

Computer Organization Homework 3 Report

This report will show the result of the Intel Pin tool used, specifically kit 98484. This report will be divided into Setup, Task1, Task2 & Task3. For benchmarking, Fibonacci is used, both iterative and Recursive. The operating system used is Linux, specifically Ubuntu version 18.04.

Setup

After downloading the PIN tool, the first step is to set the Intel Pin tool profile, which can be done with the code below:

```
Set                                     INTEL_JIT_PROFILER
/home/Desktop/pin-3.20-98437-gcc-linux/intel64/lib/libpinjitprofil
ing.so
```

After this, we can then build the files and execute the code using pin with the code below

```
../../../../pin -t obj-intel64/sy.so -- /home/hasan/Desktop/fibonacci_recursive.out
```

Or

```
../../../../pin-t                        obj-intel64/sy.so          -
/home/hasan/Desktop/fibonacci_iterative.out
```

From here we can then execute the code

Task 1

The aim of the first task is to find the total of both microinstruction and macroinstruction of the application. To achieve this, we used the code below:

```
VOID findtotalcount() {
    totalBranchCounter++;
}
```

Where it increments the counter for every instruction. We will then call this function through the insert call from the function below.

```
INS_InsertCall(ins,      IPOINT_BEFORE,      (AFUNPTR)      findtotalcount,
IARG_END);
```

Task 2

The aim of the second task is similar to that of the first one, however more specific. There are three types of instruction to count, these being: Memory Instruction, Branch Instruction and arithmetic instruction. To achieve this, the variables are at first defined, as shown with the code below

```
static UINT64 memoryInstructionCounter = 0;
static UINT64 totalBranchCounter = 0;
static UINT64 branchInstructionCount = 0;
static UINT64 arithmeticIntructionCount = 0;
```

After all the variable is defined, we can then implement the various function used to increment when each instruction is found.

```
VOID findBranchCount() { branchInstructionCount++; }
VOID findMemoryCount() { memoryInstructionCounter++; }
VOID findtotalInstructionCount() { totalBranchCounter++; }
```

Next, we can then implement an IF ELSE statement is used to determine the type of instruction. This is shown below with the code

```
if (INS_IsBranch(ins))
{
    INS_InsertCall(ins, IPOINT_BEFORE,
(AFUNPTR)findBranchCount, IARG_END);
}
for (UINT32 memOp = 0; memOp < memOperands; memOp++)
{
    If(INS_MemoryOperandIsWritten(ins,memOp)||INS_MemoryOperandIs
Read(ins, memOp))
    {
        INS_InsertCall(ins, IPOINT_BEFORE,
(AFUNPTR)findMemoryCount, IARG_END);
    }
}
```

Firstly it checks whether the code is a branch instruction and if yes, it calls the findBranchCount defined earlier to increment. After that, a for loop is called and checks if an instruction is read or written, in order to determine whether an instruction is a Memory instruction, which will then count the memory instruction count. After all the different numbers of instructions are accounted for, we can then write the result to the output file with the code below.

```
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
"o", "results.out", "specify output
file name");
```

In this case, the name of the file is results. We can then print the results to the output file with the code shown below. Since we have Memory and branch code, we can simply derive the number of arithmetic by subtracting both Memory + branch from the total number of codes.

```
OutFile << "\nResult of Task 2: " << endl;
    OutFile << "The number of Branch Instruction in the code is: "
<< branchInstructionCount << endl;
    OutFile << "The number of Memory Instructions in the code is:
" << memoryInstructionCounter << endl;
    OutFile << "The number of Arithmetic and Logic Instructions in
the code is: " << totalBranchCounter - (branchInstructionCount +
memoryInstructionCounter) << endl;
```

After showing the results we can also show the ratio of each instruction writing the result in the output file as shown below.

```
double  ratioBranchInstructions  =  (double)branchInstructionCount
/(double)totalBranchCounter;
double  ratioMemoryInstructions  =  (double)memoryInstructionCounter
/(double)totalBranchCounter;
Double          ratioArithmeticInstructions          =
(double)arithmeticIntructionCount /(double)totalBranchCounter;

OutFile    <<    "Ratio    of    branch    instructions:    "    <<
ratioBranchInstructions << endl;
    OutFile    <<    "Ratio    of    memory    instructions:    "    <<
ratioMemoryInstructions << endl;
    OutFile    <<    "Ratio    of    arithmetic    instructions:    "    <<
ratioArithmeticInstructions << endl;
```

Task 3

For Task 3, the methodology to retrieve the result is similar to that of Task 2, although a bit different. In this case, we need to determine the frequency of destination register that is used, therefore the focus is placed on the Write operation to register. At the beginning, we define the variables of the Register used, with the code shown below:

```
static UINT64 accumulatorRegCounter = 0;
static UINT64 baseIndexRegCounter = 0;
static UINT64 counterRegCounter = 0;
static UINT64 rdxRegCounter = 0;
static UINT64 rsiRegCounter = 0;
static UINT64 rdiRegCounter = 0;
```

```
static UINT64 rspRegCounter = 0;
static UINT64 rbpRegCounter = 0;
static UINT64 regularRegCounter = 0;
```

We can then include the function to increment the counter, as shown below

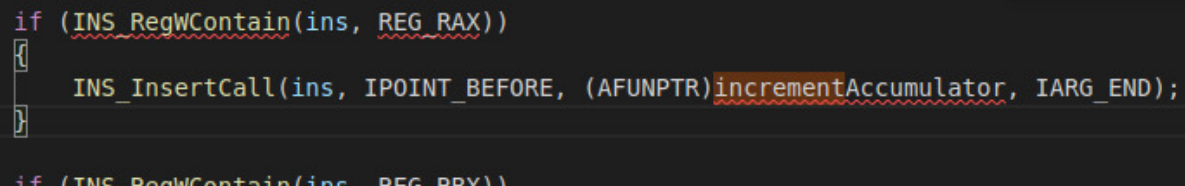
```
VOID incrementAccumulator() { accumulatorRegCounter++; }
VOID incrementBaseIndex() { baseIndexRegCounter++; }
VOID incrementCounter() { counterRegCounter++; }
VOID incrementRdx() { rdxRegCounter++; }
VOID incrementRsi() { rsiRegCounter++; }
VOID incrementRdi() { rdiRegCounter++; }
VOID incrementRsp() { rspRegCounter++; }
VOID incrementRbp() { rbpRegCounter++; }
VOID incrementRegular() { regularRegCounter++; }
```

From this, we can determine the frequency of register usage using the code below

```
BOOL INS_RegW Contain (const INS ins,const REG reg)
```

Where Reg reg, define the type of register used, for example, RDX register would be REG_RDX, in which another register is defined in the pin documentation: https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INS__INSP__ACTION.html#gab319902e019d907d773a1c6799d4ea32

From here, we can then create if statements to check the number of destination registers that are within an instruction with the code below:



```
if (INS_RegWContain(ins, REG_RAX))
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)incrementAccumulator, IARG_END);
}

if (INS_RegWContain(ins, REG_RBP))
```

We can then use the code above to check the frequencies on the various register used, by changing the attribute of the function. When it comes to the regular register, we used an if along with an OR, thus all the general register from R8 to R15 is accounted for. We can then write the result using the code shown below.

```
OutFile << "\nTask3\nRAX Count is : " << accumulatorRegCounter <<
" with a frequency of: " << (accumulatorRegCounter * 100) /
totalRegCount << endl;
    OutFile << "Base Index Count is : " << baseIndexRegCounter <<
" with a frequency of: " << (baseIndexRegCounter * 100) /
totalRegCount << endl;;
    OutFile << "RCX Count is: " << counterRegCounter << " with a
frequency of: " << (counterRegCounter * 100) / totalRegCount <<
endl;
```

```

    OutFile << "RDX Count is : " << rdxRegCounter << " with a
frequency of: " << (rdxRegCounter * 100) / totalRegCount << endl;
    OutFile << "RSI Count is : " << rsiRegCounter << " with a
frequency of: " << (rsiRegCounter * 100) / totalRegCount << endl;
    OutFile << "RDI Count is : " << rdiRegCounter << " with a
frequency of: " << (rdiRegCounter * 100) / totalRegCount << endl;
    OutFile << "RSP Count is : " << rspRegCounter << " with a
frequency of: " << (rspRegCounter * 100) / totalRegCount << endl;
    OutFile << "RBP Count is : " << rbpRegCounter << " with a
frequency of: " << (rbpRegCounter * 100) / totalRegCount << endl;
    OutFile << "general Purpose register Count is : " <<
regularRegCounter << " with a frequency of: " <<
(regularRegCounter * 100) / totalRegCount << endl;
    OutFile.close();

```

Results

The result of both iterative and recursive are displayed and analyzed below. For the screenshot, the n used is 9 for both recursive and iterative to make a fair comparison.

Fibonacci Iterative Results

The screenshot below displays the execution of the Fibonacci iterative function.

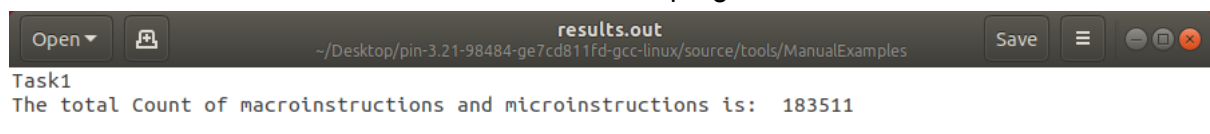
```

hasan@hasan-VirtualBox:~/Desktop/pin-3.21-98484-ge7cd811fd-gcc-linux/source/tools/ManualExamples$ ../../pin -t obj-intel64/sy.so -- /home/h
asan/Desktop/fibonacci_iterative.out
Enter the number of elements:
9
Fibonacci Series is:
34

```

Task 1

The aim of task 1 is to determine the total number of both micro and macroinstructions executed, shown below is the result of the PIN tool program.



```

Open  results.out  Save  ~/Desktop/pin-3.21-98484-ge7cd811fd-gcc-linux/source/tools/ManualExamples
Task1
The total Count of macroinstructions and microinstructions is: 183511

```

As shown in the screenshot above, there is 183511 amounts of microinstruction inside the Fibonacci iterative function. Task 2 will further analyze what these instructions are made up of.

Task 2

Task 2 results are shown below:

Task2

Result of the instructions are:

The number of Branch Instruction in the code is: 39500

The number of Memory Instructions in the code is: 54846

The number of Arithmetic and Logic Instructions in the code is: 89165

Ratio of branch instructions: 0.215246

Ratio of memory instructions: 0.29887

Ratio of arithmetic instructions: 0.485884

From this, we can analyze what made the microinstruction from task 1, as shown above, the majority of instructions are arithmetic instructions, with a ratio of 0.48 or 48 percent of the total instruction, this more or less make sense because there is a lot of calculation done on the iteration. The second most instruction type is memory instructions with branch instruction being the least with a ratio of 0.21 or 21 percent.

Task 3

Task 3 dives deeper into the problem by analyzing the destination register used that are executed, the result of which is shown below:

Task3

RAX Count is : 12068 with a frequency of: 18.0613

Base Index Count is : 4144 with a frequency of: 6.20201

RCX Count is: 12428 with a frequency of: 18.6001

RDX Count is : 5202 with a frequency of: 7.78544

RSI Count is : 1916 with a frequency of: 2.86753

RDI Count is : 3710 with a frequency of: 5.55248

RSP Count is : 6720 with a frequency of: 10.0573

RBP Count is : 1223 with a frequency of: 1.83037

general Purpose register Count is : 19406 with a frequency of: 29.0435

From the result above, we can analyze the different registers used in the instruction. As shown above, one of the most common registers used is a general-purpose register or R8 to R15, which makes sense as arithmetic instructions are the most common. One important point to note is RCX count, as although it does not represent the biggest frequency, it is still among one of the highest ones, this can be attributed due to a lot of for loops being used, thus resulting in a lot of RCX instruction as it's purpose is to act as a counter for loops. Another analysis that can be made is that both RSI and RDI is not used that frequently, this can be attributed to the fact that there is not a lot of string operations being executed, to begin with in the code.

Fibonacci Recursive Result

```
hasan@hasan-VirtualBox:~/Desktop/pin-3.21-98484-ge7cd811fd-gcc-linux/source/tools/ManualExamples$ ../../pin -t obj-intel64/sy.so -- /home/h
asan/Desktop/fibonacci_recursive.out
Enter the number of elements:
9
```

Task 1

As mentioned before, the aim of task 1 is to determine the number of microinstructions used, the results for Fibonacci Recursive are shown below.

```
Open ▾ [icon] results.out
~/Desktop/pin-3.21-98484-ge7cd811fd-gcc-linux/source/tools/ManualExamples

Task1
The total Count of macroinstructions and microinstructions is: 185334

Task2
```

One surprising observation is that both recursive and iterative does not have a major difference when it comes to instruction usage.

Task 2

Result of Task 2 is shown below

```
Task2
Result of the istructions are:
The number of Branch Instruction in the code is: 39616
The number of Memory Instructions in the code is: 55858
The number of Arithmetic and Logic Instructions in the code is: 89860
Ratio of branch instructions: 0.213755
Ratio of memory instructions: 0.301391
Ratio of arithmetic instructions: 0.484854
```

As shown above, the result of Task 2 is more or less similar to that of iterative with the arithmetic instructions consisting of the majority while memory is second and branch is third.

Task 3

Task 3 results are shown below

```
Task3
RAX Count is : 12077 with a frequency of: 17.7569
Base Index Count is : 4266 with a frequency of: 6.27233
RCX Count is: 12426 with a frequency of: 18.27
RDX Count is : 5196 with a frequency of: 7.63972
RSI Count is : 1918 with a frequency of: 2.82005
RDI Count is : 3709 with a frequency of: 5.45337
RSP Count is : 7577 with a frequency of: 11.1405
RBP Count is : 1437 with a frequency of: 2.11283
general Purpose register Count is : 19407 with a frequency of: 28.5343
```

From the result, it can be seen that the recursive result used the same amount of registers as iterative. The general purpose frequency has the most, with RCX coming second, followed by all the other registers.

Analysis and Discussion

From the result above, we can see that both iterative and recursive have a similar number of instructions used, In our case, iterative have 183511 microinstructions used while recursive

have 185344. The composition of instructions is similar on both iterative and recursive, with arithmetic and logic being having the highest ratio and branch having the lowest for both. Moreover, the frequencies of register used is also similar among the two types of Fibonacci execution, with the General purpose register having the most, followed by RCX and RAX. One explanation for the high usage of RCX is due to the register acting as a counter to loops, which both iterative and recursive technically require. When testing with other n, the value of register does not change by a lot, however, one noticeable difference for both is that RSP count increases, this is due to more values needed to be stored in the stack, thus more movement for stack pointer.

References

https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INS__INSP_ECTION.html#gab319902e019d907d773a1c6799d4ea32

https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__REG.html#gga8f899d7ad1af070aae505a85cc998fa5a8cda1231308d448ec2f7a3c07f9454b9