教程安装依赖工具和 Hyper Tuner；
- 2、修改程序，因内存空间不足，修改数据规模 N 的定义，修改为 130000000；
- 3、登录 Hyper Tuner，编译并运行源代码，创建工程和任务进行系统性能全景分析。
- 4、创建进程/线程性能分析任务；
- 5、创建 C/C++性能分析任务；
- 6、编译程序并查看 multiply.c 程序中乘法函数 multiply 消耗时间和热点函数的占用率；编译 NEON 指令优化后的代码并查看消耗时间和占用率，进行对比；
- 7、阅读 multiply\_simd.c 源代码，通过互联网搜索 NEON 指令的文档，自主编写两个 N\*N 矩阵相乘的代码并进行优化；
- 8、将自主编写的代码上传至华为云服务器，编译执行，统计执行的时间进行对比

#### 五、实验器材（设备、元器件）：

电脑一台  
FinalShell（用于远程连接云服务器）  
华为云服务器  
鲲鹏 Hyper Tuner 性能分析工具

## 六、实验步骤及操作：

### 1、准备实验环境和安装依赖工具

(1) 购买华为云 ECS 弹性服务器

规格：鲲鹏通用计算增强型 kc1.xlarge.2 | 4vCPUs | 8GiB

镜像：CentOS 7.6 64bit with ARM

计费模式：按需计费

区域：北京四

服务器成功购买后，可以查看到对应的弹性公网 IP 和私有 IP。为了避免版本冲突，将 C/C++性能分析任务部署在另一个弹性云服务器上完成，所以本实验共使用两个弹性云服务器，两个服务器的信息和 IP 地址如图 1 所示。



图 1 本实验中的两个 ECS 弹性云服务器

(2) 配置安全组

配置安全组，并开放 8086 端口，如图 2 所示。

<input type="checkbox"/> 优先级 ?	策略 ?	协议端口 ?	类型	源地址 ?
<input type="checkbox"/> 1	允许	TCP : 20-21	IPv4	0.0.0.0/0 ?
<input type="checkbox"/> 1	允许	ICMP : 全部	IPv4	0.0.0.0/0 ?
<input type="checkbox"/> 1	允许	TCP : 3389	IPv4	0.0.0.0/0 ?
<input type="checkbox"/> 1	允许	TCP : 22	IPv4	0.0.0.0/0 ?
<input type="checkbox"/> 1	允许	TCP : 443	IPv4	0.0.0.0/0 ?
<input type="checkbox"/> 1	允许	TCP : 80	IPv4	0.0.0.0/0 ?
<input type="checkbox"/> 100	允许	全部	IPv6	sg-test ?
<input type="checkbox"/> 100	允许	全部	IPv4	sg-test ?
<input type="checkbox"/> 100	允许	TCP : 8086	IPv4	0.0.0.0/0 ?

图 2 安全组配置

(3) 连接服务器

通过 FinalShell 工具连接华为云服务器，成功连接后会得到图 3 中的提示。



```
连接主机...
连接主机成功
Last failed login: Wed Apr 19 09:09:31 CST 2023 from 223.104.17.98 on ssh:notty
There was 1 failed login attempt since the last successful login.

Welcome to Huawei Cloud Service
```

图3 FinalShell 成功连接提示

#### (4) 设置 SSH 超时断开时间，防止服务器断连

sed 命令是利用 script 来处理文本文件。-i：直接修改读取的文件内容，而不是输出到终端。输入指令后得到图 4 中的提示。

```
[root@xitong ~]# sed -i '112a ClientAliveInterval 600\nClientAliveCountMax 10' /etc/ssh/sshd_config && systemctl restart sshd
```

图4 设置超时断开后的 2 提示

#### (5) 安装 centos-release-scl

yum: Centos 软件安装常用命令。输入指令后得到图 5 中的提示。

```
/run/us... 348M/348M
已安装:
  centos-release-scl.noarch 0:2-3.el7.centos

作为依赖被安装:
  centos-release-scl-rh.noarch 0:2-3.el7.centos

完毕!
[root@xitong0420 ~]#
```

图5 Centos 安装成功后的提示

#### (6) 安装并激活 devtoolset

依次输入安装指令和激活后得到图 6 中的提示。

```
作为依赖被安装:
  audit-libs-python.aarch64 0:2.8.5-4.el7      checkpolicy.aarch64 0:2.5-8.el7
  devtoolset-7-runtime.aarch64 0:7.1-4.el7     gmp-devel.aarch64 1:6.0.0-15.el7
  libmpc-devel.aarch64 0:1.0.1-3.el7           libsemanage-python.aarch64 0:2.5-14.el7
  python-IPy.noarch 0:0.75-6.el7               scl-utils.aarch64 0:20130529-19.el7

作为依赖被升级:
  policycoreutils.aarch64 0:2.5-34.el7

完毕!
[root@xitong0420 ~]# scl enable devtoolset-7 bash
[root@xitong0420 ~]#
```

图6 devtoolset 安装并激活成功后的提示

(7) 安装 jdk 11 版本并在 home 目录下重命名为 jdk 文件夹

输入安装指令后得到图 7 中的提示.

```
bisheng-jdk-11.0.9/include/jawt.h
bisheng-jdk-11.0.9/include/jni.h
bisheng-jdk-11.0.9/include/jvmti.h
[root@xitong0420 home]#
```

图 7 jdk 安装成功后的提示

## 2. 登录 Hyper Tuner 并创建工程

(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.

输入安装指令后得到图 8 中的提示.

```
Hyper_tuner/
Hyper_tuner/Hyper-Tuner-2.2.T3.tar.gz
Hyper_tuner/crldata.crl
Hyper_tuner/file_list.txt
Hyper_tuner/file_list.txt.cms
Hyper_tuner/install.sh
[root@xitong0420 home]#
```

图 8 Hyper Tuner 安装成功后的提示

(2) 安装系统性能优化工具，默认安装在“/opt”目录下

将命令中的 SIP 替换为私有 IP 地址 192.168.1.83。中途出现输入提示，输入 Y，稍等片刻安装后，运行结果如图 9 所示。

```
Start hyper-tuner service success

Hyper_tuner install Success

=====
The login URL of Hyper_Tuner is https://192.168.1.83:8086/user-management/#/login
=====

If 192.168.1.83:8086 has mapping IP, please use the mapping IP.
[root@xitong0420 Hyper_tuner]#
```

图 9 系统性能优化工具安装成功后的提示

(3) 登录 Hyper Tuner

打开 edge 浏览器新建标签页，访问地 <https://120.46.150.240:8086/user-management/#/login>，首次登录需要创建管理员密码，默认用户名为 tunadmin，输入密码 tunadmin12#\$，如图 10 所示。



图 10 Hyper Tuner 登陆界面

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，如图 11 所示：

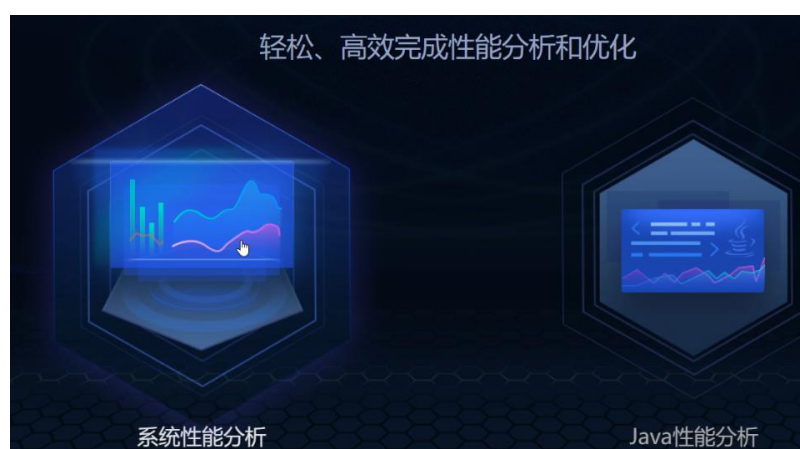


图 11 系统性能分析功能选择

### 3.编译运行“矩阵内存访问”代码

首先将 QQ 群中的测试程序压缩包解压到/opt/testdemo/multiply 文件夹下

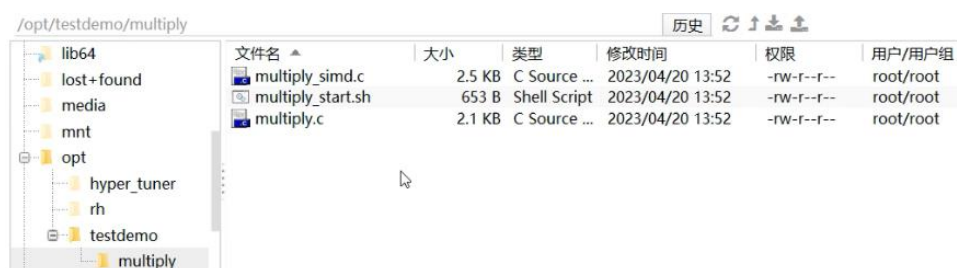


图 12 下载测试程序

再修改 multiply.c 和 multiply\_simd.c 中 N 的定义，修改为 130000000：

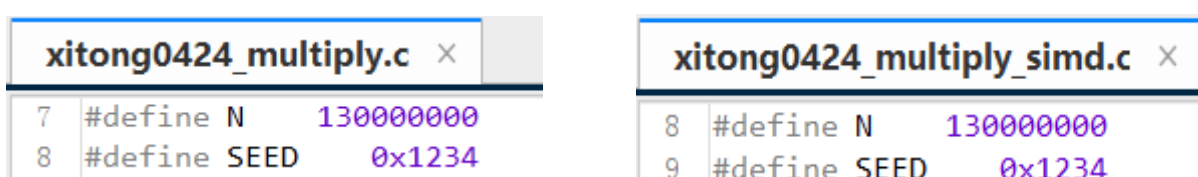
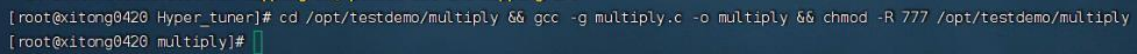


图 13 修改代码中的 N

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
```

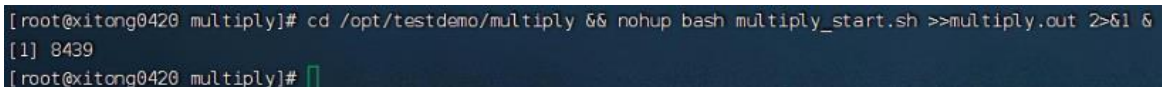


```
[root@xitong0420 Hyper_tuner]# cd /opt/testdemo/multiply && gcc -g multiply.c -o multiply && chmod -R 777 /opt/testdemo/multiply
[root@xitong0420 multiply]#
```

图 14 编译程序

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：

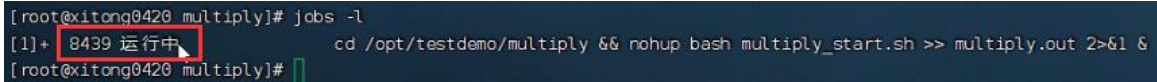
```
cd /opt/testdemo/multiply && nohup bash multiply_start.sh >>multiply.out 2>&1 &
```



```
[root@xitong0420 multiply]# cd /opt/testdemo/multiply && nohup bash multiply_start.sh >>multiply.out 2>&1 &
[1] 8439
[root@xitong0420 multiply]#
```

图 15 执行程序

执行命令：jobs -l，如果当前进程是“运行中”状态，如下图所示，则说明程序在运行状态，可以进行信息的采集。



```
[root@xitong0420 multiply]# jobs -l
[1]+  8439 运行中      cd /opt/testdemo/multiply && nohup bash multiply_start.sh >> multiply.out 2>&1 &
[root@xitong0420 multiply]#
```

图 16 程序运行状态检查

## 4.系统性能全景分析

### （1）创建系统性能全景分析任务

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：

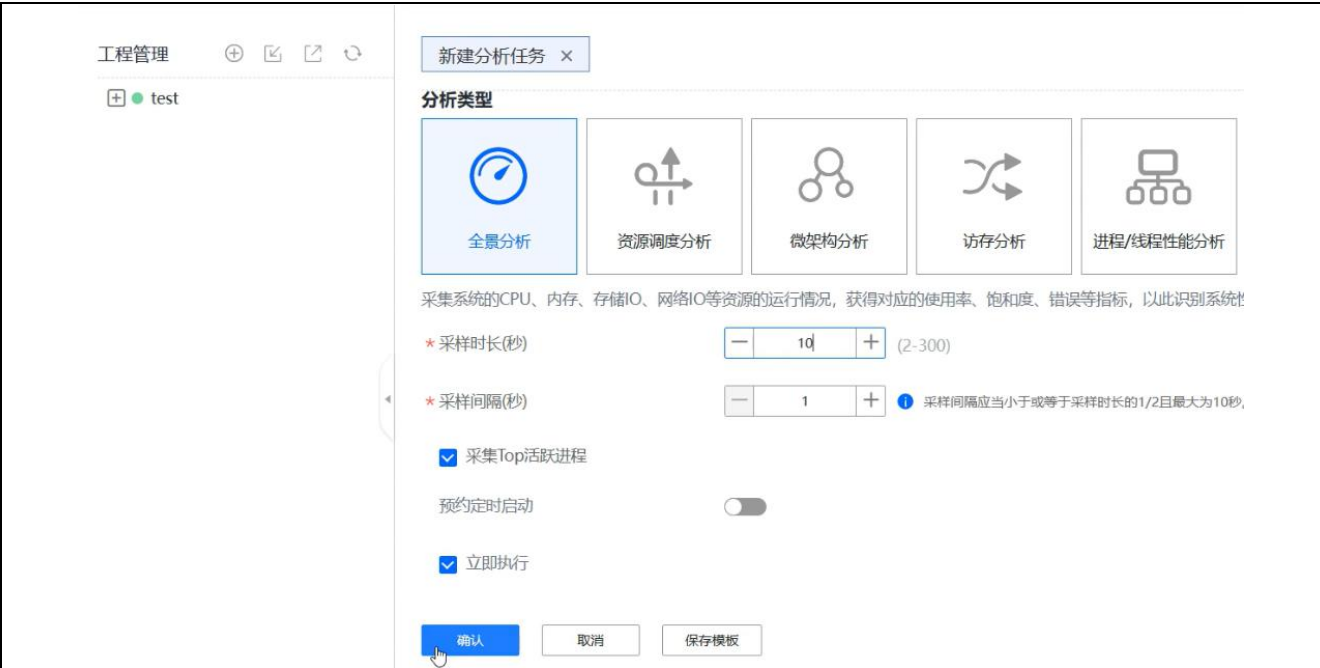


图 17 创建工程

将光标放到创建好的工程“test”上，会出现按钮，单击“+”创建任务，将任务名称设置为 multiply\_quanjing，选择分析对象设置为系统，分析类型选择全景分析，采样时长设置为 10 秒，采样间隔设置为 1 秒，并点击“确认”开始信息采集。



图 18 创建任务

(2) 查看采集分析结果

执行完毕后，显示全景分析的结果，点击“检测到 CPU 利用率高”显示优化建议。





图 19 全景分析结果总览

点击“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。



图 20 全景分析性能监测结果 1

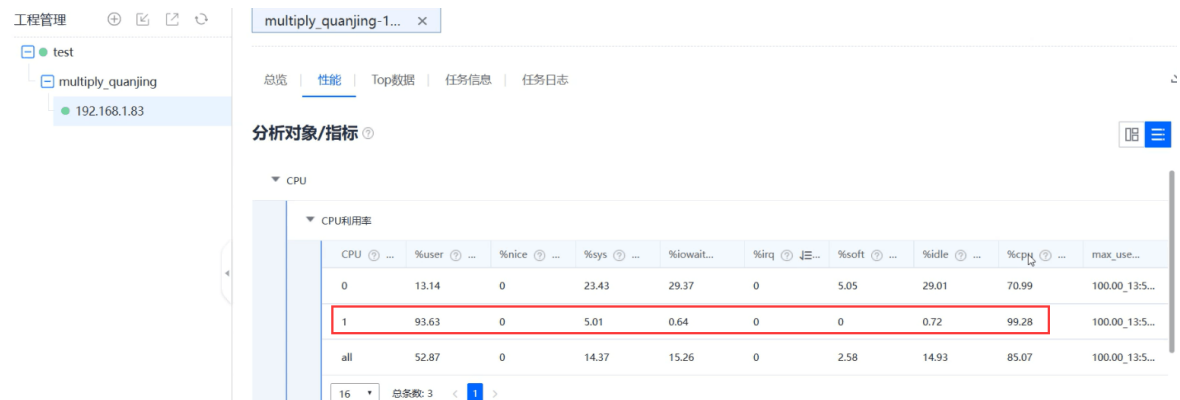


图 21 全景分析性能检测结果 2

由此可见，当前 CPU 核 1 的使用率（“性能”页签下%CPU 的数值）为 99.28%，接近 100%，并且 93.63%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。

## 5.进程/线程性能分析

### (1) 创建进程/线程性能分析任务

将光标放到创建好的工程“test”上，会出现按钮，单击“+”创建任务，将任务名称设

置为 multiply\_process，选择分析对象设置为系统，分析类型选择进程/线程性能分析，采样时长设置为 10 秒，采样间隔设置为 1 秒，采样类型全部勾选，采集线程信息选择打开，并点击“确认”开始信息采集。

Profile System即采集整个服务器系统，无需关注系统中有哪些类型的应用在运行，采样时长需要配置参数控制，适用于多:

**分析类型**

  
全景分析

  
资源调度分析

  
微架构分析

  
访存分析

  
进程/线程性能分析

采集进程/线程对CPU、内存、存储IO等资源的消耗情况，获得对应的使用率、饱和度、错误等指标，以此识别进程/线程性

\* 采样时长(秒)  (2-300)

\* 采样间隔(秒)  ⓘ 采样间隔应当小于或等于采样时长的1/2且最大为10秒。

\* 采样类型 ☐ CPU ☐ 内存 ☐ 存储IO ☐ 上下文切换

\* 采集线程信息 ☒

预约定时启动 ☐

☒ 立即执行

图 22 创建线程/进程性能分析任务

(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，如图 23 所示。

multiply\_quanjing-1... x multiply\_process-19... x

总览 | CPU | 内存 | 存储IO | 上下文切换 | 任务信息 | 任务日志

▼ CPU

	PID/TID	%user	%system	%wait	%CPU	Command
▶	PID 18271	93.27	5.77	0	99.04	multiply
▶	PID 18462	88.12	4.95	0	93.07	multiply
▶	PID 18316	71.05	9.65	0	80.70	multiply
▶	PID 18400	71.78	6.44	0	78.22	multiply
▶	PID 18370	59.00	2.00	0	61.00	multiply
▶	PID 18328	0	7.13	0	7.13	du
▶	PID 18465	0	6.47	0	6.47	du
▶	PID 375	0	4.80	0	4.80	kworker/0:1H-kblockd
▶	PID 18306	0.29	2.93	0	3.21	pidstat
▶	PID 18305	1.15	1.34	0	2.49	pidstat

10

总条数: 46 < 1 2 3 4 5 >

图 23 进程/线程性能分析结果

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，得到图 23 的结果。容易观察到 multiply 进程占用的 CPU 达到 99.04%，其中用户态占用了 93.27%的 CPUP，即 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。

## 6.C/C++性能分析

### (1) 安装最新版本的 Hyper Tuner 工具

由于 C/C++性能分析任务的完成需要安装最新版本的 Hyper Tuner 工具，因此需要输入以下指令下载 “Hyper-Tuner\_2.5.0.1\_linux.tar.gz” 到 “/home” 的根目录下。

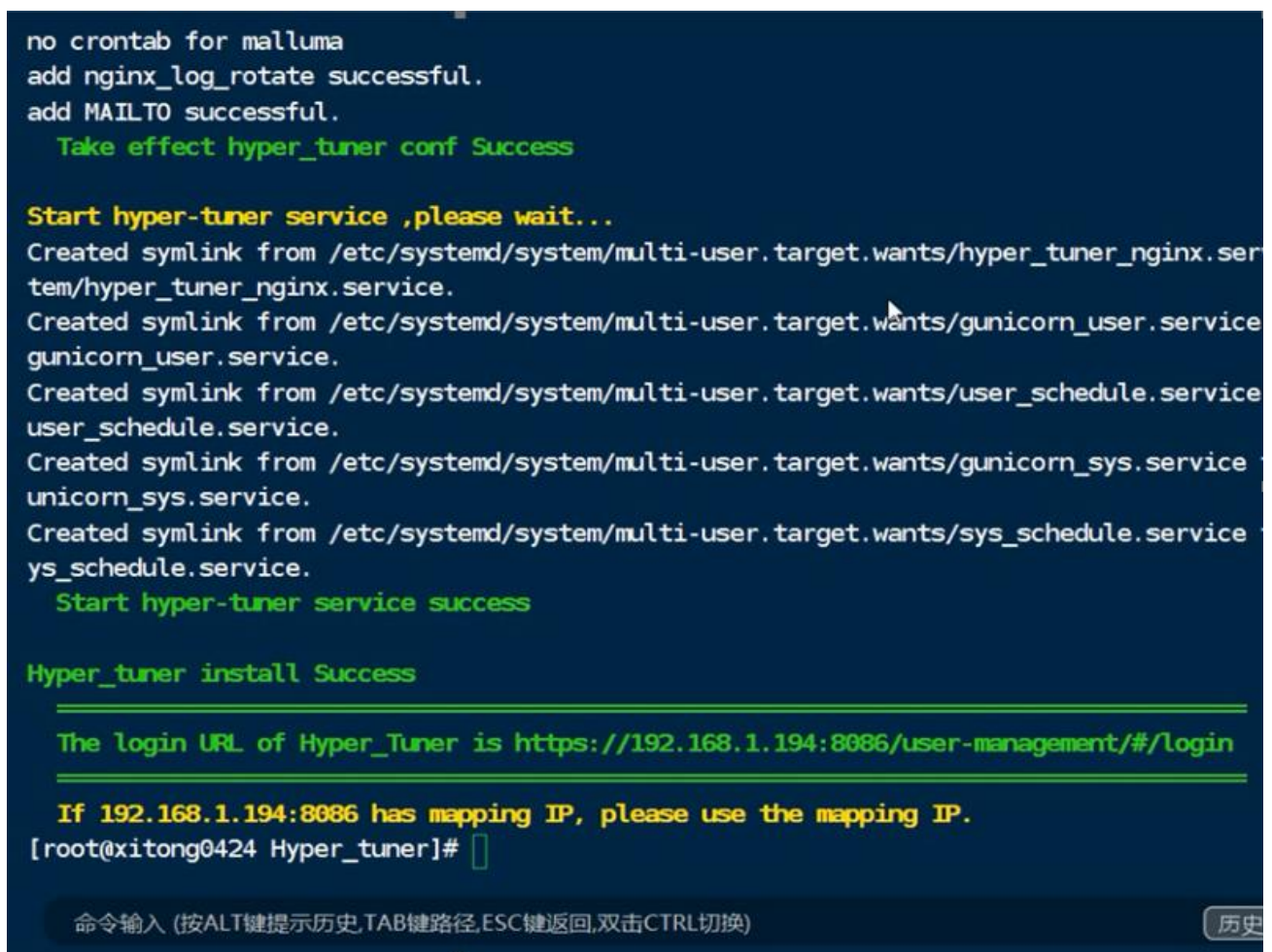
```
Wget https://kunpeng-repo.obs.cn-north-4.myhuaweicloud.com
```

```
/Hyper%20Tuner/Hyper%20Tuner%202.5.0.1/Hyper-Tuner_2.5.0.1_linux.tar.gz
```

将 “Hyper-Tuner\_2.5.0.1\_linux.tar.gz” 解压到本目录下，并输入以下指令完成工具的安装。

```
cd /home/Hyper_tuner && ./install.sh -a -i -ip=192.168.1.194 -jh=/home/jdk
```

工具安装完毕后，结果如图 24 所示。



```
no crontab for malluma
add nginx_log_rotate successful.
add MAILTO successful.
Take effect hyper_tuner conf Success

Start hyper-tuner service ,please wait...
Created symlink from /etc/systemd/system/multi-user.target.wants/hyper_tuner_nginx.service to /etc/systemd/system/hyper_tuner_nginx.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/gunicorn_user.service to /etc/systemd/system/gunicorn_user.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/user_schedule.service to /etc/systemd/system/user_schedule.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/gunicorn_sys.service to /etc/systemd/system/gunicorn_sys.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/sys_schedule.service to /etc/systemd/system/sys_schedule.service.
Start hyper-tuner service success

Hyper_tuner install Success

The login URL of Hyper_Tuner is https://192.168.1.194:8086/user-management/#/login

If 192.168.1.194:8086 has mapping IP, please use the mapping IP.
[root@xitong0424 Hyper_tuner]#
```

图 24 工具成功安装



(2) 创建热点函数分析任务

将光标放到创建好的工程“test”上，会出现按钮，单击“+”创建任务，将任务名称设置为 multiply\_C，选择分析对象设置为系统，分析类型选择热点函数分析，采样时长设置为 10 秒，采样间隔设置为 1 秒。



图 25 创建 C/C++性能分析任务

(3) 查看采集分析结果

执行完毕后，显示 C/C++性能分析的结果，如图 26 所示。

总览 函数 热火焰图 冷火焰图 任务信息 任务日志

优化建议 ^

检测到C/C++程序的CPU利用率高。

统计

10 10,707,000,000

数据采样时长 (s)

时钟周期

--

--

指令数

IPC

平台信息

操作系统 4.18.0-80.7.2.el7.aarch64 Linux

主机名 xitong0424

Top 10热点函数 Top 10热点模块

函数	模块	时钟周期	时钟周期百分比	执行时间 (s)
gen_data[0x400734,0x4008...	/opt/testdemo/multiply/multiply	4,886,000,000	45.63%	2.443000
multiply[0x4008a8,0x400938]	/opt/testdemo/multiply/multiply	4,468,000,000	41.73%	2.234000
unknown	[unknown]	162,000,000	1.51%	0.081000
_IO_vfscan[0x57f6c,0x5fbc0]	/usr/lib64/libc-2.17.so	46,000,000	0.43%	0.023000

图 26 C/C++性能分析结果

容易观察到 gen\_data 和 multiply 两个函数所占用的时钟周期百分比分别为 45.63% 和 41.73%，共占用了 87.36% 的时钟周期。

## 7.一维矩阵乘法 multiply 优化前后程序的性能分析

(1) 优化前程序的性能分析

①输入以下指令，编译程序。

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

```
&& chmod -R 777 /opt/testdemo/multiply
```

②输入以下指令，查看 multiply.c 程序中乘法函数 multiply 消耗时间，结果如图 27 所示。

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

```
[root@xitong0424 ~]# cd /opt/testdemo/multiply && gcc -g -O2 -o multiply multiply.c && chmod -R 777 /opt/testdemo/multiply
[root@xitong0424 multiply]# cd /opt/testdemo/multiply/ && ./multiply
217156.000000, 217156.000000, 217156.000000, 217156.000000
Execution time = 312.677 ms
[root@xitong0424 multiply]#
```

图 27 优化前乘法函数消耗时间

容易观察到优化前 multiply.c 程序中乘法函数 multiply 的消耗时间为 312.677ms。

③执行以下命令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
```

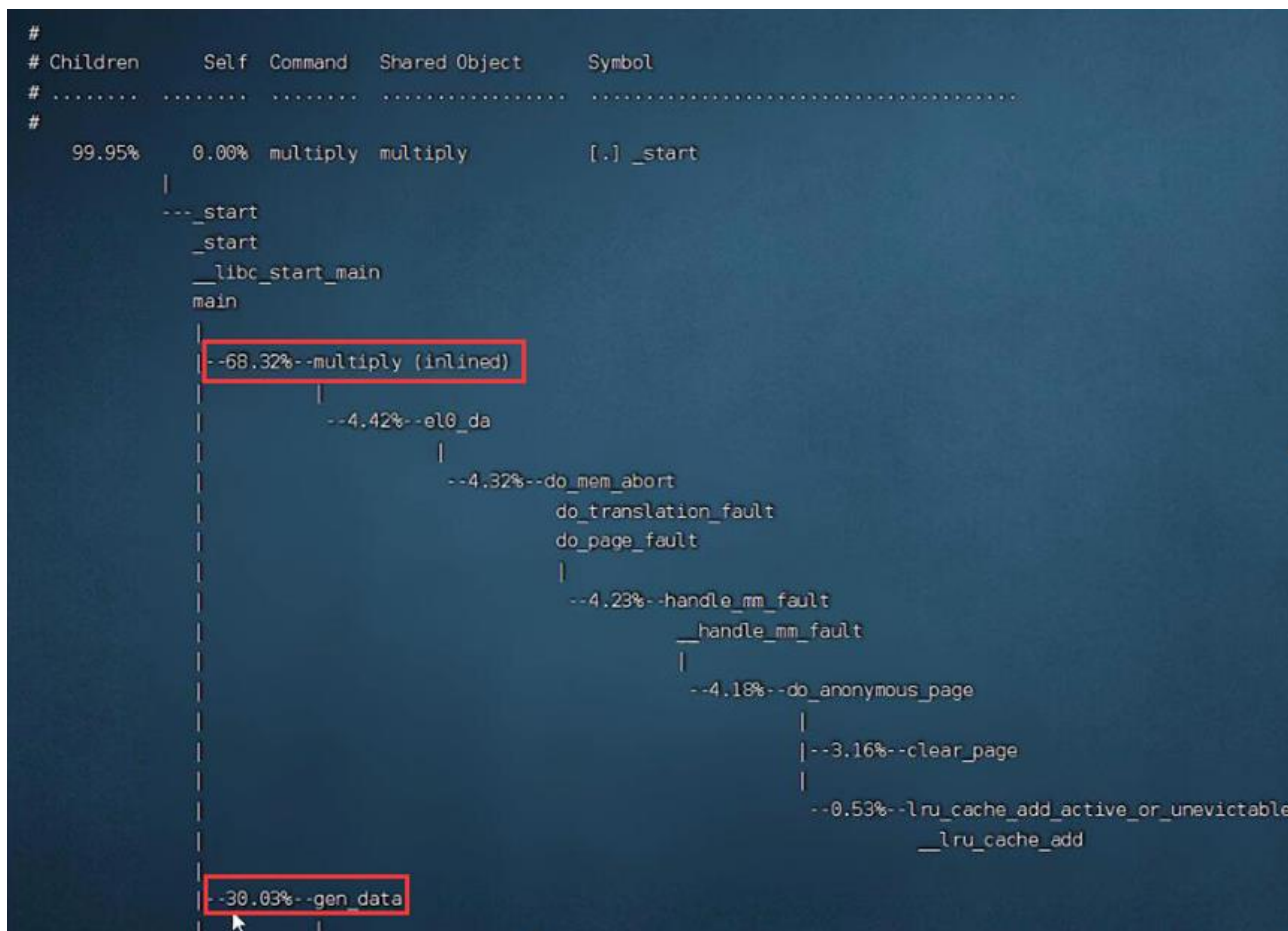


图 28 优化前 main 函数和子函数的 CPU 平均占用率

容易观察到优化前 multiply.c 程序中乘法函数 multiply 占用 68.32% 的 CPU，而生成数据函数 gen\_data 只占用了 30.03% 的 CPU，因此首先考虑优化 multiply 函数。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。

## (2) 优化后程序的性能分析

①输入以下指令，编译程序。

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

```
&& chmod -R 777/opt/testdemo/multiply
```

②输入以下指令，查看 multiply\_simd.c 程序中乘法函数 multiply 消耗时间，结果如图 29 所示。

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

```
[root@xitong0424 multiply]# cd /opt/testdemo/multiply && gcc -g -O2 -o multiply_simd multiply_simd.c && chmod -R 777 /opt/testdemo/multiply
[root@xitong0424 multiply]# cd /opt/testdemo/multiply/ && ./multiply_simd
217156.000000, 217156.000000, 217156.000000, 217156.000000
Execution time = 149.217 ms
[root@xitong0424 multiply]#
```

图 29 优化后乘法函数消耗时间

观察到优化后 multiply\_simd.c 程序中乘法函数 multiply\_neon 的消耗时间 149.217ms。

③执行以下命令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
```

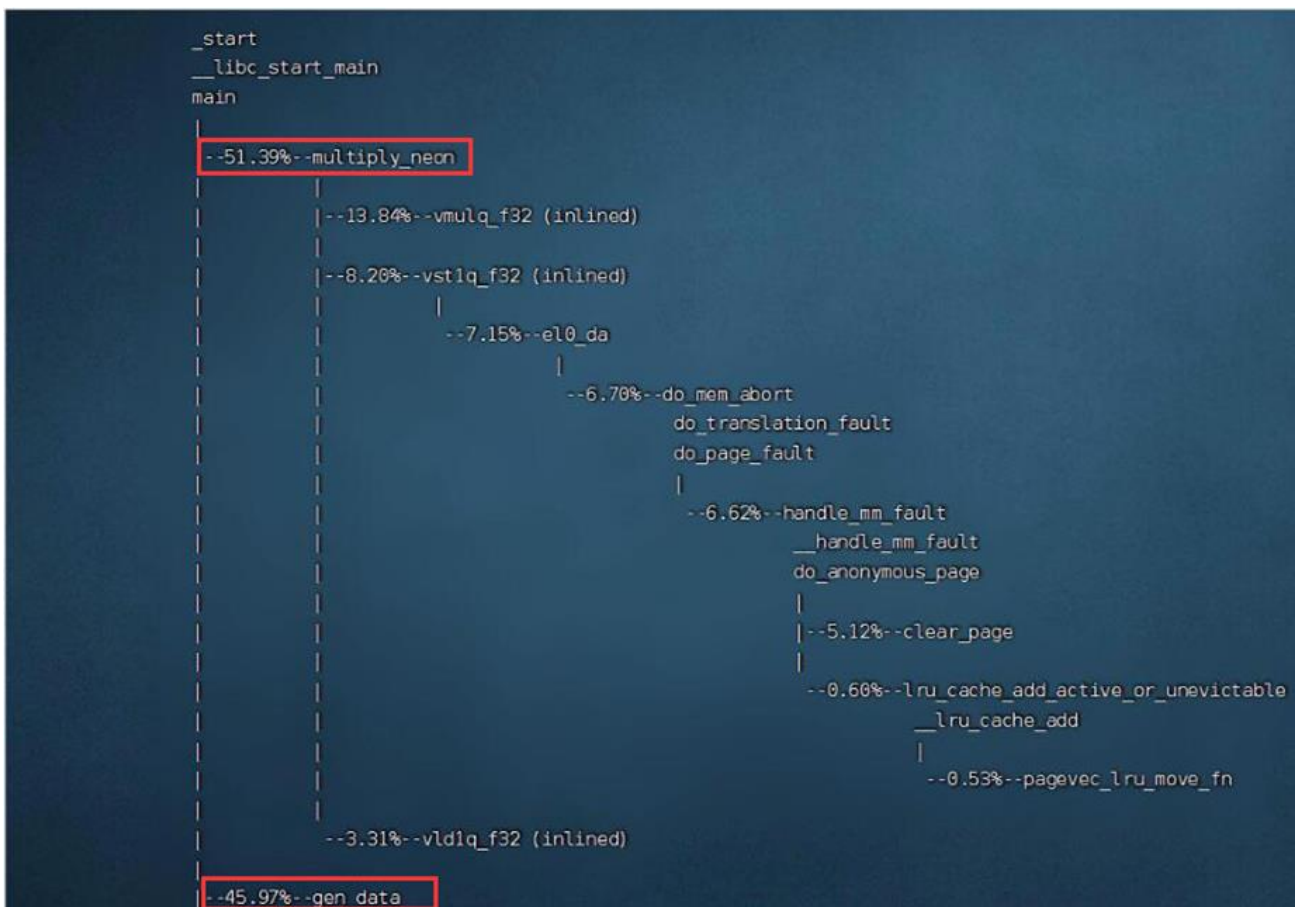


图 30 优化后 main 函数和子函数的 CPU 平均占用率

容易观察到图 30 中，优化后 `multiply_simd.c` 程序中乘法函数 `multiply_neon` 占用 51.39% 的 CPU，而生成数据函数 `gen_data` 只占用了 45.97% 的 CPU，对比优化前的 `multiply` 函数和优化后的 `multiply_neon` 函数，我们可以发现无论实在运行时间还是 CPU 占用率上，`multiply_neon` 函数都有明显的下降，说明优化有效。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。

## 8. 多维矩阵乘法 `matrix_multiply` 优化前后程序的执行时间对比

### (1) 优化前程序的执行时间

①输入以下指令，编译程序。

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

②输入以下指令，查看 `matrix_multiply.c` 程序中乘法函数 `matrix_multiply` 消耗时间。

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

```
[root@xitong0424 ~]# cd /opt/testdemo/multiply && gcc -g -O2 -o matrix_multiply matrix_multiply.c
[root@xitong0424 multiply]# cd /opt/testdemo/multiply/ && ./matrix_multiply
217156000.000000, 217156000.000000, 217156000.000000, 217156000.000000
Execution time = 1983.952 ms
```

图 31 优化前乘法函数消耗时间

结果如图 31 所示，容易观察到优化前多维矩阵乘法函数执行所消耗的时间为 1983.952ms。

### (2) 优化后程序的性能分析

①输入以下指令，编译程序。

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

②输入以下指令，查看 `multiply.c` 程序中乘法函数 `multiply_neon` 消耗时间，结果如图 32 所示，容易观察到优化前乘法函数执行所消耗的时间为 619.304ms。

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

```
[root@xitong0424 multiply]# cd /opt/testdemo/multiply && gcc -g -O2 -o matrix_multiply_simd matrix_multiply_simd.c
[root@xitong0424 multiply]# cd /opt/testdemo/multiply/ && ./matrix_multiply_simd
217156000.000000, 217156000.000000, 217156000.000000, 217156000.000000
Execution time = 619.304 ms
```

图 32 优化前乘法函数消耗时间

## 七、实验数据及结果分析：

### 1、一维矩阵相乘的主要代码及说明

(1) 未优化前的 multiply.c 的主要代码及说明

#### 一维矩阵计算

```
// 引入所需头文件

#include <stdlib.h>

#include <time.h>

#include <stdio.h>

#include <sys/time.h>

// 定义常量 N，表示矩阵中元素的数量

#define N 130000000

// 定义常量 SEED，用于生成随机数

#define SEED 0x1234

// 声明三个全局浮点指针，用于存储矩阵数据

float *g_a, *g_b, *g_c;

// 函数 gen_data 生成矩阵的数据

void gen_data(void)
{
    unsigned i;
    // 为每个矩阵分配内存空间
    g_a = (float*)malloc(N * sizeof(float));
    g_b = (float*)malloc(N * sizeof(float));
    g_c = (float*)malloc(N * sizeof(float));
    // 如果分配内存失败，输出错误信息并退出程序
    if (g_a == NULL || g_b == NULL || g_c == NULL) {
        perror("Memory allocation through malloc failed");
        exit(EXIT_FAILURE);
    }
    // 通过循环为矩阵 a 和 b 的每个元素赋值
```



```

    for (i = 0; i < N; i++) {
        // 使用 SEED * 0.1 来生成重复的随机数
        g_a[i] = (float)(SEED * 0.1);
        g_b[i] = (float)(SEED * 0.1);
    }
}

// 释放之前分配的内存空间
void free_data(void)
{
    free(g_a);
    free(g_b);
    free(g_c);
}

// multiply 函数实现矩阵 a 和 b 的元素相乘，并将结果存储在矩阵 c 中
void multiply(void)
{
    unsigned i;
    for (i = 0; i < N; i++) {
        g_c[i] = g_a[i] * g_b[i];
    }
}

// print_data 函数打印输出矩阵 c 的前两个和后两个元素
void print_data(void)
{
    // 打印前 2 个和最后 2 个元素
    printf("%f, %f, %f, %f\n", g_c[0], g_c[1], g_c[N - 2], g_c[N - 1]);
}

```

```

int main(void)
{
    double msec;
    struct timeval before, after;
    // 生成数据并执行乘法运算
    gen_data();
    gettimeofday(&before, NULL); // 记录运算开始时间
    multiply();
    gettimeofday(&after, NULL); // 记录运算结束时间

    // 使用 1000 来将 tv_sec 和 tv_usec 转换为毫秒
    msec = (after.tv_sec - before.tv_sec) * 1000.0 + (after.tv_usec - before.tv_usec) / 1000.0;
    // 输出运算结果和执行时间
    print_data();
    printf("Execution time = %2.3lf ms\n", msec);

    // 释放内存空间并退出程序
    free_data();
    return 0;
}

```

## （2）优化后的主要代码及说明

一维矩阵计算，采用 NEON 指令优化

//使用 Arm NEON 库对原程序中函数 multiply 进行优化。

```
#include <arm_neon.h>
```

//multiply\_neon 函数在处理大小为 4 的倍数的数据时优化计算速度。

```
void multiply_neon(void)
```

```
{
```

//首先用 vld1q\_f32 函数将两个相应位置的矩阵元素加载至 Arm NEON 向量中，

//然后将这两个向量相乘，并将结果存储到另一个向量 dst 中。

//vmulq\_f32 就是实现两个向量相乘的函数。



```

//接着使用 vst1q_f32 将结果保存到结果矩阵 g_c 的相应位置;
//当处理完所有大小为 4 的倍数的数据后, 处理剩余部分的数据,
int i;
float32x4_t src1, src2, dst;
// 处理大小为 4 的倍数的数据
for (i = 0; i < (N & ((~(unsigned)0x3))); i += 4) {
    // 将地址从 g_a+i 开始的四个浮点数加载进向量 src1 中
    src1 = vld1q_f32(g_a + i);
    // 将地址从 g_b+i 开始的四个浮点数加载进向量 src1 中
    src2 = vld1q_f32(g_b + i);
    dst = vmulq_f32(src1, src2);    // 对两个向量进行相乘操作
    vst1q_f32(g_c + i, dst);        // 将结果存到 g_c 的相应位置
}
// 处理剩余部分的数据
for (; i < N; i++) {
    g_c[i] = g_a[i] * g_b[i];
}
}

```

## 2、二维矩阵相乘的主要代码及说明

### (1) 优化前的主要代码及说明

#### 二维矩阵计算

```

// 引入头文件
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <sys/time.h>
// 设置矩阵维度 1000*1000
#define N    1000
#define SEED 0x1234
// 将三个矩阵的数据类型设置为 double
double *matirx_a, *matirx_b, *matirx_c;

```

// 函数 gen\_data 生成矩阵的数据

void gen\_data(void)

```
{
    unsigned i;
    matirx_a = (double*)malloc(N * N * sizeof(double));
    matirx_b = (double*)malloc(N * N * sizeof(double));
    matirx_c = (double*)malloc(N * N * sizeof(double));
    if (matirx_a == NULL || matirx_b == NULL || matirx_c == NULL) {
        perror("Memory allocation through malloc failed");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < N*N; i++) {
        // 随机为矩阵生成数据
        matirx_a[i] = (double)(SEED * 0.1);
        matirx_b[i] = (double)(SEED * 0.1);
    }
}
```

// 释放所分配给各矩阵的空间

void free\_data(void)

```
{
    free(matirx_a);
    free(matirx_b);
    free(matirx_c);
}
```

// 进行矩阵乘法计算

void multiply(void)

```
{
    unsigned i,j,k;
    //将矩阵 a 按行存储，矩阵 b 按列存储，以便于用 NEON 指令进行优化，
    for (i = 0; i < N; i++) {
```

```

        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                matirx_c[i*N+j] += matirx_a[i*N+k] * matirx_b[j*N+k];
            }
        }
    }
}

void print_data(void)
{
    // 输出矩阵前两个和最后两个位置的数据
    printf("%f, %f, %f, %f\n", matirx_c[0], matirx_c[N-1], matirx_c[(N-1)*N],
matirx_c[N*N-1]);
}

int main(void) {
    double msec;
    struct timeval before, after;
    // 生成数据并执行乘法运算
    gen_data();
    gettimeofday(&before, NULL);// 记录运算开始时间
    multiply();
    gettimeofday(&after, NULL);// 记录运算结束时间

    // 使用 1000 来将 tv_sec 和 tv_usec 转换为毫秒
    msec = (after.tv_sec - before.tv_sec) * 1000.0 + (after.tv_usec - before.tv_usec) / 1000.0;

    // 输出运算结果和执行时间
    print_data();
    printf("Execution time = %2.3lf ms\n", msec);
}

```

```
// 释放内存空间并退出程序
```

```
free_data();
```

```
return 0;
```

```
}
```

## (2) 优化后的主要代码及说明

一维矩阵计算，采用NEON指令优化

```
// 引入 NEON 相关头文件
```

```
#include <arm_neon.h>
```

```
// 使用 NEON 指令进行矩阵乘法计算的优化
```

```
void multiply_neon(void)
```

```
{
```

```
    unsigned i,j,k;
```

```
    // 用两个 float64x2_t 类型的向量存储四个数据
```

```
    float64x2_t srca1,srca2,srcb1,srcb2,tmp1,tmp2,dst1,dst2;
```

```
    for (i = 0; i < N; i++) {    // 遍历 a 的行
```

```
        for (j = 0; j < N; j++) {
```

```
            // 将向量 dst1 和 dst2 进行清零，以重新使用两个向量
```

```
            dst1 = vdupq_n_f64(0.0f);
```

```
            dst2 = vdupq_n_f64(0.0f);
```

```
            for (k = 0; k < (N & ((~(unsigned)0x3))); k+=4) {
```

```
                srca1 = vld1q_f64(matirx_a + i*N+k);
```

```
                srca2 = vld1q_f64(matirx_a + i*N+k+2);
```

```
                srcb1 = vld1q_f64(matirx_b + j*N+k);
```

```
                srcb2 = vld1q_f64(matirx_b + j*N+k+2);
```

```
                // 将对应位置的数据相乘
```

```
                tmp1 = vmulq_f64(srca1, srcb1);
```

```
                tmp2 = vmulq_f64(srca2, srcb2);
```

```
                dst1 = vaddq_f64(tmp1,dst1);
```

```
                dst2 = vaddq_f64(tmp2,dst2);
```

```
            }
```

```

        // 从向量 dst1 和 dst2 取数，并相加得到乘积总和
        matirx_c[i*N+j] = vgetq_lane_f64(dst1,0) + vgetq_lane_f64(dst1,1) + vgetq_lane_f64(dst2,0)
+ vgetq_lane_f64(dst2,0);

        // 将未被计算的剩余部分加上以求和
        for (; k < N; k++){
            matirx_c[i*N+j] += matirx_a[i*N+k] * matirx_b[j*N+k];
        }
    }
}
}

```

## 八、实验结论：

- 1、通过采用 NEON 指令对一维矩阵乘法进行优化，可以在寄存器中同时加载四个数据，通过大幅减少程序执行过程中所需的操作次数，提高程序执行过程中的并行程度，并最终提高程序的运行效率。
- 2、对于一维矩阵乘法和二维矩阵乘法两个程序，通过使用 NEON 指令和 SIMD 技术进行优化，可以大幅减少程序执行过程中函数的指令数目，缩短程序的运行时间并最终提升程序的执行效率。

## 十、总结及心得体会：

在本次的实验中，我首先掌握了如何部署合适的华为云弹性云服务器以完成任务。之后学会了使用鲲鹏性能分析工具 Hyper Tuner 创建系统性能分析任务、进程/线程分析任务和热点函数分析任务的方法。其次我学会了在优化矩阵乘法问题时，可以使用 NEON 指令,利用 SIMD 技术将对单个数据的操作扩展为对同一类型元素矢量的操作，通过大幅减少操作次数以此提升矩阵乘法执行效率，并实际分析了一维矩阵乘法和二维矩阵乘法两个案例，最后对比分析了它们在优化前后的消耗时间和占用率。

## 报告评分：

指导教师签字：