

实验报告

开课学期： 2022秋季

课程名称： 计算机体系结构(实验)

实验名称： 实验1：Pin入门及指令

依赖距离分析

学生班级： 6

学生学号： 200110618

学生姓名： 邓皓元

实验与创新实践教育中心印制

2022年11月

# 1. 样例程序分析

*借助Pin的API文档，阅读ManualExamples目录下inscount0、inscout1和inscout2工具的源码，分析插桩工具的代码框架及其执行过程。*

inscount0展示了指令级插桩来检测分析指令数量的插桩方法，而inscount1、inscount2则展示了轨迹级插桩来检测分析指令数量的插桩方法。

以inscount0作为例子分析插桩工具的代码框架和执行过程：

框架上先初始化引脚，选定一个级别的插桩方法传入对应插桩方法的参数调用分析函数，编写分析函数，设置程序结束后输出插桩分析结果，启动程序。

代码部分：

静态变量记录指令数量

static UINT64 icount = 0;

对指令执行该函数来使记录的指令数量加一

VOID docount() { icount++; }

对每条指令执行的检测分析函数

VOID Instruction(INS ins, VOID\* v)

{

调用函数来增加指令记录数量

INS\_InsertCall(ins, IPOINT\_BEFORE, (AFUNPTR)docount, IARG\_END);

}

用以执行该插桩方法的函数

KNOB< string > KnobOutputFile(KNOB\_MODE\_WRITEONCE, "pintool", "o", "inscount.out", "specify output file name");

程序结束后输出插桩分析结果

VOID Fini(INT32 code, VOID\* v)

{

OutFile.setf(ios::showbase);

OutFile << "Count " << icount << endl;

OutFile.close();

}

插桩初始化失败则调用该函数

INT32 Usage()

{

cerr << "This tool counts the number of dynamic instructions executed" << endl;

cerr << endl << KNOB\_BASE::StringKnobSummary() << endl;

return -1;

}

主函数

int main(int argc, char\* argv[])

{

初始化针脚插桩，初始化失败调用usage

if (PIN\_Init(argc, argv)) return Usage();

OutFile.open(KnobOutputFile.Value().c\_str());

对每条指令调用Instruction函数(参数对应的Instruction函数和0寄存器)

INS\_AddInstrumentFunction(Instruction, 0);

程序结束调用Fini(参数对应的Fini函数和0寄存器)

PIN\_AddFiniFunction(Fini, 0);

开始程序，不返回

PIN\_StartProgram();

return 0;

}

# 2. 设计与实现

*\*注：若做了附加题，除了提交相应代码，还需按照指导书要求，在报告中添加相应的分析、设计过程和运行结果截图和对比等内容，否则不加分。*

## 2.1 方案设计

*详细阐述insDependDist插桩工具的设计方案。对于设计中的关键问题，应给出合理分析和解决方法。建议结合算法流程图等图形化工具来辅助描述。*

原来的insDependDist：

B. 遍历regs->write向量, 利用lastInsPointer数组记录当前指令的被写寄存器所对应的PC值，即对任意的r属于regs->write, 将当前PC值赋值给lastInsPointer[r].

for (vector<reg\_t>::iterator it = regs->write.begin(); it != regs->write.end(); it++)

首先遍历write向量，将当前PC值赋给当前指令被写寄存器(相当于初始化lastInsPointer[r]，所以需要先执行write向量的遍历(若同一指令被写寄存器也是被读寄存器，且该寄存器还没有保存对应PC值，会出现使用未初始化变量的错误))

lastInsPointer[\*it] = insPointer; // TODO

A. 遍历regs->read向量, 利用lastInsPointer数组计算当前指令的被读寄存器的依赖距离，即对任意的r属于regs->read, 其依赖距离 = 当前PC值 - lastInsPointer[r].

for (vector<reg\_t>::iterator it = regs->read.begin(); it != regs->read.end(); it++)

{

reg\_t reg = \*it;

被读寄存器需要先被写，不然出错

if (lastInsPointer[reg] > 0)

{

依赖距离为当前PC值减去先前被写时的PC值

INT32 distance = insPointer - lastInsPointer[reg]; // TODO

深度数组长度是maxSize

if (distance <= maxSize)

insDependDistance[distance - 1]++; // TODO

}

}

对每条指令的插桩函数部分：只展示修改部分

找到所有被读寄存器

for (uint32\_t ir = 0; ir < INS\_MaxNumRRegs(ins); ir++)

{

// 获取当前指令中被读的寄存器(即目的寄存器)

REG rr = INS\_RegR(ins, ir);

// 获取寄存器名

rr = REG\_FullRegName(rr);

if (!REG\_valid(rr))

continue;

// 将被写寄存器保存到regs向量当中

if (std::find(regs->read.begin(), regs->read.end(), rr) == regs->read.end())

regs->read.push\_back(rr);

}

改进后的轨迹级insDependDist：

updateInsDependDistance函数不变，将INS\_AddInstrumentFunction(Instruction, 0)指令级插桩调用Instruction函数改为TRACE\_AddInstrumentFunction(Trace, 0)轨迹级插桩并调用Trace函数。

Trace函数：

VOID Trace(TRACE trace, VOID \*v)

{

遍历每个trace块

for (BBL bbl = TRACE\_BblHead(trace); BBL\_Valid(bbl); bbl = BBL\_Next(bbl))

{

遍历trace块内的每个指令

for (INS ins = BBL\_InsHead(bbl); INS\_Valid(ins); ins = INS\_Next(ins))

{

保存写寄存器到regs中(未修改)

Registers\* regs = new Registers();

for (uint32\_t iw = 0; iw < INS\_MaxNumWRegs(ins); iw++)

{

REG wr = INS\_RegW(ins, iw);

wr = REG\_FullRegName(wr);

if (!REG\_valid(wr))

continue;

if (std::find(regs->write.begin(), regs->write.end(), wr) == regs->write.end())

regs->write.push\_back(wr);

}

保存读寄存器到regs中(未修改)

for (uint32\_t ir = 0; ir < INS\_MaxNumRRegs(ins); ir++)

{

REG rr = INS\_RegR(ins, ir);

rr = REG\_FullRegName(rr);

if (!REG\_valid(rr))

continue;

if (std::find(regs->read.begin(), regs->read.end(), rr) == regs->read.end())

regs->read.push\_back(rr);

}

调用updateInsDependDistance函数，传入regs

INS\_InsertCall(ins,IPOINT\_BEFORE, (AFUNPTR)updateInsDependDistance, IARG\_PTR, (void\*)regs, IARG\_END);

}

}

}

## 2.2 实验结果及分析

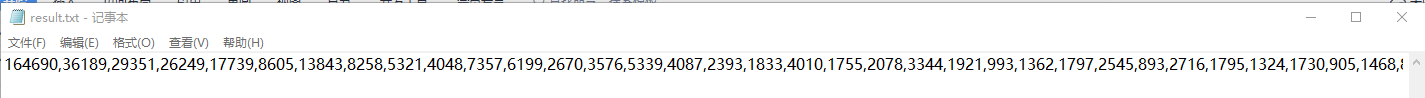
*描述实验过程及得出的实验结果，并进行合理分析。*

*如果进行了优化，需给出优化前后的结果对比截图及文字分析（如果优化没有效果，也需分析原因）。*

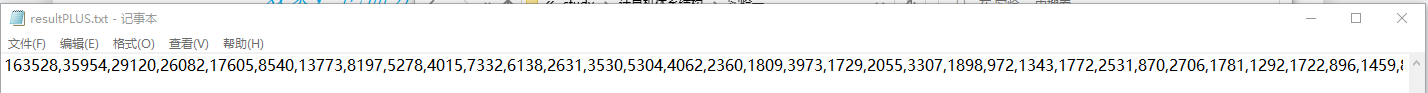
insDependDist函数利用python脚本paintInsDependDist.py输出依赖距离分布图

修改前和修改后依赖距离结果几乎一样，依赖距离分布图也几乎一样

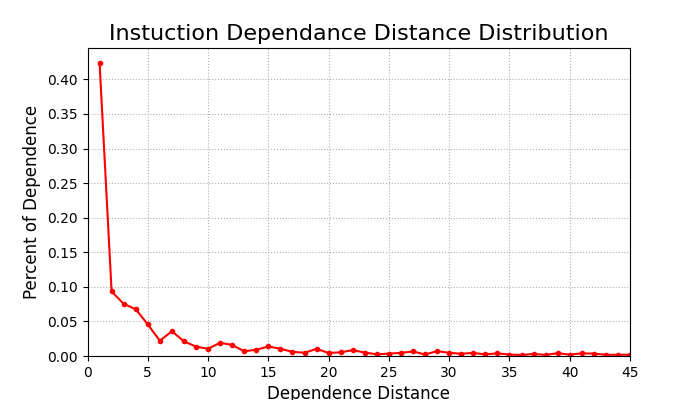
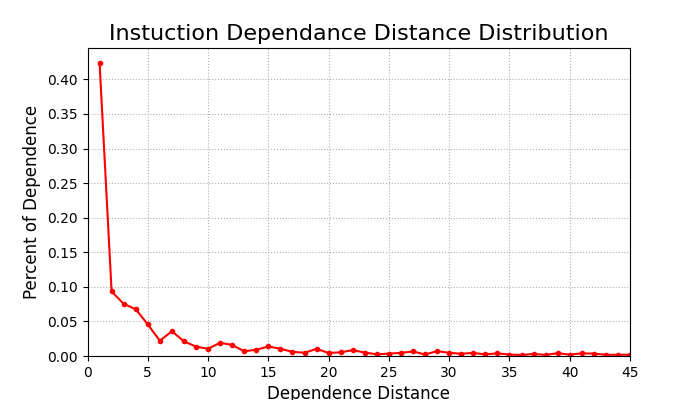
修改前依赖距离结果：



修改后依赖距离结果



左边是修改前依赖距离分布图，右边是修改后依赖距离分布图。



依赖距离分布图的描述与分析：

描述：依赖距离分布图较小的依赖距离(0-5)几乎占据了依赖距离的百分之80，小的依赖距离(0-10)占据了依赖距离的百分之90，依赖距离为1甚至占据了依赖距离的百分之40，依赖距离的数量随着依赖距离的增加整体大幅下降，到依赖距离增加到10之后依赖距离数量已经十分少了。

分析：依赖分布图表明指令中相同寄存器的读和写大部分距离较近，也有极少寄存器的读和写距离较远，寄存器读写距离从小到大数量逐渐减少。

这两个结果相近表明优化后能够正确输出结果，对于优化效果来说，优化后的插销将插销方法从指令级变为轨迹级，虽然分析代码依然需要对每一个指令进行遍历和分析，但基于轨迹级的插销方法对于插销函数调用TRACE\_AddInstrumentFunction相比于指令级插销方法大幅度减少，可以有效减少分析代码的执行次数(每条指令执行前分析修改为每个trace分支执行前分析)

插销时间的计算：

加入#include <sys/time.h>头函数

定义UINT64 usedTime = 0全局变量

定义struct timeval startTime全局结构体

定义struct timeval endTime全局结构体

指令级插销：

在Instrucion函数开头加入取得当前系统运行用时的函数赋给startTime

在Instrucion函数结尾加入取得当前系统运行用时的函数赋给endTime

轨迹级插销：

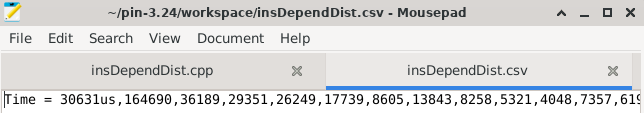
在Trace函数开头加入取得当前系统运行用时的函数赋给startTime

在Trace函数结尾加入取得当前系统运行用时的函数赋给endTime

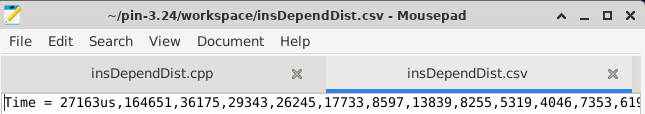
将两个时间的差值加到usedTime变量，最后输出usedTime变量

usedTime = usedTime + (endTime.tv\_sec\*1000000 + endTime.tv\_usec) - (startTime.tv\_sec\*1000000 + startTime.tv\_usec);

优化前分析代码执行时间30631us



优化后分析代码执行时间27163us



优化了插销性能，减少插销需要的时间

# 3. 回答问题

## 3.1 可以发现，指令依赖距离分布图存在很长的尾巴。试分析尾巴可能是由哪些寄存器引起的？为什么？

栈指针寄存器sp以及被用来传递函数参数的寄存器等等，sp寄存器在程序开始前进行内存空间分配(寄存器写)，很可能在之后的许多个指令对内存数据进行读取(寄存器读)，但只有程序结束后释放内存空间(寄存器写)，因此造成了许多长的依赖距离(占比较少，依赖距离较长，因此叫做尾巴)。被用来传递函数参数的寄存器在函数调用前需要保存传递的参数(寄存器写)，之后很可能在函数调用很久之后才使用该参数(寄存器读)造成了长的依赖距离。

## 3.2 设有2个不同架构的处理器平台，现分别于其上运行由相同版本编译器所编译的测试程序，并分别在这两个平台上使用insDependDist工具对该测试程序进行插桩分析，得到如图6-1所示的指令依赖距离分布图。请问架构A、B之中谁具有更多的寄存器？为什么？

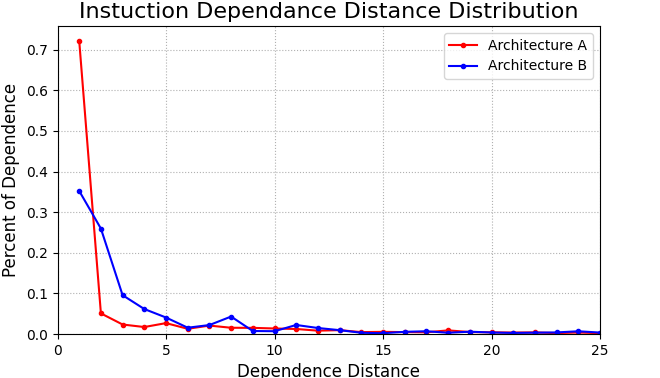


图6-1 2种架构下，同一程序的指令依赖距离分布图

架构B拥有更多的寄存器，因为拥有更多寄存器的架构，可以更不频繁读取被写过的寄存器，而拥有更少寄存器的架构在必要时不得不对寄存器进行多次读和写来得到需要的结果(短依赖距离较多)，因此B的短距离依赖占比更少，寄存器更多

## 3.3 现有基于相同ISA（Instruction Set Architecture）设计的架构A和架构B。若架构A采用停顿法解决流水线数据冲突，架构B则采用数据转发法，当二者执行相同的测试程序时，它们的指令依赖距离分布图是否相同？为什么？

相同，不管是停顿法还是数据转发法，都不会改变某个指令中使用过的寄存器的依赖距离，停顿法让流水线暂停一个时钟周期而不改变根据指令顺序和基于PC值计算出来的依赖距离，数据转发法则是将计算好的数据发往需要提前使用的寄存器位置，也不改变根据指令顺序和PC值计算出来的依赖距离。