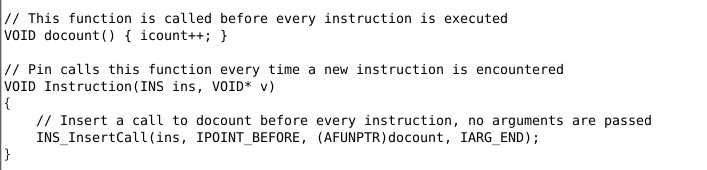
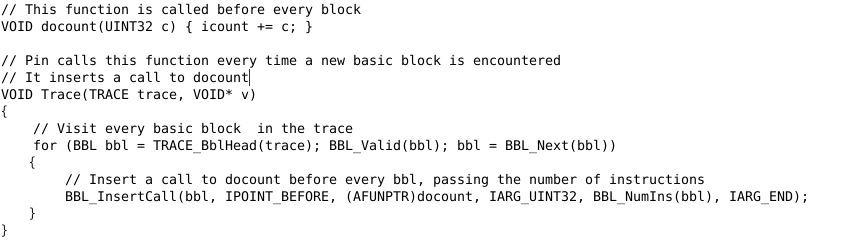
Pin入门及指令依赖距离分析

一、源码分析：

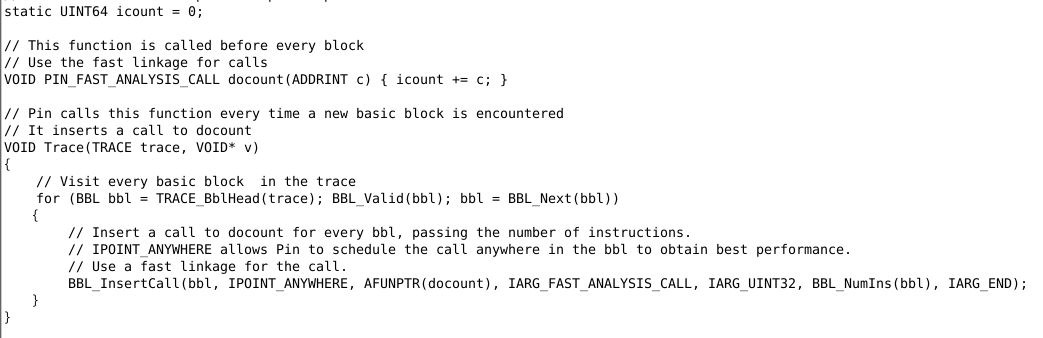
Inscount0如下，是指令级插桩，docount是分析代码，Instruction是插桩代码，茶插桩代码会在执行程序第一条指令后的代码当中动态写入分析代码，对于本例来说就是每执行一条指令将icount加一，可以统计可执行文件的指令数目：



Inscount1如下，是轨迹级插桩，每一个BBL块执行一次分析代码，更加高效，分析代码每一个块增加一次指令总数：

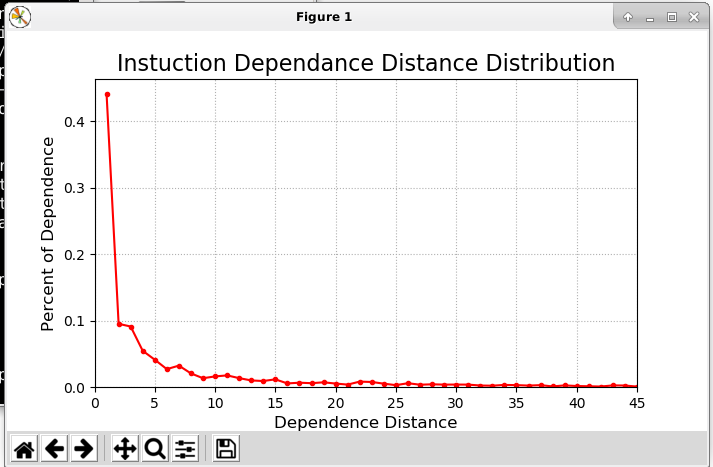


Inscount2如下，也是轨迹级插桩，插入的时候选择最小的寄存器存储恢复代价的位置：



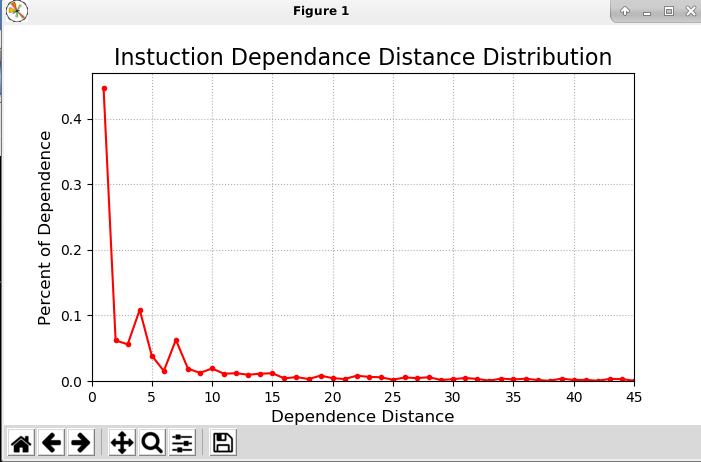
二、指令依赖距离分析

Ls程序插桩之后的指令依赖距离分析统计图如下，可以看到，依赖距离为1的比例很大，依赖大于10的很少发生，说明寄存器在载入数据之后很快就会进行读取运算，也说明可执行程序当中大部分都是运算指令，即将数据读入寄存器并立即进行计算，函数调用等指令比例较小：

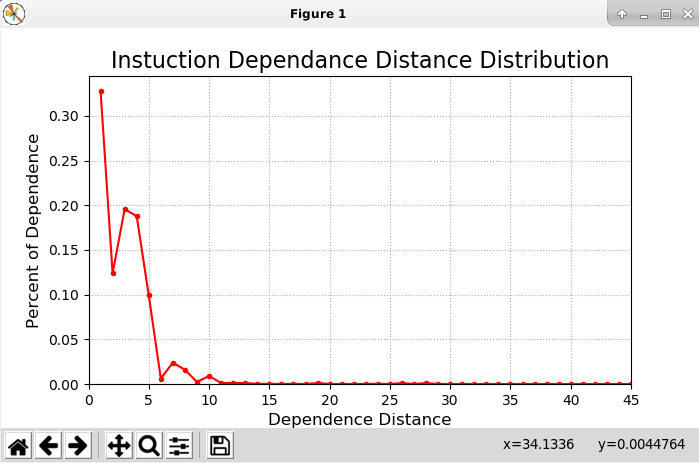


（ls的指令依赖距离分析统计图）

Pi的指令依赖距离分析统计图和ls的类似，只是在距离在5左右波动比较大，因为Pi程序中有浮点运算，其运算周期较长，指令依赖距离在5附近产生了两个峰：



（Pi测试程序的指令依赖距离分析统计图）



（memtester测试程序的指令依赖距离分析统计图）

三、回答问题

A、rsp、rbp这些栈指针寄存器，当产生函数调用时，这两个寄存器都会保存之前的运行时栈顶，并更新为当前栈顶，函数调用返回时恢复，中间函数体如果没有新的函数调用就不会涉及到这两个寄存器，因此他们的指令依赖距离应该比较大。

B、架构B有更多的寄存器，根据指令依赖距离分析统计图来看，架构B的指令依赖距离分析统计图更加平滑，指令之间的依赖距离整体来看更大，假设有相同个数的计算单元，拥有更多寄存器的架构自然可以对指令数据进行提前加载到寄存器当中，等待计算单元进行计算，因此寄存器更多的架构指令依赖距离就会更大一些。

C、不相同，架构A采用停顿法，在产生式数据相关的指令之间插入停顿，加大了二者之间的指令依赖距离，而数据转发可以在两个指令运行的过程中进行数据传递，从指令层面来看两个指令之间几乎没有增加新的指令，其指令依赖距离几乎没有增大，因此架构A的指令依赖距离分布图应该较为平缓，架构B的指令依赖距离分布图，距离较小的所占比例更大。