



PA039: Supercomputer Architecture and Intensive Computing

Profiling and Benchmarking

Luděk Matyska

Spring 2024



Measurement of processing/compute time

- Optimization impossible without knowledge what to optimize
- We need to know data about program run
 - Processing/run time of the whole program: **time** command
 - Processing/run time of individual program components: profiling
 - Run time comparison: benchmarking

time command

- User time
 - Processor time consumed by user's processes
- System time
 - Processor time consumed by the kernel services
- Elapsed time
 - Total run time (wall clock time)

CPU time = user time + system time

time command – further data

- GNU **time**: usually as `/usr/bin/time`
- Extensive set of commands (see **man 1 time**)
- Shared memory space
- Private (unshared) memory space
- Number of block input operations
- Number of block output operations
- Number of page faults
- Number of page swaps
- ...

Example

```
/usr/bin/time -v pdflatex parallel.tex
```

```
Command being timed: "pdflatex parallel.tex"  
User time (seconds): 1.32  
System time (seconds): 0.02  
Percent of CPU this job got: 97%  
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.38  
Average shared text size (kbytes): 0  
Average unshared data size (kbytes): 0  
Average stack size (kbytes): 0  
Average total size (kbytes): 0  
Maximum resident set size (kbytes): 39760  
Average resident set size (kbytes): 0  
Major (requiring I/O) page faults: 3
```

```
Minor (reclaiming a frame) page faults: 7407  
Voluntary context switches: 252  
Involuntary context switches: 128  
Swaps: 0  
File system inputs: 19704  
File system outputs: 2848  
Socket messages sent: 0  
Socket messages received: 0  
Signals delivered: 0  
Page size (bytes): 4096  
Exit status: 0
```

Profiling

- Attempt to get timing information about program parts
- Emphasis on dynamic (run time) behavior
 - static analysis is a part of software engineering
- Profiling shows a result of an interaction between program and the computing system it runs on
 - Time spent in individual blocks
 - Time spent in individual commands
 - Number of repetition of blocks/commands
- Primary interest on procedures
- Profile: graph
 - X axis: individual procedures
 - Y axis: run time

Basic principles

- Uses **software tools** for collection of data needed for the profile construction
- Usually some operating system support
 - access to information available to kernel only
- Examples of usual profiling tools
 - gprof, oprofile, valgrind, pin
- Profile collected during the run time – **dynamic profile**
- **Performance Engineering** is the name of the profile data analysis

Types of data collected

- **Call graph** at the procedures and functions level
- **Call graph** at the basis blocks level
- **Memory performance**
- **Events** related to the architecture, e.g. incorrect branch predictions, exemptions, cache performance (hit/miss), ...
- **Performance counters** values

Profile types

■ Sharp profile

- Peaks related to the dominating blocks/procedures
- Numerical applications, technical computing with matrices etc.
- “Easily” optimizable

■ Flat profile

- Program spends its run time uniformly in all blocks/procedures
- Usually databases, information systems, operating systems, ...
- Difficult to optimize

■ Amdahl rule is valid here, too

Profilers

- Tools for profile construction
 - Naturally possible also “manually”
- Procedure oriented
 - **gprof**
- Block (line) oriented
 - **pixie**
 - **tcov**
 - **lprof**

Profilers' use

- Two phases:
 - Instrumented program execution (with or without a need for re-compiling)
 - Profile Analysis Report
- Access to the source code
 - Knowledge of program structure

It is possible to profile a program even without an access to the source code

Procedure oriented profilers

- Typical representative: **gprof**
- Instrumentation
 - Re-compiling of the program
 - Usually available through a special compiler key (e.g. **-pg**)
- Run time
 - Instrumented program creates a compute (processing) trail
 - File **gmon.out**
- Report construction
 - **gprof** run over the previously generated **gmon.out** file

Types of profiles

SpeedShop as an example

- usertime – user time
- [f]pcsamp[x] – sampling
- ideal (pixie) – block profiler
- fpe – floating point
- prof_hwc – hardware counters
 - f gi_hwc, [f]cy_hwc, [f]ic_hwc, [f]isc_hwc, [f]dc_hwc, [f]dsc_hwc,
[f]tlb_hwc, [f]gfp_hwc
- PAPI Framework

Measurement precision

- Time measurement
 - Absolute time of a procedure entry/exit
 - Nested procedures
 - Short procedures
 - Instruction counter value read in uniform intervals
 - Sampling interval influences the precision of the measurement

Block oriented profilers

- Provides information about basic block's processing
- Number of executions per command (program line)
- Number of processor cycles spent in each command

Example

```

1: static void foo(), bar(),
2:           baz();
3: main()
4: { int l;
5:   for (l=0; l <1000; l++)
6:     { if ( l == 2*(l/2) )
7:       foo();
8:       bar();
9:       baz();
10:    }
11: }
12:
13: void foo()
14: { int j;;
15:   for (j=0; j<200; j++);
16: }
17: void bar()
18: { int j;
19:   for (j=0; j<200; j++);
20: }
21: void baz()
22: { int j;
23:   for (j=0; j<300; j++);
24: }

```


Ustertime

ustertime:

index	%Samples	self	descendents	total	name
[1]	100.0%	0.00	0.03	1	main
[2]	100.0%	0.03	0.00	1	bar

Sampling effect

```
pcsamp:
samples      time(%)      cum time(%)      procedure
      2      0.02s( 50.0)    0.02s( 50.0) foo
      1      0.01s( 25.0)    0.03s( 75.0) bar
      1      0.01s( 25.0)    0.04s(100.0) baz
```

```
fpcsamp:
samples      time(%)      cum time(%)      procedure
      18      0.02s( 41.9)    0.02s( 41.9) baz
      12      0.01s( 27.9)    0.03s( 69.8) bar
      12      0.01s( 27.9)    0.04s( 97.7) foo
```

Block profile

ideal :

cycles(%)	cum %	secs	instrns	alls	procedure
3918000(49.63)	49.63	0.03	2111000	1000	baz
2618000(33.16)	82.80	0.02	1411000	1000	bar
1309000(16.58)	99.38	0.01	705500	500	foo
47024(0.60)	99.98	0.00	25017	1	main

Block profile II

ideal -h:

	cycles(%)	cum %	times	line	procedure
	3907858(49.50%)	49.50%	300000	23	baz
	2607858(33.04%)	82.54%	200000	19	bar
	1303930(16.52%)	99.06%	100000	15	foo
	14000(0.18%)	99.24%	1000	6	main
	13009(0.16%)	99.40%	1000	5	main
	8000(0.10%)	99.50%	1000	9	main
	8000(0.10%)	99.60%	1000	8	main
	7000(0.09%)	99.69%	1000	24	baz
	7000(0.09%)	99.78%	1000	20	bar
	4000(0.05%)	99.83%	500	7	main
	3500(0.04%)	99.88%	500	16	foo
	3142(0.04%)	99.92%	1000	18	bar
	3142(0.04%)	99.96%	1000	22	baz
	1570(0.02%)	99.98%	500	14	foo

Block profile (tcov)

tcov:

```

1 ->   for (l=0; l <1000; l++)
1000 -> { if ( l == 2*(l/2) )
500  ->   foo();
1000 ->   bar();
      baz();

      void foo()
500  ->   for (j=0; j<200; j++);

      void bar()
1000 ->   for (j=0; j<200; j++);

      void baz()
1000 ->   for (j=0; j<300; j++);

```

Block profile – continuation

Top 10 Blocks

Line	Count
6	1000
8	1000
19	1000
23	1000
7	500
15	500
5	1

```

7      Basic blocks in this file
7      Basic blocks executed
100.00 Percent of the file executed
5001  Total basic block executions
714.43 Average executions per basic block

```



Profiling

see also [http://www.site.uottawa.ca/~mbolic/elg6158/
Subbasis_profiling.pdf](http://www.site.uottawa.ca/~mbolic/elg6158/Subbasis_profiling.pdf)

Benchmarking

- Attempt to compare performance of whole *systems*
 - Jointly hardware and software
- No “silver bullet” solution
- Basic approaches
 - Professional benchmarks
 - Comparability, vendor independence
 - “Private” benchmarks
 - Specific requirements (deeper knowledge what you need/want)

Mysterious MIPS and MFLOPS

- Comparison based on number of instructions per second
- **MIPS** – million instructions per second
- **MFLOPS** – million floating point instructions per second
- Problems
 - Which instructions
 - Which order
- Artificial, not adequate

Fixed point benchmarks

- VAX MIPS
- Dhrystone

Floating point benchmarks

- Whetstone (artificial mix, scalar)
- Linpack (daxpy, vectorization)
 - 100*100
 - 1000*1000

SPEC benchmarks

- Independent organization
 - Standard Performance Evaluation Corporation
- Standardized benchmarks for different architectures
- Based on the so called *kernel codes*
 - Part or a whole **existing** program
 - Available in the source code
 - It is possible to “fine tune” the kernels

SPEC groups

- Open Systems Group (OSG)
- High Performance Group (HPG)
- Graphics Performance Characterization Group (GPC)

SPEC OSG subgroups

- SPEC CPU
- Graphics
 - SPECviewperf 13, SPECcapc for 3ds Max 2015, ...
- JAVA
 - SPECjAppServer 2004, SPECjbb 2015, SPECjvm 2008, Java client and server benchmarks
- MAIL
 - SPECmail 2009 (already retired)
- SFS
 - Systémy souborů (SPEC SFS 2014)
- SPEC Virt_SC 2013

CPU2006

- Composition
 - CINT2006 – fixed point calculations
 - CFP2006 – floating point calculations

CINT2000

■ Individual components

164.gzip	Compression
175.vpr	FPGA Circuit Placement and Routing
176.gcc	C Programming Language Compiler
181.mcf	Combinatorial Optimization
186.crafty	Game Playing: Chess
197.parser	Word Processing
252.eon	Computer Visualization
253.perlbnk	PERL Programming Language
254.gap	Group Theory, Interpreter
255.vortex	Object-oriented Database
256.bzip2	Compression
300.twolf	Place and Route Simulator

CINT2006

■ Individual components

400.perlbench	C	PERL Programming Language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: go
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess
462.libquantum	C	Physics: Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

CFP2000

■ Individual components

168.wupwise	Physics / Quantum Chromodynamics
171.swim	Shallow Water Modeling
172.mgrid	Multi-grid Solver: 3D Potential Field
173.applu	Parabolic / Elliptic Partial Differential Equations
177.mesa	3-D Graphics Library
178.galgel	Computational Fluid Dynamics
179.art	Image Recognition / Neural Networks
183.quake	Seismic Wave Propagation Simulation
187.facerec	Image Processing: Face Recognition
188.amp	Computational Chemistry
189.lucas	Number Theory / Primality Testing
191.fma3d	Finite-element Crash Simulation
200.sixtrack	High Energy Nuclear Physics Accelerator Design
301.apsi	Meteorology: Pollutant Distribution

CFP2006

410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry
433.milc	C	Physics: Quantum Chromodynamics
434.zeusmp	Fortran	Physics/CFD
435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
436.cactusADM	C/Fortran	Physics/General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology/Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450ZZ.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454.calculix	C/Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C/Fortran	Weather Prediction
482.sphinx3	C	Speech recognition

SPEC CPU2017

- Up to date CPU benchmark
- 4 parts

intspeed SPECSpeed 2017 Integer: 10 integer benchmarks

fpspeed SPECSpeed 2017 Floating Point: 10 integer benchmarks

Always just one copy of each benchmark

OpenMP can be used

Run time is measured

intrate SPECrate 2017 Integer: 10 integer benchmarks

fprate SPECrate 2017 Floating Point: 10 integer benchmarks

Always more copies of each benchmark (configurable at start)

OpenMP forbidden

Throughput is measured (how many tasks per time unit)

SPECspeed 2017 Integer

500.perlbench_r	C	Perl interpreter
502.gcc_r	C	GNU C compiler
505.mcf_r	C	Route planning
520.omnetpp_r	C++	Discrete event simulation
523.xalancmbk_r	C++	XML to HTML conversion
525.x264_r	C	Video compression
531.deepsjeng_r	C++	AI: $\alpha - \beta$ search (Chess)
541.leela_r	C++	AI: Monte Carlo search (Go)
548.exchange2_r	Fortran	AI: recursive solution generator (Sudoku)
557.xz_r	C	General data compression

SPECrate 2017 Integer

600.perlbench_s	C	Perl interpreter
602.gcc_s	C	GNU C compiler
605.mcf_s	C	Route planning
620.omnetpp_s	C++	Discrete event simulation
623.xalancmbk_s	C++	XML to HTML conversion
625.x264_s	C	Video compression
631.deepsjeng_s	C++	AI: $\alpha - \beta$ search (Chess)
641.leela_s	C++	AI: Monte Carlo search (Go)
648.exchange2_s	Fortran	AI: recursive solution generator (Sudoku)
657.xz_s	C	General data compression

SPECspeed 2017 Floating Point

503.bwaves_r	Fortran	Explosion modelling
507.cactuBSSN_r	C++,C,Fortran	Physics: relativity
508.namd_r	C++	Molecular dynamics
510.parest_r	C++	Biomedical imaging
511.povray_r	C++,C	Ray tracing
519.lbm_r	C	Fluid dynamics
521.wrf_r	Fortran,C	Weather forecasting
526.blender_r	C++,C	3D rendering and animation
527.cam4_r	Fortran,C	Atmosphere modeling
538.imagick_r	C	Image manipulation
544.nab_r	C	Molecular dynamics
549.fotonik3d_r	Fortran	Computational Electromagnetics
554.roms_r	Fortran	Regional ocean modeling

SPECrate 2017 Floating Point

603.bwaves_s	Fortran	Explosion modelling
607.cactuBSSN_s	C++,C,Fortran	Physics: relativity
619.lbm_s	C	Fluid dynamics
621.wrf_s	Fortran,C	Weather forecasting
627.cam4_s	Fortran,C	Atmosphere modeling
628.pop2_s	Fortran,C	Wide-scale ocena modeling
638.imagick_s	C	Image manipulation
644.nab_s	C	Molecular dynamics
649.fotonik3d_s	Fortran	Computational Electromagnetics
654.roms_s	Fortran	Regional ocean modeling

Transaction benchmarks

- Database performance

- TPC-A

- Test the interaction with an ATM (6 requests per minute)
 - 1 TPS means that 10 ATMs are concurrently asking for data and results are received within 2 s (90 % probability)

- TPC-B

- Similar to TPC-A, but direct connection, no slow network

- TPC-C

- Complex, transactions are requests, payments, questions, with a certain percentage of faults requiring automatic correction

Network benchmarks

- netperf
- iperf
- End to end measurements
- Be aware, what you are actually measuring in the network
 - Not so difficult if within a parallel computer interconnect

Own benchmarks

- Concrete (specific) requirements
- Important parameters:
 - What to test
 - How long to test
 - Memory requirements
- Benchmark types
 - Single stream (repetition)
 - Throughput (benchmark stone wall)

Checks

- Mandatory part of any benchmarking activity
 - Are we actually measuring what we want to?
- Potential influencers:
 - Used compiler optimization
 - Memory size
 - Other process served by the operating system
- What must be controlled explicitly
 - CPU time and wall clock time
 - **Results!**
 - Comparison with “a known” standard