Homework 0

Yu Liu

September 17, 2019

Instructions:

- Please type your solutions into a document and convert it into a PDF file. Your solution document should contain your name, student ID, the course name, and homework number. Please submit your solution PDF via Canvas.
- Make reasonable assumptions where necessary and clearly state them!
- You may discuss concepts with your classmates. This fosters group learning and improves the class' progress as a whole. However, make sure to submit your own independent and individual solutions.

Problem 1:
For the first part of this homework, compile and run the Dhrystone benchmark on 2 different microarchitectural platforms. The benchmark is provided on this page (drystone.c). Note, you may get some warnings while compiling this benchmark - you can ignore them. You can also use any systems you have available. As a suggestion, you can use your personal PC/Laptop and UNCC mosaic Linux machines. Note: you may have to adjust the number of LOOPS specified in the source code to get a reasonable Dhrystone numbers. You may also have to modify the libraries used to get the C code to compile. Try different numbers of iterations until you get a reasonable result. Also, make multiple runs of Dhrystone to guarantee you have factored out any sampling or cold-start effects. Generally 10-20 runs is enough to obtain statistical significance. Make sure to discussion this issue in your paper and report on the error in your measurements.

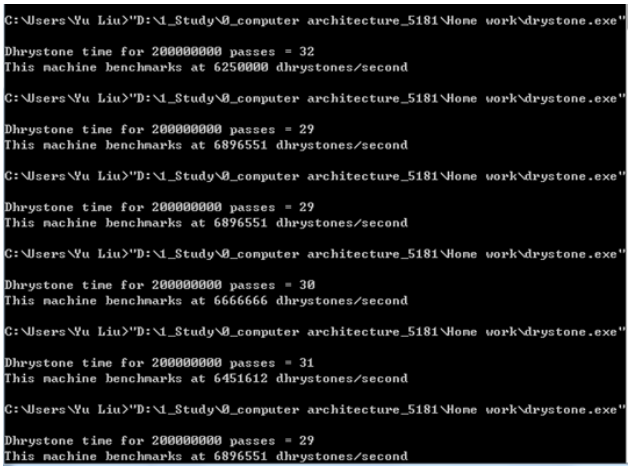
1.a. Did Dhrystone run slower the first time you ran it? Why? Try to explain this phenomenon (10 pts).

Answer:
Yes, the first time runs slower. Because at first beginning, the program has not been complied yet, after that, it has been stored in cache, which is faster than be fetched from memory.
Table 1 is the statistics of the Dhrystone runs under different loops and platforms.
Figure 1 indicates two of the program execution screenshots.

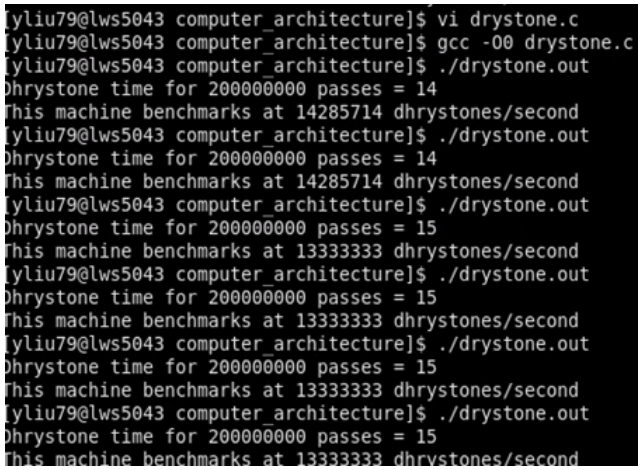
Table 1: Dhrystone runs on two different platforms

Model\Loops	Loops				
	100,000	30,000,000	50,000,000	100,000,000	200,000,000
Laptop X86 64	dump	5,5,5,4,5,4	9,8,6,8,8,8	16,15,13,15,15,15	32,29,29,30,31,29
Mosaic Linux	dump	2,2,2,2,2,1	4,3,4,4,4,2	7,7,7,7,7,7	15,14,15,15,15,15

1.b. Run Dhrystone compiled with and without optimization (find out how to use the compiler switches, the man gcc pages should hep). Explain the results you are getting. How does optimization affect the results obtained on the different architectures and why (10ptes)? There are probably more than a million different switches you can try. Start by using the -OX switch, where X is (0, 1, 2, 3). This is an open-ended question, that should challenge everyone to find the switches that most impact this program. Make sure to explain why a particular switch is giving you good performance. This is the most important part of the assignment. Give examples that illustrate your reasoning (e.g., provide assembly listings and show what the compiler is doing, but do not just print out the entire listing) (20 pts).



(a) Laptop X86/64



(b) Mosaic Linux

Figure 1: Program execution screenshots under two platforms

Answer:
From the Table 2, we can see the optimization O1 obviously reduces this benchmark's running time. Meanwhile, O2 and O3 offer the best performance in the result of passes.

Table 2: Dhrystone runs on different optimizations

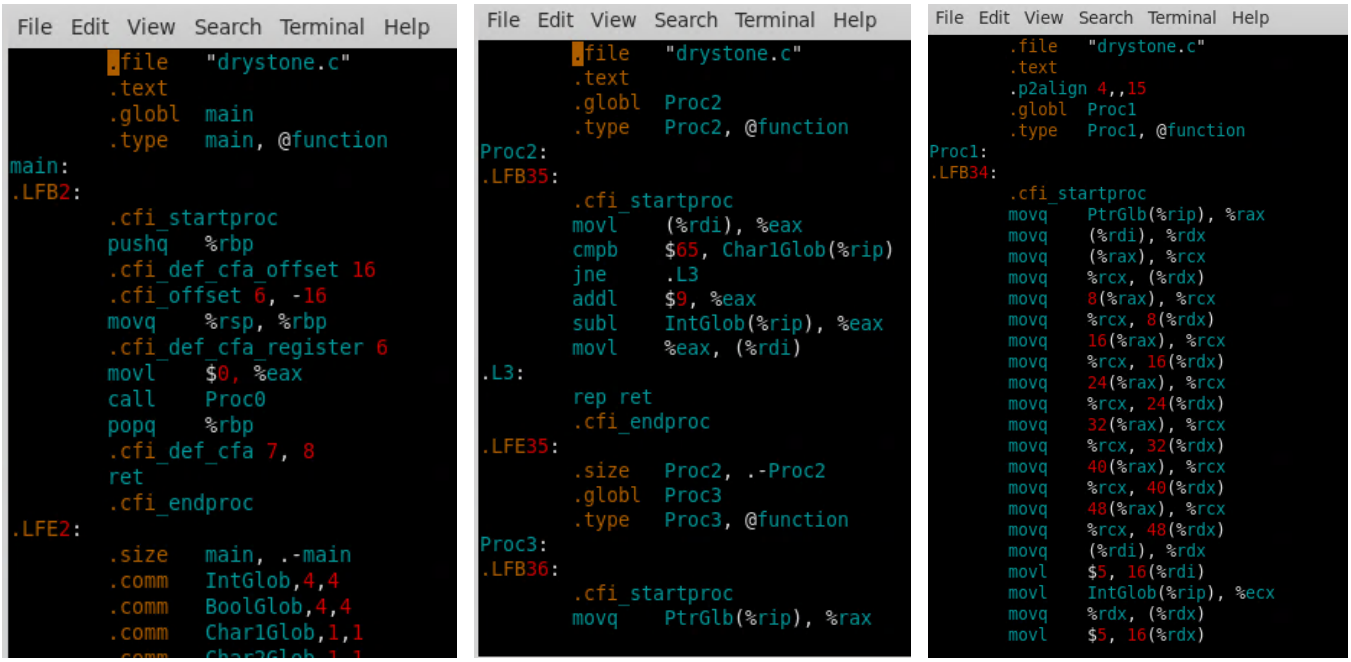
Optimization\Loops	Loops				
	100,000	50,000,000	100,000,000	200,000,000	500,000,000
O0	dump	4,3,4,4,4,3	7,7,7,7,7,7	14,14,15,15,15,15	37,35,35,37,36,35
O1	dump	1,1,1,1,2,1	3,3,2,3,3,3	6,6,5,6,6,5	14,13,13,14,12,14
O2	dump	1,1,1,2,1,1	2,2,2,3,2,2	4,4,4,4,4,4	11,12,12,11,11,11
O3	dump	1,1,2,1,1,1	2,2,2,2,3,2	4,5,4,4,4,4	10,11,11,10,11,11

When compare the assembly codes between no optimization and optimization O1, some condition branch sequence is found to be changed, as well as the code size is smaller, these re-organization contribute the improvement. O3 is expected to further optimize the cross functions, the reason of improvement result is not significant is because O1 does some sequence re-organization already.

When compared with optimization O3's assembly code, some condition functions (such as Proc3) are found to be broken down to small parts, as well as complier O3 delete some commands which are "meaningless" ,so to make the program run faster.
Some part of assembly code please refer Figure 2.

1.c. Answer these questions: - What is the most frequently executed function (5pts)?
- What percentage of the entire execution time does it consume (5pts)?
- How does optimization change this percentage and why (5pts)?
Note: You should use gprof (on Linux) to profile the execution of the program. To run gprof, you will need to compile your code with debug information (figure out which compiler switch to use). gprof will help you to find the "hot" portions of the code.

Answer:
Most frequently execute function are Proc3 and _fini.

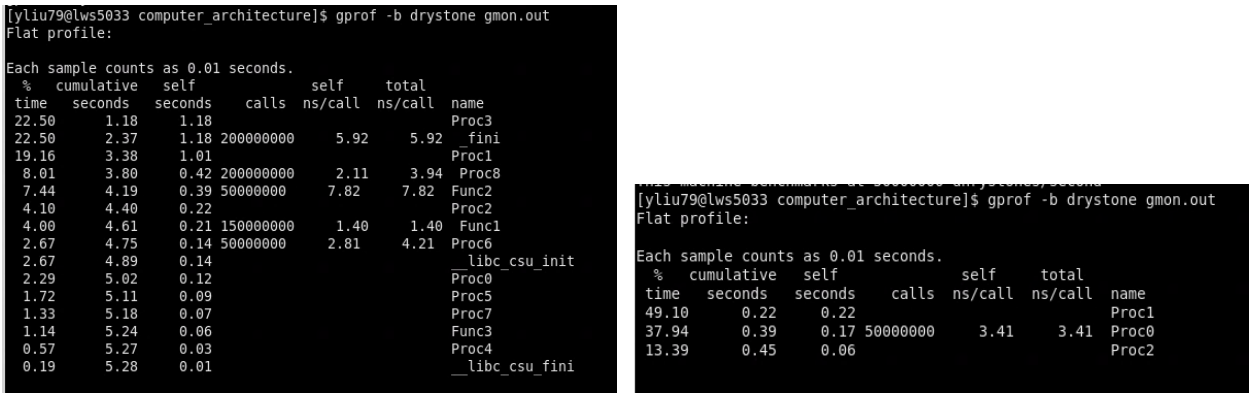


(a) Without optimization (b) optimizationO1 (c) optimizationO3

Figure 2: Assembly codes of dhrystone with and without optimization

Both of them occupy 22.5% of the entire execution time. Below first figure is what of without optimization and the second figure is what has be optimized by O3.
The optimization O3 "ignores" the some functions such as Proc3, which complier regards it do not bring the result to the program, that makes the execution time faster.
Please refer the gprof output result under no optimization and optimization O3 in Fugure 3.

Problem 2:
Next we are going to look at the Linpack benchmark.
Compile the program on X86/Linux and provide a detailed analysis how optimizations in the compiler can improve performance. Note, this is a floating point benchmark, so you will need to look at using math libraries and understand floating point instructions. You only need to complete this problem on one machine (20pts).



[yliu79@lws5033 computer_architecture]$ gprof -b drystone gmon.out
Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
22.50	1.18	1.18				Proc3
22.50	2.37	1.18	200000000	5.92	5.92	_fini
19.16	3.38	1.01				Proc1
8.01	3.80	0.42	200000000	2.11	3.94	Proc8
7.44	4.19	0.39	50000000	7.82	7.82	Func2
4.10	4.40	0.22				Proc2
4.00	4.61	0.21	150000000	1.40	1.40	Func1
2.67	4.75	0.14	50000000	2.81	4.21	Proc6
2.67	4.89	0.14				__libc_csu_init
2.29	5.02	0.12				Proc0
1.72	5.11	0.09				Proc5
1.33	5.18	0.07				Proc7
1.14	5.24	0.06				Func3
0.57	5.27	0.03				Proc4
0.19	5.28	0.01				__libc_csu_fini

(a) Gprof output_No optimization

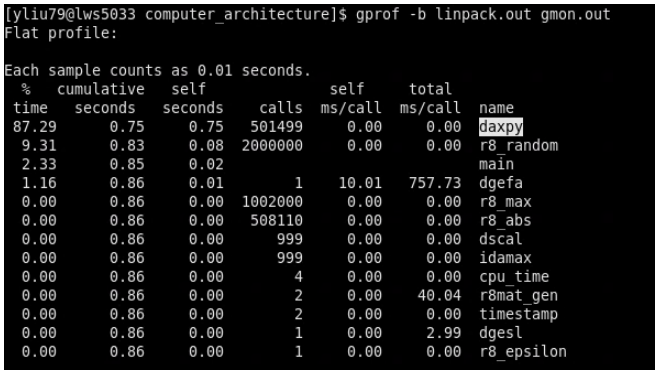
[yliu79@lws5033 computer_architecture]$ gprof -b drystone gmon.out
Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
49.10	0.22	0.22				Proc1
37.94	0.39	0.17	50000000	3.41	3.41	Proc0
13.39	0.45	0.06				Proc2

(b) Gprof output_Optimization O3

Figure 3: Gprof Output



```
[yliu79@lws5033 computer_architecture]$ gprof -b linpack.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 87.29      0.75     0.75   501499     0.00     0.00  daxpy
  9.31      0.83     0.08  2000000     0.00     0.00  r8_random
  2.33      0.85     0.02                             main
  1.16      0.86     0.01        1    10.01   757.73  dgefa
  0.00      0.86     0.00  1002000     0.00     0.00  r8_max
  0.00      0.86     0.00   508110     0.00     0.00  r8_abs
  0.00      0.86     0.00      999     0.00     0.00  dscal
  0.00      0.86     0.00      999     0.00     0.00  idamax
  0.00      0.86     0.00        4     0.00     0.00  cpu_time
  0.00      0.86     0.00        2     0.00    40.04  r8mat_gen
  0.00      0.86     0.00        2     0.00     0.00  timestamp
  0.00      0.86     0.00        1     0.00     2.99  dgesl
  0.00      0.86     0.00        1     0.00     0.00  r8_epsilon
```

(a) Run without optimization

```
15 September 2019 03:20:31 PM
[yliu79@lws5033 computer_architecture]$ gprof -b linpack.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 85.33      0.23     0.23   501499     0.00     0.00  daxpy
  7.42      0.25     0.02  2000000     0.00     0.00  r8_random
  7.42      0.27     0.02        1    20.03   249.50  dgefa
  0.00      0.27     0.00      999     0.00     0.00  dscal
  0.00      0.27     0.00      999     0.00     0.00  idamax
  0.00      0.27     0.00        4     0.00     0.00  cpu_time
  0.00      0.27     0.00        2     0.00    10.02  r8mat_gen
  0.00      0.27     0.00        2     0.00     0.00  timestamp
  0.00      0.27     0.00        1     0.00     0.92  dgesl
  0.00      0.27     0.00        1     0.00     0.00  r8_epsilon
```

(b) Run after optimization O1

Figure 4: Gprof of linpack

Answer:
We got the result of MFLOPS as below table by running under without optimization, , Optimization O0, Optimization O1, Optimization O2 and Optimization O3.
Please refer to Table 3.

Table 3: Linpack runs on different optimizations

Optimization Level	No Optimization	Optimization O0	Optimization O1	Optimization O2	Optimization O3
MFLOPS	777.915380	835.833333	2571.794872	2786.111111	3184.126984

It can be observed that performance generally improved as the sequence of : O3 , O2, O1, O0, None, relatively, O0 improve significantly..
When checking the execution profile, we found function of "daxpy" consumes the most execution time. And the execution time drops down from 0.75 to 0.23 after running the optimization O1. Please refer to the Figure 4.

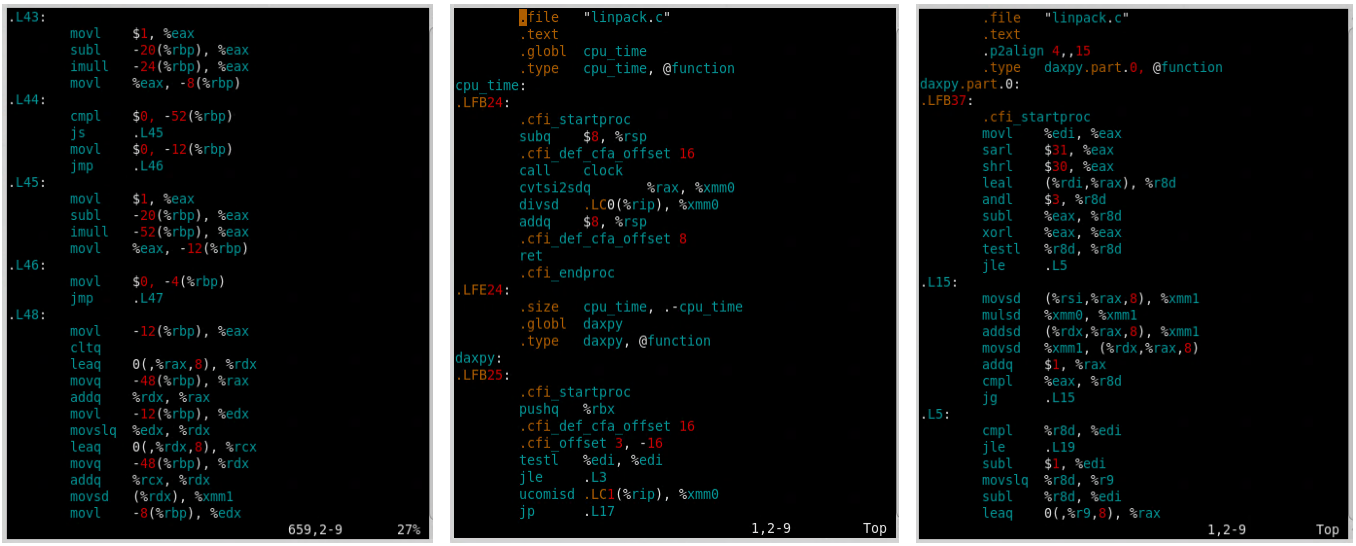
Further checking the source code in linpack.c, it indicates this function is of implementing huge loops of " computing constant times a vector plus a vector" with float format (double format) variables. O1 optimizes the code sequence and make loop optimization like "floop-optimize", O3 further optimize such as "link rename-registers", which is benefit for the floating calculation. Please refer to the Figure 5 as comparison..

Problem 3:
Find a benchmark on the web (not Dhrystone or Linpack). Compile and run it on a system of your choice.
3.a Discuss what the benchmark is designed to evaluate, and discuss the benchmarking results you obtain for this benchmark on the system you have chosen to run it on (10pts).
3.b Can you suggest a system where it might run more efficiently (5pts)? Make sure to justify your answer.

Answer:

I choose the Benchmark of whetstone to run on X86_64_Intel Core 2 Due. Figure 6 indicates the different optimization levels impact the running time, higher optimization level brings faster running time.

Whetstone is programmed for measuring floating point for minicomputers, so use it to testing embedded hardware, for example , single-precision FPU's floating capability will get more solid result.



(a) Without optimization (b) Optimization O1 (c) Optimization O3

Figure 5: Part of assembly code of linpack.c

```
D:\temp>whetstone

Loops: 100000, Iterations: 1, Duration: 11 sec.
C Converted Double Precision Whetstones: 909.1 MIPS

D:\temp>whetstone_o0

Loops: 100000, Iterations: 1, Duration: 10 sec.
C Converted Double Precision Whetstones: 1000.0 MIPS

D:\temp>whetstone_o1

Loops: 100000, Iterations: 1, Duration: 5 sec.
C Converted Double Precision Whetstones: 2000.0 MIPS

D:\temp>whetstone_o2

Loops: 100000, Iterations: 1, Duration: 6 sec.
C Converted Double Precision Whetstones: 1666.7 MIPS

D:\temp>whetstone_o3

Loops: 100000, Iterations: 1, Duration: 3 sec.
C Converted Double Precision Whetstones: 3333.3 MIPS
```

Figure 6: Whetstone measurement result under different optimization level

Problem 4:
A benchmark suite is a set of applications used to characterize the performance of a processor or system. There are many different suites available. Select two suites, and answer the following questions:
- What application domain or system architecture is the suite designed to evaluate (5pts)?
- How is performance evaluated with the benchmark? What is the metric used to evaluate performance (5pts)?
- Cite 2 technical/scientific (IEEE or ACM) papers each (a total of 4) where each suite has been used in a research study (5pts).
- Provide your thoughts on how the suite could be improved (5pts).

Answer:
I select two benchmark suits of UnixBench and LMbench.

UnixBench measures a basic indicator of the performance of a Unix-like operating system. It combined the benchmark tests including Dhrystone ,Whetstone ,Execl Throughput, File Copy, Pipe Throughput, Pipe-based Context Switching, Process Creation, Shell Scripts, System Call Overhead and Graphical Tests.
After install and running the suit, there will output each test item's score and a total score, which higher score indicates higher performance.
Below are two papers searched from IEEE.

References

[1] Predicting the Effect of Memory Contention in Multi-Core Computers Using Analytic Performance Models
Shouvik Bardhan ; Daniel A. Menascé
IEEE Transactions on Computers
Year: 2015 — Volume: 64, Issue: 8 — Journal Article — Publisher: IEEE

[2] CPU and memory performance analysis on dynamic and dedicated resource allocation using XenServer in Data Center environment
Haydar Ali Ismail ; Mardhani Riasetiawan
2016 2nd International Conference on Science and Technology-Computer (ICST)
Year: 2016 — Conference Paper — Publisher: IEEE

LMbench is a suite of simple and portable benchmarks to measure the operating system and hardware system metrics in UNIX/POSIX, covering memory, cache and networking, etc.
LMbench mainly focus on two features of latency and bandwidth.
After installing and running the LMbench, the result can be directly got in the metric of bandwidth in MB/s and latency in nano-second.
Below are two papers searched via IEEE:

References

[1] A Linux Server Operating System's Performance Comparison Using Lmbench
Zexin Jiang
2016 International Conference on Network and Information Systems for Computers (ICNISC)
Year: 2016 — Conference Paper — Publisher: IEEE

[2] Pthreads Performance Characteristics on Shared Cache CMP, Private Cache CMP and SMP
Ian K.T. Tan ; Ian Chai ; Poo Kuan Hoong
2010 Second International Conference on Computer Engineering and Applications
Year: 2010 — Volume: 1 — Conference Paper — Publisher: IEEE

My improvement suggestion is to make the two benchmarks more easy to use, to be compatible to CMU , such as ARM and window embedded, support measuring them without modifying the source code.