MO601 - Project 1 Report

093311 - Alberto Arruda de Oliveira[1]

[1]Institute of Computing – University of Campinas (UNICAMP)

{alberto.oliveira}@ic.unicamp.br

Abstract. *The main objectives of this project are twofold: (1) learning to use the benchmark tool SPEC2006 and the instrumentation tool Pin, and (2) Employing Pin to instrument the code of different SPEC2006 benchmarks. A default Pin tool, used to count instructions, was used with every benchmark from SPEC2006, in addition to a custom Pin tool which was developed specifically for this project. Moreover, this custom Pin tool, used to count branches in code, was also applied to small C++ examples, in order to asses how the addition of code alters the count of instructions and number of branches.*

1. Introduction

This project explores two tools fundamental to evaluating computer systems: SPEC2006 [SPE] and Pin [Pin]. The first is a benchmarking tool, that is, it seeks to evaluate different computer systems, according to performance metrics, by using standardized software, together with their inputs and expected outputs. By creating this standard, it is possible to run the same benchmarks in different machines, resulting in a comparative evaluation between those compute systems. Pin is a software used to create instrumentation tools called Pin tools. A instrumentation tool allows the analysis of source code at multiple possible levels, such as instruction level or a complete binary module. Among the possible Pin tools, there is instruction counters, tools to check memory usage, and to count certain types of instructions.

The main objective of this project is to use Pin in conjunction with SPEC2006 to instrument the code of its benchmarks. First, one of the default Pin tools is employed to count the number of instructions of each of SPEC2006's benchmarks. Then, a custom Pin tool was created and used with a subset of SPEC2006's benchmarks. Additionally, the custom Pin tool created for this project, a branchcounter, was run with C++ code specifically made for this project, with the object of testing how the addition of certain snippets of C++ code affect the instruction and branch count of code.

2. Instruction Count

To count program instructions, the Pin tools *inscount0* was used. To run SPEC2006 benchmarks, a configuration file is required to set several parameters of the system and benchmark execution. Alongside with those parameters, there are additional options available in case the user wants call arbitrary shell code before executing the benchmark. This option is vital to run Pin alongside SPEC, and is called *submit*, requiring the presence of the additional option *use_submit_for_speed=yes* in the config file. With the latter present, we add the option *submit* with the instruction to call Pin with the desired Pin tool, along wit the outputs desired. Besides inscount, a custom Pin tool named *branchcount* was developed, and both configuration files to run Spec with inscount and branchcount

are included in the project. Tables 1 and 2 show the instruction count for all SPEC2006's benchmarks, running with their *ref* inputs.

Table 1. SPEC CINT Benchmarks

Benchmark	Workload	Input	Instruction Count
Perlbench	0	checkspam.pl	1069544937088
	1	diffmail.pl	371964871042
	2	splitmail.pl	669091562527
Bzip	0	input.source	413615853925
	1	chicken.jpg	178450462610
	2	liberty.jpg	298209843911
	3	input.program	542582451452
	4	text.html	623096107014
	5	input.combined	331760956288
GCC	0	166.in	75706142591
	1	200.in	146966530703
	2	c-typeck.in	133776560679
	3	cp-decl.in	100369517875
	4	expr.in	110289618513
	5	expr2.in	149680662990
	6	g23.in	177923543828
	7	s04.in	164705611026
	8	scilab.in	56545708079
MCF	0	inp.in	311029794650
Gobmk	0	13x13.tst	234560548252
	1	nngs.tst	623716638224
	2	score2.tst	322198876011
	3	trevorc.tst	235576420223
	4	trevord.tst	337969056601
Hmmer	0	nph3.hmm	876601187172
	1	retro.hmm	1855017589667
Sjeng	0	ref.txt	2258529991100
Libquantum	0	control	2289896883609
h264ref	0	foreman_ref_baseline.cfg	497188994414
	1	foreman_ref_main.cfg	337637736169
	2	sss_main.cfg	3010067532567
omnetpp	0	omnetpp.ini	579903797320
Astar	0	BigLakes2048.cfg	404577450279
	1	rivers.cfg	811982820904
xalancbmk	0	xalanc.xsl	985814730456

Table 2. SPEC CFP Benchmarks

Benchmark	Workload	Input	Instruction Count
Bwaves	0	bwaves.in	3437552355500
Gamess	0	cytosine.2	1063965179977
	1	h2ocu2+.gradient	829516185715
	2	triazolium	3472860200957
MILC	0	su3imp.in	1134108297247
ZeusMP	0	zmp_inp	1808264464497
Gromacs	0	gromacs.tpr	1963926886552
CactusADM	0	benchADM.par	2714628034402
Leslie3D	0	leslie3d.in	1533984586721
Namd	0	namd.input	2284200619271
Deall	0	DummyData	2033745980320
Soplex	0	pds-50.mps	360970972907
	1	ref.mps	381470483535
Povray	0	SPEC-benchmark-ref.pov	979010201501
Calculix	0	hyperviscoplastic.inp	6175756751379
GemsFDTD	0	ref.in	1404866912815
Tonto	0	stdin	2597365708209
Lbm	0	lbm.in	1245406278460
Wrf	0	namelist.input	2978070671050
Sphinx3	0	args.an4	3356034639320

3. Branch Count and Additional Experiments

Alongside with running inscount with each of SPEC2006's Benchmarks, another task of this project was the development of a custom Pin tool to be run with a subset of such benchmarks. The tool developed, named *branchcount*, counts the number of branches in the code, alongside with the total number of instructions and instructions that are not branch. The idea was not only to check the number of branches in the code, but also if the number of branches and not-branches add up to the total of instructions. Table 3 shows the results for a subset of SPEC2006 benchmarks.

The branchcount Pin tool maintains three counters: one for the total instructions, one for the number of branches, and the last for the number of not-branches. After each instruction, the instruction counter is incremented, and the current instruction is checked by using the function:

```
INS_IsBranch(ins)
```

where *ins* is the coming instruction. When the program exits, the branchcount outputs a file with the values of the three aforementioned counters.

In addition, another experiment was conducted using the branchcount. The idea of this experiment was to start with a very simple C++ program, counting its instructions and branches, and then count again after small modifications, like adding a variable attribution or a conditional. The starting program is shown in Listing 1:

Table 3. branchcount run on selected SPEC2006 benchmarks

Benchmark	Workload	Total Instructions	Branch Instructions	Not Branch Instructions
Perlbench	0	1069541150994	211869495007	857671655987
	1	371964870703	69289379553	302675491150
	2	669091534863	129596970248	539494564615
Bzip	0	413616954955	59554233510	354062721445
	1	178450449617	26496125150	151954324467
	2	298209948983	46488189675	251721759308
	3	542583552467	79876551145	462707001322
	4	623095308774	101593985328	521501323446
	5	331760999512	48018770385	283742229127
GCC	0	75705953101	15673908796	60032044305
	1	146954198754	29506411052	117447787702
	2	133785988462	26944433960	106841554502
	3	100372388190	21313633558	79058754632
	4	110291547494	23848598229	86442949265
	5	149677230432	32348269718	117328960714
	6	177923181638	41266146953	136657034685
	7	164704487353	33383284205	131321203148
	8	56547057263	11365993623	45181063640
Bwaves	0	3437552355298	255328726473	3182223628825
MILC	0	1134108297247	51767664170	1082340633077

Listing 1. Simple C Program

```
int main(void){

    return 0;

}
```

This program simply enters the main function and returns, with no includes and no input arguments for the main function. Next, Listing 1 is modified by adding an integer attribution of the variable k, depicted by Listing 2:

Listing 2. Variable attribution

```
int main(void){

    int k = 0;
    return 0;

}
```

Listings 3 and 4 show two variants with an if condition, the first being a satisfied if condition ($k < 10$), and the second a non-satisfied one ($k > 10$). Both if conditions are devoid of any commands inside.

Listing 3. Satisfied IF condition

```
int main(void){

    int k = 0;

    if(k < 10)

    return 0;

}
```

Listing 4. Non-Satisfied IF condition

```
int main(void){

    int k = 0;

    if(k > 10)

    return 0;

}
```

Finally, the last variant shown at Listing 5 adds an empty $else$ to the non-satisfied if of Listing 4.

Listing 5. Non-Satisfied IF condition then ELSE

```
int main(void){

    int k = 0;

    if(k > 10);
    else

    return 0;

}
```

Table 4 shows the results of branchcount for the aforementioned C programs. Although Listing 1 consists of simply calling the main function and returning, with no $includes$ or other commands whatsoever, the tool counts a total of around 87.7k, with around 15.5k of them being branches. This is surprising, given the simplicity of the source code. Moreover, if we compare it with Listing 2, which consists in the addition of an integer attribution, this second program has only one additional instruction, meaning that the number of 87.7k instructions should be needed to run even the simplest code. Next, we have the scenario with the satisfied and non-satisfied if condition. The first adds three new instructions, and interestingly, two of them are branches, even though there is only one if condition. On the other hand, the satisfied if condition adds only two new instructions, one of them a branch. A possible reason for this happening is the way the compiler views both conditionals, and some possible optimization done by it. Because

the condition of Listing 4 is not satisfied, it is feasible that the compiler sees it and avoids the branch entirely. In Listing 5, on the other hand, because of the presence of the $else$, the compiler cannot avoid the branch, resulting in the same count as Listing 3.

Table 4. Branchcount results for programs 1 to 5

Listing Number	Total Instructions	Branch Instructions	Not Branch Instructions
1	87710	15501	72209
2	87711	15501	72210
3	87714	15503	72211
4	87713	15502	72211
5	87714	15503	72211

4. Conclusion

In this project, we employed the instrumentation tool Pin with the different benchmarks available in the tool SPEC2006. The first objective was to use a Pin tool to count the instructions of the different benchmarks. Next, we were tasked with developing a custom Pin tool, and testing it against a subset of the aforementioned benchmarks. In addition to those two tasks, a small experiment was performed on simple C++ programs to check their number of instructions and branches.

References

Pin - a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`. Accessed: 2016-09-15.

Spec cpu 2006. `https://www.spec.org/cpu2006/`. Accessed: 2016-09-15.