

Assigned Mar.21,2018

<http://cs.hit.edu.cn>

Summary

在本学期的计算机体系结构实验中，将使用由英特尔提供的二进制指令分析工具 Pin。本实验的目的是让你熟悉如何编译和运行 Pin 工具，以及确保你能够在装有 Pin 的计算机上使用 Pin 的基础架构。为了测试这个基础架构，你将构建并运行两个小型 Pin 工具。

为了开发更高效运行的程序的架构，架构师们必须清楚地了解具有代表性的程序的特性。他们通常尝试使用不同的程序来帮助设计将要包含在处理器中的功能。有几种方法可以用来评估程序的行为。一个是模拟。在这种情况下，可以建立处理器的软件模型，并在模型上执行程序。模拟器具有任意细节的优势：理论上可以构建 SPICE（仿真电路级）处理器模拟器。通常，架构模拟器会提供精确到周期的时序估计。这种细节级别的代价是仿真速度：软件仿真器越详细，执行程序越慢。先进的软件模拟器可以以几十 KIPS（千指令每秒）的速率进行模拟。另一种方案是代码工具。代码工具收集有关程序特征的信息。一般而言，代码工具不如仿真更详细，但其实现起来更快，并且拥有更快的程序执行速度。因此，在更详细的模拟器可以使用之前，代码工具可以在早期开发过程的指导架构过程中起到很大作用。几乎所有程序员、架构师或其他人都会使用到最简单的代码工具——终端显示语句（例如 printf）。然而，这样的手动指令在编写代码和收集执行结果方面都很耗时。一个更好的选择是使用元语言来描述代码工具并开发一个能在编译或运行时有效地测试目标程序的工具。

Pin 是英特尔公司生产的免费（但未开源的）工业级二进制指令分析工具，广泛用于工业和学术界。Pin 接受编译好的 Pintool 和一个待分析的可执行文件作为输入。Pintool 是一个 C++ 程序，它对 Pin 的 API 进行一系列调用，在可执行程序的任何地方插入你所编写的代码（用 C 或 C++ 编写）。由于 Pin 的特点，我们也可以将其看作一款 JIT 编译器——Pin 将代码在程序运行的时候动态添加到该程序中（修改内存映像）。同样，这也使得 Pin 可以附加到进程上。Pin 的主要优点是它可以在不需要重新编译的情况下对程序进行测试。由于 Pin 在本地执行大部分目标程序，因此它可以非常快速。

你可以在本次实验附带的资源列表中找到 Pin 的全部内容。这个实验需要你单独完成，你可以通过阅读附带的 Pin 指导内容来学习使用，或与同学讨论你遇到的问题。

Setting Up

1.操作系统配置：如果你使用的是 Windows 操作系统的计算机，建议使用 Linux 系统搭配虚拟机（VMware 等）使用。

使用 Mac OS 系统的同学应该可以正常使用 Pin，但我们并未对此平台下的安装及使用做过测试。

Pin 工具及 Linux 自身运行不需要太强的性能，为虚拟机分配 1~2GB 的内存空间、1~2 个处理器核即可。



图 1.开发及测试用机

2. 编译器配置：由于 CentOS 7 系统附带 GCC 4.9/4.8 版本，在使用过程中会出现无法启用 C++11 新特性的情况，因此，我们建议将编译器更新至 GCC 6.2.0。Ubuntu 系统应该附带有此版本，无需更新。

请使用 `gcc -version` 命令查看你当前安装的 GCC 版本，如果不符合要求，请尽量更新至 6.2 或其相近版本，以免使用过程中出现问题。你也可以使用 4.8 版本正常编译大部分 Pin 程序，但由于程序中有很多需要你自已完成的地方，因为你个人的 C++ 使用习惯不同，可能会出现一些编译器无法支持的错误。这不代表你的语法是错误的，仅仅是这样写在 Pin 工具中无法使用而已。

我们在本文最后 Addition 部分附带有 GCC 6.2.0 的详细安装步骤，你也可以另外寻找适合你自己的安装方法。本文所测试过的内容仅限 CentOS 系统。

编译安装 GCC 6.2 需要 1.5~2 小时左右，请注意合理规划你的时间。以免错过提交。

3.Pin 安装与使用

从本次实验的资源列表里即可找到 Pin 3.5，解压后进入目录 `pin-3.5-XXXXX/source/tools/ManualExamples` 中，在此处你可以新建 `pintool`，或者将实验模板 `labX.cpp` 放置进来并进行编辑。

需要注意的是，当你完成一个 pintool 并打算编译它的时候，你需要将你本次要用到的 .cpp 或 .h 文件（如果你的 Pin 工具使引用了头文件，仅放置你自己编写的 .h 头文件，不需要放 C++ 库函数头文件）放在这个目录下，才能确保要用到的 .h 头文件能被它的 .cpp 文件正确引用到，然后打开 makefile.rules 文件，在第一项 TEST_TOOL_ROOTS 后加入你要编译的 .cpp 文件名，如 lab0_1 lab0_2，多个 pintool 之间空格隔开，不需要输入后缀。如图所示：

```
TEST_TOOL_ROOTS := inscount0 inscount1 inscount2 proccount imageload staticcount detach malloctrace \
                  malloc_mt inscount_tls stack-debugger pinatrace itrace isampling safecopy invocation countreps \
                  nonstatica test0 test1
```

此后所有用到 Pin 的实验均需如此操作。加入到列表后再运行 make，才能在本目录下的 obj-intel64 文件夹中看到以你 pintool 命名的 .so 文件（如 lab0_1.so，lab0_2.so），这就是你本次编译生成的可运行的 pintool。

为了你可以在任何地方使用 pin，你需要在 pin-3.5-XXXXXX 找到可执行文件 pin，并将它所在的目录放入系统路径 PATH 中，即在 ~/.bashrc 中添加：

```
export PATH=$PATH:/mnt/USER/software/pin-3.4-XXXXXX
```

在这之后，你就可以在任何地方运行 pintool 进行程序的一次插桩了，指令格式：

```
pin -t XXXX.so -- YYYY
```

XXXX.so 为 pintool，YYYY 为输入的待分析二进制程序，均需指明路径。如果报错的话，请检查你输入的两个文件的路径是否正确，尤其是当你使用相对路径的时候。

4.注意：请不要在 Windows 环境下编写好程序后放入 Linux 编译执行，可能会因为编码问题发生意想不到的错误！

Example 1 指令计数

在本实验的第一部分中，你将开发一个简单的 Pin 工具来计算应用程序执行的指令数量。你可以根据实验模板里的注释以及提供的文档来初步学习与体会一个 Pintool 的基本结构，并尝试补全这个 Pintool。与你学习 C 语言时编写的 HelloWorld 一样，指令计数程序将是你的 Pin 的第一堂课。

实验模板中缺少几处重要的核心代码，需要你来完成。直接编译运行这个模板将会出错。在后面的实验中也同样需要你来完成一些留空的核心代码，但会更加注重于体系结构的知识，而非过多关注 Pin 的相关内容。

实验要求：

补全实验模板中为你留空的内容，使得 pintool（即 .cpp 文件）可以被正常编译运行，并可以对一个你指定的二进制可执行程序进行指令计数，统计出这个程序执行期间一共执行的指令条数（此处均指汇编指令，instruction），并输出至文件。

实验输入：

一个任意的二进制可执行程序

实验输出：

输出至文件（文件读写部分已为你完成，计数结果输出至同目录下 lab0_1.out，可直接用 cat lab0_1.out 指令查看）。

示例：`|Count 60252493` （代表被测程序一共执行了 602 万多条指令）

Example 2 指令依赖距离

在第二部分，我们要求开发出一款可以分析出可执行程序中的指令依赖距离的 Pintool。指令依赖距离的定义如下：一条指令向一个寄存器写入数值之后可能会被后续的另一条指令在所读取出来，这两条指令相隔的指令数就被定义为指令的依赖距离。

例子如下：在下图 3-1 中我们可以看到因为寄存器 r1，第二条指令和第四条指令都依赖于第一条指令，依赖距离分别是 1 和 3。

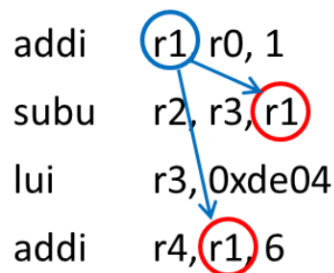


图 4-1 指令依赖距离说明实例

实验要求：

你所编写的指令依赖距离分析工具要对所有指令之间的依赖关系进行分析，记录下它们之间的依赖距离，将结果按照依赖距离分类统计，最终要记录下每种大小的依赖距离分别有多少条指令、占比有多大，然后按照比重从高到低对各依赖距离的统计结果输出至文件。（指令依赖距离若大于 30，均按 31 统计）

实验输入：

一个任意的二进制可执行程序

实验输出：

输出至 lab0_2.out 文件，每行一条，格式为：

[依赖距离]X->[此依赖距离指令条数]Y->[占比]Z%

```
依赖距离1 -> 27434225 -> 43.151081%
依赖距离2 -> 3362716 -> 5.289190%
依赖距离3 -> 2725251 -> 4.286526%
依赖距离4 -> 6036694 -> 9.495070%
依赖距离5 -> 512870 -> 0.806689%
依赖距离6 -> 511994 -> 0.805311%
依赖距离7 -> 565756 -> 0.889873%
依赖距离8 -> 689176 -> 1.084000%
依赖距离9 -> 396802 -> 0.624127%
依赖距离10 -> 1068228 -> 1.680208%
依赖距离11 -> 433801 -> 0.682322%
依赖距离12 -> 608426 -> 0.956989%
依赖距离13 -> 372664 -> 0.586160%
依赖距离14 -> 73415 -> 0.115474%
依赖距离15 -> 424225 -> 0.667260%
依赖距离16 -> 218412 -> 0.343539%
依赖距离17 -> 212917 -> 0.334896%
依赖距离18 -> 219061 -> 0.344559%
依赖距离19 -> 416124 -> 0.654518%
依赖距离20 -> 213413 -> 0.335676%
依赖距离21 -> 15275 -> 0.024026%
依赖距离22 -> 16249 -> 0.025558%
示例: 依赖距离23 -> 207591 -> 0.326518%
```

Submit

供你测试及入门使用，不需要提交代码和实验报告。

Addition

1.GCC 6.2 安装

(1) 下载 gcc-6.2.0 源码包 (<https://bigsearcher.com/mirrors/gcc/releases/gcc-6.2.0/>) 随后解压：(本文将 software 视为下载目录)

```
[root@node1 software]# tar -zxvf gcc-6.2.0.tar.gz
```

(可使用命令 `vim contrib/download_prerequisites` 查看所需要的依赖包版本是否与下述的依赖包版本相符)

(2) 下载依赖包：

```
[root@node1 software]# wget ftp://gcc.gnu.org/pub/gcc/infrastructure/mpfr-2.4.2.tar.bz2
```

```
[root@node1 software]# wget ftp://gcc.gnu.org/pub/gcc/infrastructure/gmp-4.3.2.tar.bz2
```

```
[root@node1 software]# wget ftp://gcc.gnu.org/pub/gcc/infrastructure/mpc-0.8.1.tar.gz
```

(3) 安装 GMP

```
[root@node1 software]# bzip2 -d gmp-4.3.2.tar.bz2
```

```
[root@node1 software]# tar xvf gmp-4.3.2.tar
```

```
[root@node1 software]# cd gmp-4.3.2/
```

```
[root@node1 gmp-4.3.2]# ./configure --prefix=/opt/gcc-6.2.0
```

```
[root@node1 gmp-4.3.2]# make -j
```

```
[root@node1 gmp-4.3.2]# sudo make install //此处必须 sudo 提权，否则会 make 失败
```

(4) 安装 MPFR

```
[root@node1 gmp-4.3.2]# cd ..
```

```
[root@node1 software]# bzip2 -d mpfr-2.4.2.tar.bz2
```

```
[root@node1 software]# tar xvf mpfr-2.4.2.tar
```

```
[root@node1 software]# cd mpfr-2.4.2/
```

```
[root@node1 mpfr-2.4.2]# ./configure --prefix=/opt/gcc-6.2.0
```

```
[root@node1 mpfr-2.4.2]# ./configure --prefix=/opt/gcc-6.2.0 --with-gmp=/opt/gcc-6.2.0
```

```
[root@node1 mpfr-2.4.2]# make
```

```
[root@node1 mpfr-2.4.2]# sudo make install
```

(5) 安装 MPC

```
[root@node1 mpfr-2.4.2]# cd ..
```

```
[root@node1 software]# tar xvf mpc-0.8.1.tar.gz
```

```
[root@node1 software]# cd mpc-0.8.1/
```

```
[root@node1 mpc-0.8.1]# ./configure --prefix=/opt/gcc-6.2.0 --with-gmp=/opt/gcc-6.2.0 --with-mpfr=/opt/gcc-6.2.0
```

```
[root@node1 mpc-0.8.1]# make -j
```

```
[root@node1 mpc-0.8.1]# sudo make install
```

(6) 安装 gcc

```
[root@node1 mpc-0.8.1]# cd ..
```

```
[root@node1 software]# cd gcc-6.2.0/
```

```
[root@node1 gcc-6.2.0]# mkdir build
```

```
[root@node1 gcc-6.2.0]# cd build/
```

```
[root@node1 build]# ../configure --prefix=/opt/gcc-6.2.0 --with-gmp=/opt/gcc-6.2.0 --with-mpfr=/opt/gcc-6.2.0 --with-mpc=/opt/gcc-6.2.0 -enable-checking=release -enable-languages=c,c++ -disable-multilib
```

```
[root@node1 gcc-6.2.0]# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/gcc-6.2.0/lib
```

```
[root@node1 build]# make
```

```
[root@node1 build]# sudo make install
```

(7) 设置环境变量

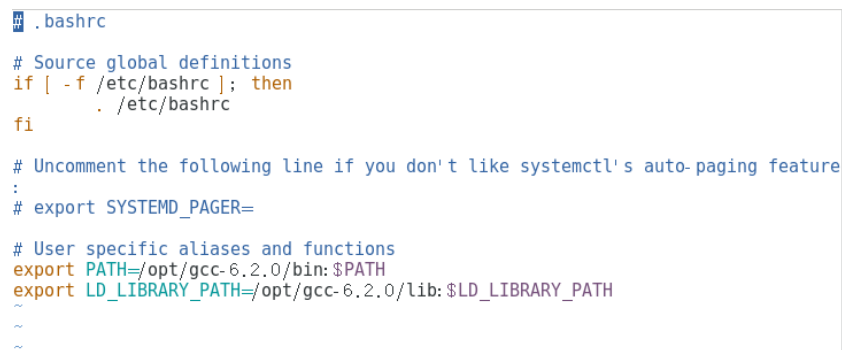
```
[root@node1 software]# export PATH=/opt/gcc-6.2.0/bin:$PATH
```

```
[root@node1 software]# export LD_LIBRARY_PATH=/opt/gcc-6.2.0/lib:$LD_LIBRARY_PATH
```

为使这项改动永久生效，还需要将这两行命令添加至.bashrc 文件中：

```
[root@node1 software]# sudo vim ~/.bashrc
```

改动如图：



```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# Uncomment the following line if you don't like systemctl's auto-paging feature
:
# export SYSTEMD_PAGER=

# User specific aliases and functions
export PATH=/opt/gcc-6.2.0/bin:$PATH
export LD_LIBRARY_PATH=/opt/gcc-6.2.0/lib:$LD_LIBRARY_PATH
~
~
```

关闭 shell 终端后重新打开生效。

(8) 查看 gcc 版本

```
[root@node1 ~]# gcc -v
```

```
使用内建 specs。
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/opt/gcc-6.2.0/libexec/gcc/x86_64-pc-linux-gnu/6.2.0/lto-wrapper
目标：x86_64-pc-linux-gnu
配置为：../configure --prefix=/opt/gcc-6.2.0 --with-gmp=/opt/gcc-6.2.0 --with-mpfr=/opt/gcc-6.2.0 --with-mpc=/opt/gcc-6.2.0 --enable-checking=release --enable-languages=c,c++ --disable-multilib
线程模型：posix
gcc 版本 6.2.0 (GCC)
```

(9)你可能在以后的使用中发现如下报错：

```
"/usr/lib/libstdc++.so.6: version `CXXABI_1.3.9' not found"
```

这是由于更新了新版 GCC 编译器却没有将一个动态链接库更新的原因，手动更新步骤详见：

见：<https://blog.csdn.net/zx714311728/article/details/69628836>。请注意！版本号及系统

位数稍有区别，请按照你安装的版本修改。如：gcc-6.2.0 /usr/lib64/

2.Pin 开发

Pin 工具又称 Pintool，Pintool 是开发人员使用 C/C++ 结合 Pin 提供的 API 所开发出的系列程序插桩工具，下面给出 ManualExamples 中给出的一些 Pintool 的用途：

- imagerload 跟踪程序上传和下载 imagei
- itrace 跟踪记录每个指令执行的内存地址
- malloctrace 记录函数参数传递到函数的值或返回的值
- pinatrace 检测指令读取和写入的内存地址
- proccount 记录程序被调用的次数

Pintool 编写是比较容易的，只需要明白 Pintool 的工作流程就可以了。

Pintool 由 `int main(int argc, char * argv[])` 函数开始，然后初始化 `PIN_Init(argc, argv)`，初始化之后便可以编写相关工作的代码了。

PIN tool 分为指令级插桩、轨迹级插桩、镜像级插桩以及函数级的插桩。

指令级插桩 (instruction instrumentation)，通过函数 `INS_AddInstrumentFunction` 实现。

轨迹级插桩 (trace instrumentation)，通过函数 `TRACE_AddInstrumentFunction` 实现。

镜像级插桩 (image instrumentation)，使用 `IMG_AddInstrumentFunction` 函数，由于其依赖于符号信息去确定函数边界，因此必须在调用 `PIN_Init` 之前调用 `PIN_InitSymbols`。

函数级插桩 (routine instrumentation)，使用 `RTN_AddInstrumentFunction` 函数。函数级插桩比镜像级插桩更有效，因为只有镜像中的一小部分函数被执行。

四种插装粒度，其中，`IMG_AddInstrumentFunction` 和 `RTN_AddInstrumentFunction` 需要先调用 `PIN_InitSymbols()`，来分析出符号。在无符号的程序中，`IMG_AddInstrumentFunction` 和 `RTN_AddInstrumentFunction` 无法分析出相应的需要插装的块。

在各种粒度的插装函数调用时，可以添加自己的处理函数在代码中，程序被加载后，在被插装的代码运行时，自己添加的函数会被调用。

`INS_AddInstrumentFunction`、`TRACE_AddInstrumentFunction`、`IMG_AddInstrumentFunction`、`RTN_AddInstrumentFunction` 指定的回调函数只有在相应的代码被分析到时才会被调用，即分析到一次只被调用一次，但程序运行过程中一般不再被调用，但 `INS_InsertCall` 之

类的程序添加的函数,是在相应的代码位置添加函数,根据程序运行的情况,会被多次调用。

在 INS_AddInstrumentFunction 指令级插桩的代码中,只有在 INS_AddInstrumentFunction 指定的函数被调用时 INS 指令才有效,在 INS_InsertCall 函数中,INS 无效。

下面以最简单的 Pintool 工具 inscount0 为例对 pintool 代码进行说明:

了解 pintool 的自定义函数是看懂其代码的关键,如下代码所示, pintool 的主函数可分为三个部分,输出文件建立部分、程序检测函数部分和结尾部分。

```
int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

输出文件建立部分确定输出文件的格式和说明内容,程序检测函数部分调用各函数对可执行程序进行检测,结尾部分一般为其自定义的 Fini 函数,用来做将检测数据写入输出文件、关闭文件等的收尾工作。显然,第二部分是最重要的一部分,从前面声明的函数可知 INS_AddInstrumentFunction 函数调用 Instruction 函数,Instruction 函数的参数为要监测的指令,它拦截每一条执行中的指令,并执行 InsertCall 中指定要插入的函数内容。了解这些规律就很容易看懂这些代码。所以,需要多阅读 Intel 提供的 Pintool 的源码加深对 Pintool 工作过程的理解。

Instruction 代码如下所示:

```
// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to do before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)do, IARG_END);
}
```

do 函数如下所示：

```
// This function is called before every instruction is executed  
VOID do () { /*      your code      */; }
```

更加详细的资料请查阅 Pin 的 API 文档：

<https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html/modules.html>

API 文档中的 INS:Instruction Object 一栏下的 API 是本次实验主要使用的 API。