PERFORMANCE OPTIMIZATION OF ALGORITHMS

SOFTWARE PROGRAMMING FOR PERFORMANCE

Dolton Fernandes and Naren Akash, R J

International Institute of Information Technology Hyderabad

January 24, 2020

Software Programming for Performance

Performance Profiling

Performance analysis of computer systems and applications have gained a significant attention in the recent years. As the complexity and requirements increases, applications are expected to perform greater computation in lesser time. Developers require tools to aid them in determining bottlenecks in the code which if optimized would yield the best overall speed-up. Such tools may be used to decrease the execution time, increase the efficiency of resource utilization, or the combination of the two.

Profilers

Performance profiling tools allows us to identify the performance bottlenecks in a program and measure salient performance properties, such as performance and branch misses.

Perf

perf is a profiler tool for Linux 2.6+ based systems. It uses sampling (ie. measuring applications without inserting any modifications) to gather data about important software, kernel, and hardware events in order to locate performance bottlenecks in a program. It also generates a detailed record of where the time in spent in our code.

```
>> perf record <program_name> <program_arguments>
```

perf record generates a record of the events that occur when we run the code

```
>> perf report
```

perf record allows us to view such records of events interactively.

Cachegrind

Cachegrind is used for collecting statistics about cache misses. It simulates L1 (I), L1 (D) and L2 cache with the default size the same as the current machine's cache. It outputs total program run hit and miss count, per function hit and miss count and per source code line hit and miss count.

```
>> valgrind –tool=cachegrind <program_name> --I1=<size> --D1=<size> --L2=<size>
```

Summary of instruction fetch cache accesses, data cache accesses (read + write) and combined instruction and data cache accesses of L2 will be printed in the same order.

Gprof

gprof is also a profiling tool

```
>> gcc –g –pg –o <executable_name>  <program_name>
```

First, we compile the program specifically for profiling purpose. Once we have compiled with profiling turned on, running the program to completion creates a file named gmon.out in the current directory. grof analyzes data collected during the execution of the program after its completion.

```
>> gprof <program_name> [data_file]  [ > output_file]
```

gprof's analysis report is saved as program_name.output. The profile tells us the total amount of time spent on each function.

clock_gettime

clock_gettime system call has the ability to request specific clocks with high precision. It fills in a structure containing two fields – a seconds and a nanosecond count of the time since the Epoch.

```
#include <time.h>

struct timespec {

   time_t   tv_sec;       /* seconds */

   long     tv_nsec;      /* nanoseconds  */

};

 int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

References

Carnegie Mellon University - Valgrind User Manual

TecMint - Perf: A Performance Monitoring and Analysis Tool for Linux

University of Michigan - gprof Quick-Start Guide

MIT 6.172: Performance Engineering of Software Systems

Rutgers CS: Linux's High Resolution Clock

Matrix Multiplication

1. Loop Order

Changing the order of the loops in the matrix-multiplication algorithm improves the performance of the program drastically without affecting the correctness of the code.

In C, matrices are laid out in row-major order. The access patterns of I, j and k in the matrix-multiplication code affects the last-level cache miss rates due to locality in caches.

2. register keyword

register keyword requests the compiler that the specified value to be stored in a register of the processor instead of memory. This is a way to gain more speed.

3. Struct versus Array

4. Storing Array Element in a Variable

In the code, a->matrix[I][j] is used on all the iterations of the inner loop. Hence, we store it in an integer variable to reduce the access time and hence, get a higher performance.

Merge Sort

1. register keyword

2. Bit Hacks

Changes made in the mathematical operations led to drastic increase in the performance by reducing computations and branch missing.

3. memcpy instead of usual copy-loop

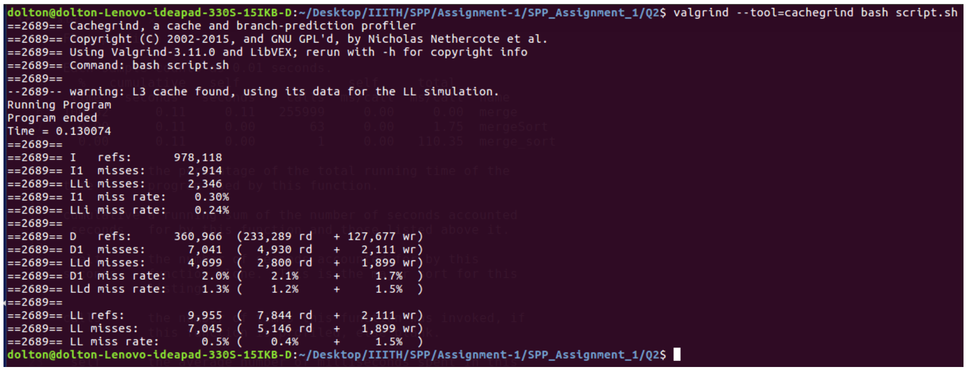
4. Insertion sort when n is very small

Insertion sort performs better when n is very small.

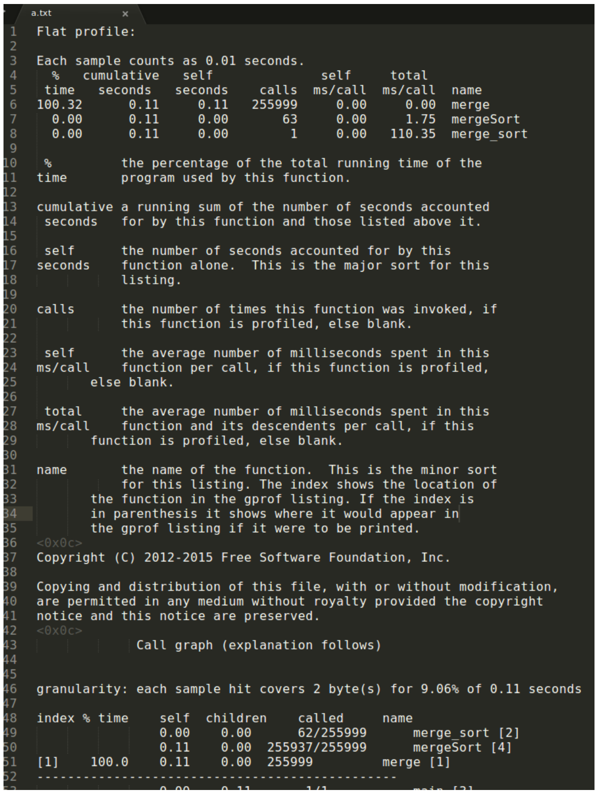
5. Array Divided in chuncks of size < 16,000 : Divide and Conquer

Cache size is 32,000 B. These chuncks are sorted using base merge sort and combined using the usual merge function.

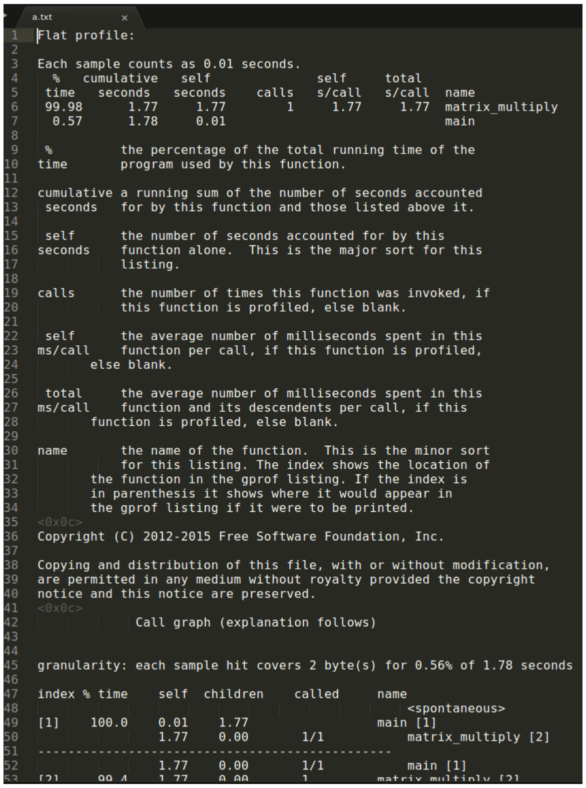
Annexures



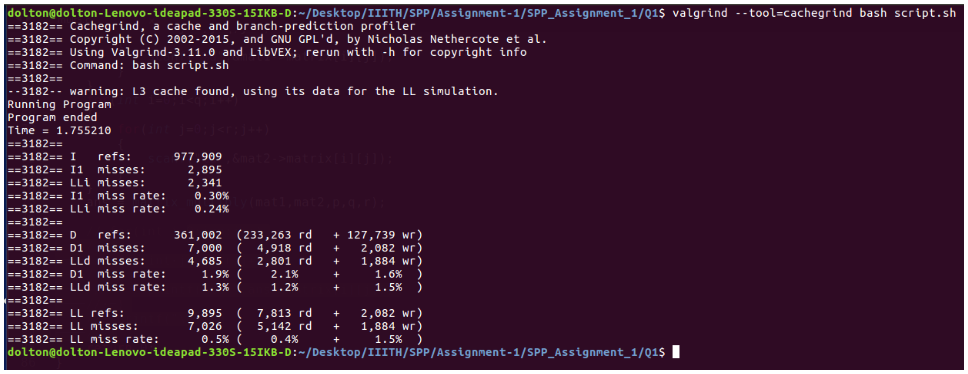
```
dolton@dolton-Lenovo-ideapad-330S-15IKB-D:~/Desktop/IIITH/SPP/Assignment-1/SPP_Assignment_1/Q2$ valgrind --tool=cachegrind bash script.sh
==2689== Cachegrind, a cache and branch-prediction profiler
==2689== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==2689== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2689== Command: bash script.sh
==2689==
--2689-- warning: L3 cache found, using its data for the LL simulation.
Running Program
Program ended
Time = 0.130074
==2689==
==2689== I   refs:      978,118
==2689== I1  misses:      2,914
==2689== LLi misses:      2,346
==2689== I1  miss rate:    0.30%
==2689== LLi miss rate:    0.24%
==2689==
==2689== D   refs:      360,966  (233,289 rd   + 127,677 wr)
==2689== D1  misses:      7,041  (  4,930 rd   +   2,111 wr)
==2689== LLd misses:      4,699  (  2,800 rd   +   1,899 wr)
==2689== D1  miss rate:     2.0% (    2.1%     +     1.7%  )
==2689== LLd miss rate:     1.3% (    1.2%     +     1.5%  )
==2689==
==2689== LL refs:         9,955  (  7,844 rd   +   2,111 wr)
==2689== LL misses:       7,045  (  5,146 rd   +   1,899 wr)
==2689== LL miss rate:      0.5% (    0.4%     +     1.5%  )
dolton@dolton-Lenovo-ideapad-330S-15IKB-D:~/Desktop/IIITH/SPP/Assignment-1/SPP_Assignment_1/Q2$ 
```



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
100.32      0.11     0.11   255999     0.00     0.00  merge
  0.00      0.11     0.00       63     0.00     1.75  mergeSort
  0.00      0.11     0.00        1     0.00   110.35  merge_sort

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
       else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
       function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
       the function in the gprof listing. If the index is
       in parenthesis it shows where it would appear in
       the gprof listing if it were to be printed.
<0x0c>
Copyright (C) 2012-2015 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
<0x0c>
             Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 9.06% of 0.11 seconds

index % time    self  children    called     name
                0.00    0.00      62/255999      merge_sort [2]
                0.11    0.00  255937/255999      mergeSort [4]
[1]    100.0    0.11    0.00  255999         merge [1]
-----------------------------------------------
                0.00    0.11       1/1           main [3]
```



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls   s/call   s/call  name    
 99.98      1.77     1.77        1     1.77     1.77  matrix_multiply
  0.57      1.78     0.01                             main

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
       else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
       function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
       the function in the gprof listing. If the index is
       in parenthesis it shows where it would appear in
       the gprof listing if it were to be printed.
<0x0c>
Copyright (C) 2012-2015 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
<0x0c>
             Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.56% of 1.78 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01    1.77                 main [1]
                1.77    0.00       1/1           matrix_multiply [2]
-----------------------------------------------
                1.77    0.00       1/1           main [1]
[2]     99.4    1.77    0.00       1         matrix_multiply [2]
```



```
dolton@dolton-Lenovo-ideapad-330S-15IKB-D:~/Desktop/IIITH/SPP/Assignment-1/SPP_Assignment_1/Q1$ valgrind --tool=cachegrind bash script.sh
==3182== Cachegrind, a cache and branch-prediction profiler
==3182== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==3182== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3182== Command: bash script.sh
==3182==
--3182-- warning: L3 cache found, using its data for the LL simulation.
Running Program
Program ended
Time = 1.755210
==3182==
==3182== I   refs:      977,909
==3182== I1  misses:      2,895
==3182== LLi misses:      2,341
==3182== I1  miss rate:    0.30%
==3182== LLi miss rate:    0.24%
==3182==
==3182== D   refs:      361,002  (233,263 rd   + 127,739 wr)
==3182== D1  misses:      7,000  (  4,918 rd   +   2,082 wr)
==3182== LLd misses:      4,685  (  2,801 rd   +   1,884 wr)
==3182== D1  miss rate:     1.9% (    2.1%     +     1.6%  )
==3182== LLd miss rate:     1.3% (    1.2%     +     1.5%  )
==3182==
==3182== LL refs:         9,895  (  7,813 rd   +   2,082 wr)
==3182== LL misses:       7,026  (  5,142 rd   +   1,884 wr)
==3182== LL miss rate:      0.5% (    0.4%     +     1.5%  )
dolton@dolton-Lenovo-ideapad-330S-15IKB-D:~/Desktop/IIITH/SPP/Assignment-1/SPP_Assignment_1/Q1$
```