利用鲲鹏处理器的流水线来优化汇编代码性能实验

版本：1.0



华为技术有限公司

目录

[1 课程介绍 2](#_Toc50553612)

[1.1 简介 2](#_Toc50553613)

[1.2 内容描述 2](#_Toc50553614)

[1.3 读者知识背景 2](#_Toc50553615)

[1.4 实验环境说明 3](#_Toc50553616)

[2 实验：利用鲲鹏处理器的流水线来优化汇编代码性能实验 4](#_Toc50553617)

[2.1 实验介绍 4](#_Toc50553618)

[2.1.1 关于本实验 4](#_Toc50553619)

[2.1.2 教学目标 4](#_Toc50553620)

[2.1.3 实验内容介绍 4](#_Toc50553621)

[2.2 实验任务操作指导 5](#_Toc50553622)

[2.2.1 创建原始示例程序源码 5](#_Toc50553623)

[2.2.2 进行原始程序的编译运行 6](#_Toc50553624)

[2.2.3 代码的第一阶段改进 6](#_Toc50553625)

[2.2.4 代码第一阶段改进版的编译运行 7](#_Toc50553626)

[2.2.5 代码的第二阶段改进 8](#_Toc50553627)

[2.2.6 代码第二阶段改进版的编译运行 9](#_Toc50553628)

[2.3 思考题 10](#_Toc50553629)

# 课程介绍

## 简介

本手册适用于学习ARM平台汇编课程的学生进行实验练习，完成本实验手册后，您将能掌握华为鲲鹏云服务器环境配置，并更加充分理解GNU ARM汇编代码运行环境的搭建、配置及汇编代码的编译运行。

## 内容描述

鲲鹏920处理器的流水线支持2个访存指令的发射队列，如下图中的红色框内所示，其访存单元WB支持每拍2条读或写访存指令。本实验指导书通过在华为鲲鹏云服务器上，利用该特性来不断改进代码的实现方式提高代码的运行效率。完成实验操作后，读者会了解到双发射队列流水线指令的适用范围，并掌握如何利用该特性编写和改进代码程序。



鲲鹏920处理器流水线结构图

## 读者知识背景

本课程为GNU ARM汇编基础课程，为了更好地掌握本书内容，阅读本手册的读者应首先具备以下基本条件：

* 具备基本的Linux命令能力。
* 了解GNU ARM汇编中的代码结构、指令格式、伪指令等基础知识。

## 实验环境说明

* 华为鲲鹏云主机、openEuler20.03操作系统。
* 安装gcc7.3+版本。
* 每套实验环境可供1名学员上机操作。

# 实验：利用鲲鹏处理器的流水线来优化汇编代码性能实验

## 实验介绍

### 关于本实验

实现GNU ARM汇编中如何利用Aarch64 架构“其访存单元支持每拍2条读或写访存指令”的特性，来提升改进代码，提高代码执行效率。

### 教学目标

掌握利用Aarch64架构下的提高汇编代码执行效率的方式。

### 实验内容介绍

关于在C代码和汇编代码之间进行参数传递，根据Arm公司的AAPCS64，即Aarch64程序调用标准，Aarch64标准提供了8个通用寄存器（x0-x7）用于传递函数参数，依次对应于参数0、参数1、参数2…参数7。第8个参数需要通过sp访问，第9个参数需要通过sp + 8 访问，第n个参数需要通过sp + 8\*(n-8)访问。一般来说，对于只带有少量参数的函数，仅使用寄存器就足够了；超过8个的参数会存放在堆栈中用于传递给子例程。在本例子中，需要传递的参数有三个：第一个参数是目标字符串的首地址，用寄存器X0来传递；第二个参数是源字符串的首地址，用寄存器X1来传递；第三个参数是传输的字节数目，用寄存器X2来传递。

在使用ldrb/ldp和str/stp等访存指令时，要注意区分这三种形式：

* 前索引方式，形如：ldrb w2,[X1,#1] //将x1+1指向的地址处的一个字节放入w2中；x1寄存器的值保持不变。
* 自动索引方式，形如：ldrb w2,[X1,#1]！ //将x1+1指向的地址处的一个字节放入w2中；然后x1+1->x1。
* 后索引方式，形如ldrb w2,[X1],#1 //将x1指向的地址处的一个字节放入w2中，然后x1+1->x1。

该程序由两部分组成：第一部分是主函数，采用Linux C语言编码，用来测试内存拷贝函数的执行时间；第二部分是内存拷贝函数，采用GNU Arm64汇编语言编码。在下面的代码中，用到了上面三种形式的指令，需仔细体会其不同。

为了较为准确的测量内存拷贝函数memorycopy()的执行时间，调用了clock\_gettime()来分别记录memorycopy()执行前和执行后的系统时间，以纳秒为计时单位。

## 实验任务操作指导

### 创建原始示例程序源码

以下步骤以在华为鲲鹏云服务器上执行为例。

执行以下命令，创建memorycopy目录存放该程序的所有文件, 并进入该目录

cd

mkdir memorycopy

cd memorycopy

执行以下命令，创建主函数的Linux C代码memorycopy.c

vim memorycopy.c

代码内容如下：

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define len 60000000

char src[len],dst[len];

long int len1=len;

extern void memorycopy(char \*dst,char \*src,long int len1);

int main()

{

struct timespec t1,t2;

int i,j;

for(i=0;i<len-1;i++)

{

src[i]='a';

}

src[i]=0;

clock\_gettime(CLOCK\_MONOTONIC,&t1);

memorycopy(dst,src,len1);

clock\_gettime(CLOCK\_MONOTONIC,&t2);

printf("memorycopy time is %11u ns\n",t2.tv\_nsec-t1.tv\_nsec);

return 0;

}

执行以下命令，创建内存拷贝函数的GNU Arm64汇编语言编码copyfunc.s

vim copyfunc.s

代码内容如下：

.global memorycopy

memorycopy:

ldrb w3,[x1],#1

str w3,[x0],#1

sub x2,x2,#1

cmp x2,#0

bne memorycopy

ret

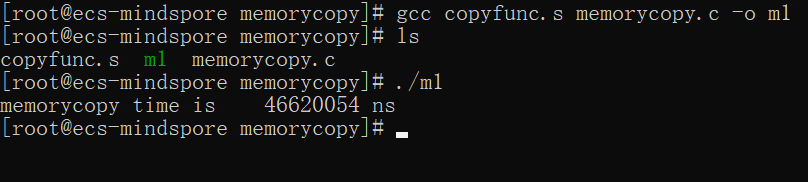
说明：内存拷贝函数memorycopy()的功能是实现将尺寸为len（这里设为60000000）的src字符数组的内容拷贝到同样尺寸的dst字符数组中。 memorycopy()函数用Arm64/Aarch64汇编代码实现。根据所用访存指令和循环展开粒度的不同，可以有多种实现方式。以上的汇编代码是最原始的方式，不进行循环展开，每次循环只使用1个ldrb和1个str指令。

### 进行原始程序的编译运行

保存示例源码文件，然后退出vim编辑器。在当前目录中依次执行以下命令，进行代码编译运行。

gcc copyfunc.s memorycopy.c -o m1

./m1



通过上述代码运行，可以看出,原始方法编写的memorycopy示例程序已经在华为鲲鹏云服务器上通过编译和运行，已经成功输出结果，可以看到执行时间为46620054纳秒。

### 代码的第一阶段改进

采用循环展开的方法，充分利用流水线的多发射机制，对函数memorycopy()原始汇编代码主体部分的两种改进，方式如下：

方法1：循环展开的宽度为2。将该方法命名为copyfunc\_v2\_1.s，汇编代码内容如下：

.global memorycopy

memorycopy:

sub x1,x1,#1

sub x0,x0,#1

lp:

ldrb w3,[x1,#1]

ldrb w4,[x1,#2]!

str w3,[x0,#1]

str w4,[x0,#2]!

sub x2,x2,#2

cmp x2,#0

bne lp

ret

方法2：循环展开的宽度为4。将该方法命名为copyfunc\_v2\_2.s，汇编代码内容如下：

.global memorycopy

memorycopy:

sub x1,x1,#1

sub x0,x0,#1

lp:

ldrb w3,[x1,#1]

ldrb w4,[x1,#2]

ldrb w5,[x1,#3]

ldrb w6,[x1,#4]!

str w3,[x0,#1]

str w4,[x0,#2]

str w5,[x0,#3]

str w6,[x0,#4]!

sub x2,x2,#4

cmp x2,#0

bne lp

ret

### 代码第一阶段改进版的编译运行

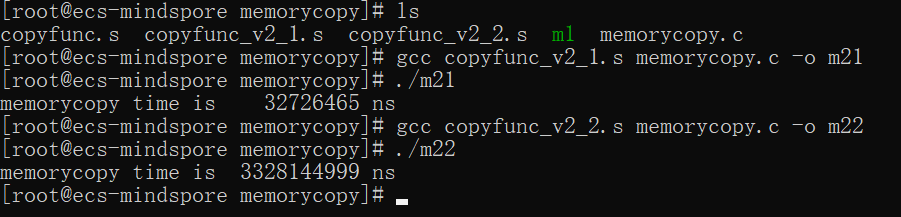
分别保存两段新的汇编源码文件，并退出vim编辑器。在当前目录中依次执行以下命令，进行代码编译运行。

gcc copyfunc\_v2\_1.s memorycopy.c -o m21

./m21

gcc copyfunc\_v2\_2.s memorycopy.c -o m22

./m22



从以上的测试结果可以看出，这两种改进后代码的性能都要优于原始代码，这两种改进代码之间的性能相差则不大。

原因分析：鲲鹏920有两个load/store流水线，其访存单元支持每拍2条读或写访存操作。原始代码下由于源字符串的地址和目标的字符串的地址并不连续，而且这种不连续地址的一读一写交替进行，导致内存访问的连续性很差，cache命中率较低。另一方面，由于循环分支指令较多，会经常刷新流水线，这也大大降低了两个访存指令发射队列中访存指令的充满速度。比起原始代码，改进代码则有效改进了这些缺点，使得其访存延迟大大降低。

### 代码的第二阶段改进

第一次改进中中每次读/写内存都是以一个字节为单位进行的，其访存效率较低。可以采用一次读/写16个字节的方法，充分利用内存突发传输方式的优势（即内存在连续读/连续写多个数据时，其性能要优于非连续读/写数据的方式），对上一节的代码再次进行改进。Arm64/Aarch64提供了ldp指令和stp指令，这两条指令可以一次访问16个字节的内存数据，其读/写内存的连续性非常高，可以有效降低访存延时。使用ldp和stp指令进行改进有如下三种典型的改进方式：

方法1：未经循环展开。将该方法命名为copyfunc\_v3\_1.s，汇编代码内容如下：

.global memorycopy

memorycopy:

ldp x3,x4,[x1],#16

stp x3,x4,[x0],#16

sub x2,x2,#16

cmp x2,#0

bne memorycopy

ret

方法2：循环展开的宽度为2。将该方法命名为copyfunc\_v3\_2.s，汇编代码内容如下：

.global memorycopy

memorycopy:

sub x1,x1,#16

sub x0,x0,#16

lp:

ldp x3,x4,[x1,#16]

ldp x5,x6,[x1,#32]!

stp x3,x4,[x0,#16]

stp x5,x6,[x0,#32]!

sub x2,x2,#32

cmp x2,#0

bne lp

ret

方法3：循环展开的宽度为4。将该方法命名为copyfunc\_v3\_3.s，汇编代码内容如下：

.global memorycopy

memorycopy:

sub x1,x1,#16

sub x0,x0,#16

lp:

ldp x3,x4,[x1,#16]

ldp x5,x6,[x1,#32]

ldp x7,x8,[x1,#48]

ldp x9,x10,[x1,#64]!

stp x3,x4,[x0,#16]

stp x5,x6,[x0,#32]

stp x7,x8,[x0,#48]

stp x9,x10,[x0,#64]!

sub x2,x2,#64

cmp x2,#0

bne lp

ret

### 代码第二阶段改进版的编译运行

分别保存两段新的汇编源码文件，并退出vim编辑器。在当前目录中依次执行以下命令，进行代码编译运行。

gcc copyfunc\_v3\_1.s memorycopy.c -o m31

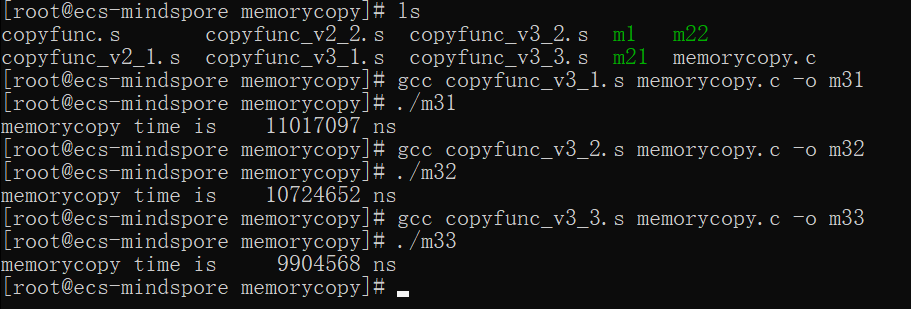
./m31

gcc copyfunc\_v3\_2.s memorycopy.c -o m32

./m32

gcc copyfunc\_v3\_3.s memorycopy.c -o m33

./m33



经测试，上述三种改进代码的性能要远远高于第一阶段的改进代码。

在上述各改进代码中，各同一类型的访存指令之间（比如各ldrb指令之间，各stp指令之间）是没有依赖关系的（既没有寄存器资源的依赖，也没有数据的依赖），因此在循环展开之后就可以充分两条load/store流水线来并行地执行这些访存指令，从而有效提升访存性能。如果改进代码中访存指令之间存在依赖关系，例如如下的情况：

ldrb w3,[x1],#1

ldrb w4,[x1],#1

…..

后一条ldrb指令中x1的取值依赖于前一条ldrb执行完毕之后x1取值的更新，那么这种改进的效果就会差很多。感兴趣的读者可以试写一下这种有依赖关系的代码，将其与无依赖关系的改进代码进行性能对比。

## 思考题

* 采用以2byte、4byte、8byte为基本传输单位的访存指令，其各自性能如何？
* 如下两种代码写法，哪一种性能更优？为什么？

代码1：

memorycopy:

sub x1,x1,#1

sub x0,x0,#1

lp:

ldrb w3,[x1,#1]!

ldrb w4,[x1,#1]!

str w3,[x0,#1]!

str w4,[x0,#1]!

sub x2,x2,#2

cmp x2,#0

bne lp

代码2：

memorycopy:

sub x1,x1,#1

sub x0,x0,#1

lp:

ldrb w3,[x1,#1]

ldrb w4,[x1,#2]!

str w3,[x0,#1]

str w4,[x0,#2]!

sub x2,x2,#2

cmp x2,#0

bne lp

* 是否可以采用8路/16路/32路…循环展开的方式？如果可以，会遇到什么问题？性能会怎样变化？
* 查阅一下Linux Arm64环境下，C函数memcpy的汇编源代码的实现过程（例如：<https://blog.csdn.net/forever_2015/article/details/50286009>），并测试其性能。
* 有一段C语言代码，如下所示：

for(i=0;i<10000;i++,out++)

\*out=array[\*i];

是否可以基于相同的原理来对其进行改写，以优化其性能？

说明：《汇编与接口技术》课程配套实验手册中的实验内容由北京交通大学计算机与信息技术学院赵宏智老师提供，华为公司负责实验手册文档的编写。