

Performance measurements of computer systems: tools and analysis

M2R PDES

Jean-Marc Vincent and Arnaud Legrand

Laboratory LIG
MESCAL Project
Universities of Grenoble
{Jean-Marc.Vincent,Arnaud.Legrand}@imag.fr

October 21, 2011

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

1 Introduction, Definitions, Classifications

- Performance Metric
- Finding Bottlenecks
- Monitors & Measurements

2 Monitoring Examples

- Measuring Time: Practical Considerations
- Sequential Program Execution Monitoring: Profiling
- “API-based” Monitoring Examples
- Indirect Metrics

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

Metrics are criteria to **compare the performances** of a system. In general, the metrics are related to **speed, accuracy, reliability** and **availability** of services.

The basic characteristics of a computer system that we typically need to measure are:

- ▶ a **count** of how many times an event occurs,
- ▶ the **duration** of some time interval, and
- ▶ the **size** of some parameter.

From these types of measured values, we can derive the actual value that we wish to use to describe the system: the **performance metric**.

Performance Metric Characterization

Reliability A system A always outperforms a system $B \Leftrightarrow$ the performance metric indicates that A always outperforms B .

Repeatability The same value of the metric is measured each time the same experiments is performed.

Consistency Units of the metrics and its precise definition are the same across different systems and different configurations of the same system.

Linearity The value of the metric should be linearly proportional to the actual performance of the machine.

Easiness of measurement If a metric is hard to measure, it is unlikely anyone will actually use it. Moreover it is more likely to be incorrectly determined.

Independence Metrics should not be defined to favor particular systems.

Many metrics do not fulfill these requirements.

- ▶ Time between the start and the end of an operation
 - ▶ Also called running time, elapsed time, wall-clock time, response time, latency, execution time, ...
 - ▶ Most straightforward measure: “my program takes 12.5s on a Pentium 3.5GHz”
 - ▶ Can be normalized to some reference time
- ▶ Must be measured on a “dedicated” machine

Used often so that performance can be **independent** on the “size” of the application (e.g., compressing a 1MB file takes 1 minute. compressing a 2MB file takes 2 minutes \rightsquigarrow the performance is the same).

MIPS Millions of instructions / sec = $\frac{\text{instruction count}}{\text{execution time} \times 10^6} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}$.

But Instructions Set Architectures are not equivalent

- ▶ 1 CISC instruction = many RISC instructions
- ▶ Programs use different instruction mixes
- ▶ May be ok for same program on same architectures

MFlops Millions of floating point operations /sec

- ▶ Very popular, but often misleading
- ▶ e.g., A high MFlops rate in a stupid algorithm could have poor application performance

Application-specific ▶ Millions of frames rendered per second

- ▶ Millions of amino-acid compared per second
- ▶ Millions of HTTP requests served per seconds

Application-specific metrics are often preferable and others may be misleading

“Peak” Performance?

Resource vendors always talk about **peak performance** rate

- ▶ Computed based on specifications of the machine
- ▶ For instance:
 - ▶ I build a machine with 2 floating point units
 - ▶ Each unit can do an operation in 2 cycles
 - ▶ My CPU is at 1GHz
 - ▶ Therefore I have a $1 * 2 / 2 = 1$ GFlops Machine
- ▶ Problem:
 - ▶ In real code you will never be able to use the two floating point units constantly
 - ▶ Data needs to come from memory and cause the floating point units to be idle

Typically, real code achieves only an (often small) fraction of the peak performance

- ▶ Since many performance metrics turn out to be misleading, people have designed **benchmarks**
- ▶ Example: SPEC Benchmark
 - ▶ Integer benchmark
 - ▶ Floating point benchmark
- ▶ These benchmarks are typically a collection of several codes that come from “real-world software”
- ▶ The question “what is a good benchmark” is difficult
 - ▶ A benchmark is representative of a given **workload**.
 - ▶ If the benchmarks do not correspond to what you’ll do with the computer, then the benchmark results are not relevant to you
- ▶ Other typical benchmarks
 - ▶ Livermore loops, NAS kernels, LINPACK, ...
 - ▶ SPEC SFS, SPECWeb, ...

How About GHz?

- ▶ This is often the way in which people say that a computer is better than another
 - ▶ More instruction per seconds for higher clock rate
- ▶ Faces the same problems as MIPS

Processor	Clock Rate	SPEC FP2000 Benchmark
IBM Power3	450 MHz	434
Intel PIII	1.4 GHz	456
Intel P4	2.4GHz	833
Itanium-2	1.0GHz	1356

- ▶ But usable within a specific architecture

1 Introduction, Definitions, Classifications

- Performance Metric
- **Finding Bottlenecks**
- Monitors & Measurements

2 Monitoring Examples

- Measuring Time: Practical Considerations
- Sequential Program Execution Monitoring: Profiling
- “API-based” Monitoring Examples
- Indirect Metrics

Why is Performance Poor?

Performance is poor because the code suffers from a performance **bottleneck**

Definition:

- ▶ An application runs on a platform that has many components (CPU, Memory, Operating System, Network, Hard Drive, Video Card, etc.)
- ▶ Pick a component and make it faster
- ▶ If the application performance increases, that component was the bottleneck!

There are two may approaches to remove a bottleneck:

Brute force Hardware Upgrade

- ▶ Is sometimes necessary
- ▶ But can only get you so far and may be very costly (e.g., memory technology)

Modify the code ▶ The bottleneck is there because the code uses a “resource” heavily or in non-intelligent manner

- ▶ We will learn techniques to alleviate bottlenecks at the software level

Identifying a Bottleneck

- ▶ It can be difficult
 - ▶ You're not going to change the memory bus just to see what happens to the application
 - ▶ But you can run the code on a different machine and see what happens
- ▶ One Approach
 - ▶ Know/discover the characteristics of the machine
 - ▶ Instrument the code with `gettimeofday` everywhere
 - ▶ Observe the application execution on the machine
 - ▶ Tinker with the code
 - ▶ Run the application again
 - ▶ Repeat
 - ▶ Reason about what the bottleneck is

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

A **Monitor** is a tool to observe the activities on a system. In general, a monitor:

- ① makes measurements on the system (**Observation**)
- ② collects performance statistics (**Collection**),
- ③ analyzes the data (**Analysis**),
- ④ displays results (**Presentation**).

Why do we want a monitor ?

- ▶ A programmer wants to find frequently used segments of a program to optimize them.
- ▶ A system administrator wants to measure resource utilization to find performance **bottlenecks**.
- ▶ A system administrator wants to tune the system and measure the **impact** of system parameters **modifications** on the system performance.
- ▶ An analyst wants to characterize the **workload**. Results may be used for capacity planning and for creating test workloads.
- ▶ An analyst wants to find **model** parameters, to validate models or to develop inputs for models.

Monitor Terminology

- Event** A change in the system state (context switch, seek on a disk, packet arrival, ...).
- Trace** Log of events usually including the time of the events, their type and other important parameters.
- Overhead** Measurement generally induce slight perturbations and consume system resources (CPU, storage, ...).
- Domain** The set of activities observable by the monitor (CPU time, number of bytes sent on a network card, ...).
- Input Rate** The maximum frequency of events that a monitor can correctly observe. One generally distinguish between burst rate and sustained rate.
- Resolution** Coarseness of the information observed.
- Input Width** The number of bits of information recorded on an event.
- Portability** Amount of system modifications required to implement the monitor.

The different types of metrics that an analyst may wish to measure can be classified into the following categories:

Event-count metrics Simply count the number of times a specific event occurs (e.g., number of page faults, number of disk I/O made by a program).

Secondary-event metrics These types of metrics record the values of some secondary parameters whenever a given event occurs (e.g., the average message size).

Profiles A profile is an aggregate metric used to characterize the overall behavior of an application program or of an entire system.

Observation is commonly performed with three mechanisms:

Implicit spying it is sometimes possible to spy the system without really interfering with it (listening on a local Ethernet bus or in wireless environments). Thus, there is almost no impact on the performance of the system being monitored. Implicit-spying is generally used with filters because many observed data are not interesting.

Explicit Instrumenting By incorporating trace points, probe points, hooks or counters, additional information to implicit-spying can be obtained. Some systems offer an API to incorporate such hooks or exports the values of internal counters.

Probing Making “feeler” requests on the system to sense its current performance.

Some activities can be observed by only one of the three mechanisms.

Activation Mechanisms

- Event-driven** Record the information whenever a given event occurs. Generally done with counters. The overhead is thus proportional to the frequency of events.
- Tracing** Similar to event-driven strategies, except that parts of the system state are also recorded. This is thus even more time-consuming and also also requires much more storage.
- Sampling** Recording of information occurs periodically. The overhead and the resolution of the sampling can thus be adjusted. This strategies produces a statistical summary of the overall behavior of the system. Events that occurs infrequently may be completely missed. Results may also be different from one run to the other.
- Indirect** An indirect measurement must be used been the metric that is to be determined is not directly accessible (e.g., for portability reasons or for practical reasons). In this case, one must find another metric that can be measured directly, from which one can deduce or derive the desired performance metric.
Generally, a **model** of the system is underlying the deduced metric and the quality of this model is fundamental.

The collection mechanism highly depends on whether we are working on a **on-line** monitor (system state is displayed during monitoring) or on a **batch** monitor (system state is stored for later analysis).

Most data need to be recorded in buffers, hence **buffer management** issues:

- ▶ buffer size is a function of input rate, input width and emptying rate,
- ▶ larger number of buffers allows to cope with variations in filling and emptying rates,
- ▶ buffer overflow management and tracking,
- ▶ data compression, on-line analysis,

Abnormal events should also be monitored and even handled at higher priority (low probability \rightsquigarrow low overhead, possibility to take preventive action before the system becomes unavailable).

- ▶ The problem of communication between m collectors and n observers often arises.
- ▶ We generally resort to hierarchy of collectors/observers for a better scalability. This intensifies all previous buffer management issues.
- ▶ When collecting data from several observers, clock synchronization often becomes an important issue. The tolerance or maximum allowed clock skew is often related to the round-trip delay. The larger the system, the more problematic clock skews.

This layer is closely tied to the applications for which the monitor is used (performance monitoring, configuration monitoring, fault monitoring, . . .).

- ▶ Presentation **frequency** (for on-line monitors).
- ▶ Hierarchical representation/**aggregation** (space/time/states/values).
- ▶ **Alarm** mode (thresholds, abnormal events).

Measurement Induces Perturbations

- ▶ The system resources consumed by the measurement tool itself as it collects data will strongly affect how much **perturbation** the tool will cause in the system.
- ▶ Tracing produces the highest level of perturbation (both CPU and disk are used), in particular on time measurements, spatial and temporal memory access (cache flush, different paging, ...), or on system response time (and thus on **workload characterization**).
- ▶ The largest the overhead, the more likely the **system behavior** will be **modified**.

Measuring a system alters it

Remain alert to how these perturbations may bias your measurements and, ultimately, the conclusions you are able to draw from your experiments.

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

Try to find the main characteristics, advantages and drawbacks of the following monitors.

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

Measuring time by hand?

- ▶ One possibility would be to do this by just “looking” at a clock, launching the program, “looking” at the clock again when the program terminates
- ▶ This of course has some drawbacks
 - ▶ Poor resolution
 - ▶ Requires the user’s attention
- ▶ Therefore operating systems provide ways to time programs automatically
- ▶ UNIX provide the `time` command

The UNIX time Command

- ▶ You can put `time` in front of any UNIX command you invoke
- ▶ When the invoked command completes, time prints out timing (and other) information

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

- ▶ 4.17 seconds of user time
- ▶ 4.34 seconds of system time
- ▶ 2 minutes and 55.85 seconds of wall-clock time
- ▶ 4% of CPU was used
- ▶ 0+0k memory used (text + data)
- ▶ 0 input, 0 output output (file system I/O)
- ▶ 1344 minor pagefaults and 0 swaps

User, System, Wall-Clock?

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

- ▶ User Time: time that the code spends executing user code (i.e., non system calls)

User, System, Wall-Clock?

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

- ▶ User Time: time that the code spends executing user code (i.e., non system calls)
- ▶ System Time: time that the code spends executing system calls

User, System, Wall-Clock?

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

- ▶ User Time: time that the code spends executing user code (i.e., non system calls)
- ▶ System Time: time that the code spends executing system calls
- ▶ Wall-Clock Time: time from start to end

User, System, Wall-Clock?

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

- ▶ User Time: time that the code spends executing user code (i.e., non system calls)
- ▶ System Time: time that the code spends executing system calls
- ▶ Wall-Clock Time: time from start to end
- ▶ Wall-Clock \geq User + System. Why?

User, System, Wall-Clock?

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

- ▶ User Time: time that the code spends executing user code (i.e., non system calls)
- ▶ System Time: time that the code spends executing system calls
- ▶ Wall-Clock Time: time from start to end
- ▶ Wall-Clock \geq User + System. Why?
 - ▶ because the process can be suspended by the O/S due to contention for the CPU by other processes

User, System, Wall-Clock?

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

- ▶ User Time: time that the code spends executing user code (i.e., non system calls)
- ▶ System Time: time that the code spends executing system calls
- ▶ Wall-Clock Time: time from start to end
- ▶ Wall-Clock \geq User + System. Why?
 - ▶ because the process can be suspended by the O/S due to contention for the CPU by other processes
 - ▶ because the process can be blocked waiting for I/O

- ▶ It's interesting to know what the user time and the system time are
 - ▶ for instance, if the system time is really high, it may be that the code does too many calls to `malloc()`, for instance
 - ▶ But one would really need more information to fix the code (not always clear which system calls may be responsible for the high system time)
- ▶ Wall-clock - system - user \simeq I/O + suspended
 - ▶ If the system is **dedicated**, suspended $\simeq 0$
 - ▶ Therefore one can estimate the cost of I/O
 - ▶ If I/O is really high, one may want to look at reducing I/O or doing I/O better
- ▶ Therefore, time can give us insight into bottlenecks and gives us wall-clock time
- ▶ Measurements should be done on **dedicated systems**
- ▶ `time` relies on `times(2)`, `getrusage(2)` and `clock(3)`.

- ▶ Measuring the performance of a code must be done on a “quiet”, “unloaded” machine (the machine only runs the standard O/S processes)
- ▶ The machine must be dedicated
 - ▶ No other user can start a process
 - ▶ The user measuring the performance only runs the minimum amount of processes (basically, a shell)
- ▶ Nevertheless, one should always present measurement results as averages over several experiments (because the (small) load imposed by the O/S is not deterministic)

- ▶ The `time` command has poor resolution
 - ▶ “Only” milliseconds
 - ▶ Sometimes we want a higher precision, especially if our performance improvements are in the 1-2% range
- ▶ `time` times the whole code
 - ▶ Sometimes we're only interested in timing some part of the code, for instance the one that we are trying to optimize
 - ▶ Sometimes we want to compare the execution time of different sections of the code

- ▶ `gettimeofday` from the standard C library
- ▶ Measures the number of microseconds since midnight, Jan 1st 1970, expressed in seconds and microseconds

```
struct timeval start;
...
gettimeofday(&tv, NULL);
printf("%ld,%ld\n", start.tv_sec, start.tv_usec);
```

- ▶ Can be used to time sections of code
 - ▶ Call `gettimeofday` at beginning of section
 - ▶ Call `gettimeofday` at end of section
 - ▶ Compute the time elapsed in microseconds:
 $(\text{end.tv_sec} * 1000000.0 + \text{end.tv_usec} - \text{start.tv_sec} * 1000000.0 - \text{start.tv_usec}) / 1000000.0$

- ▶ `ntp_gettime()` (Internet RFC 1589)
 - ▶ Sort of like `gettimeofday`, but reports estimated error on time measurement
 - ▶ Not available for all systems
 - ▶ Part of the GNU C Library
- ▶ Java: `System.currentTimeMillis()`
 - ▶ Known to have resolution problems, with resolution higher than 1 millisecond!
 - ▶ Solution: use a native interface to a better timer
- ▶ Java: `System.nanoTime()`
 - ▶ Added in J2SE 5.0
 - ▶ Probably not accurate at the nanosecond level
- ▶ Tons of “high precision timing in Java” on the Web

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

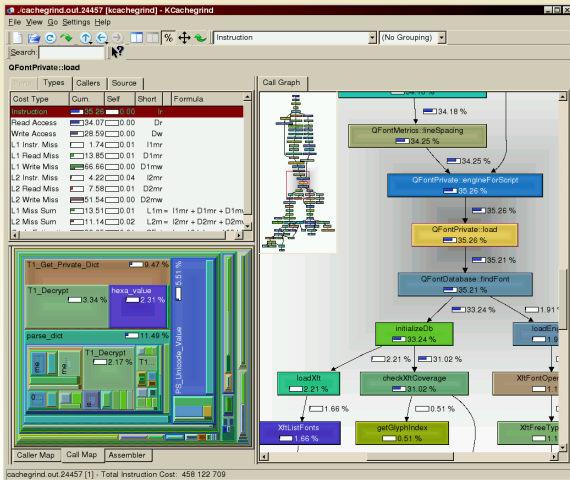
- ▶ A **profiler** is a tool that monitors the execution of a program and that reports the amount of time spent in different functions
- ▶ Useful to identify the expensive functions
- ▶ Profiling cycle
 - ▶ Compile the code with the profiler
 - ▶ Run the code
 - ▶ Identify the most expensive function
 - ▶ Optimize that function (i.e. call it less often if possible **or** make it faster)
 - ▶ Repeat until you can't think of any ways to further optimize the most expensive function

- ▶ Compile your code using `gcc` with the `-pg` option
- ▶ Run your code until completion
- ▶ Then run `gprof` with your program's name as single command-line argument
- ▶ Example: `gcc -pg prog.c -o prog; ./prog gprof prog > profile_file`
- ▶ The output file contains all profiling information (amount and fraction of time spent in which function)

Function/Method	Count	Total (s)	%	Self (s)	Total ms/call	Self ms/call
CProfileInfo::CProfileInfo(void)	69	0.02	0 0	0	0	0
CProfileViewItem::CProfileViewItem(QListView *, CProfileInfo *)	157	0.02	0 0	0.03	0	0
CProfileViewItem::CProfileViewItem(QListViewItem *, CProfileInfo *)	224	0.02	0 0	0	0	0
KAboutData::~KAboutData(void)	1	0.02	0 0	0	0	0
KProfTopLevel::KProfTopLevel(int, QWidget *, char const *)	1	0.02	0 0	58.87	0	0
KProfTopLevel::setupActions(void)	1	0.02	0 0	0	0	0
KProfWidget::KProfWidget(QWidget *, char const *)	1	0.02	0 0	45.94	0	0
KProfWidget::applySettings(void)	2	0.02	0 0	7.18	0	0
KProfWidget::loadSettings(void)	1	0.02	0 0	4.31	0	0
KProfWidget::prepareProfileView(QListView *, bool)	3	0.02	0 0	10.05	0	0
QString::~QString(void)	6965	0.01	50 0.01	1.44	1.44	0
QShared::deref(void)	28621	0.02	0 0	0	0	0
QVector<CProfileInfo>::QVector(void)	1	0.02	0 0	0	0	0
KProfTopLevel::setupActions(void)	1	0.02	0 0	0	0	0
KProfTopLevel::staticMetaObject(void)	107	0.02	0 0	0	0	0
KProfWidget::KProfWidget(QWidget *, char const *)	1	0.02	0 0	45.94	0	0
KProfWidget::applySettings(void)	2	0.02	0 0	7.18	0	0
KProfWidget::fillFlatProfileList(void)	1	0.02	0 0	1.89	0	0
KProfWidget::fillHierProfileList(void)	1	0.02	0 0	1.89	0	0
KProfWidget::fillHierarchy(CProfileViewItem *, CProfileInfo *, QAr...	69	0.02	0 0	0	0	0

- ▶ **Callgrind** is a tool that uses runtime code instrumentation framework of **Valgrind** for call-graph generation
- ▶ Valgrind is a kind of emulator or **virtual machine**.
 - ▶ It uses JIT (just-in-time) compilation techniques to translate x86 instructions to simpler form called ucode on which various tools can be executed.
 - ▶ The ucode processed by the tools is then translated back to the x86 instructions and executed on the host CPU.
- ▶ This way even shared libraries and dynamically loaded plugins can be analyzed but this kind of approach results with **huge slow down** (about 50 times for callgrind tool) of analyzed application and big memory consumption.

Data produced by callgrind can be loaded into KCachegrind tool for browsing the performance results.



Data produced by callgrind can be loaded into KCacheGrind tool for browsing the performance results.

Valgrind actually includes several tools:

- ▶ Memcheck (check the validity of every memory access)
- ▶ Cachegrind (memory access profiler/simulator)
- ▶ Callgrind (instruction profiler)
- ▶ Massif (heap profiler)
- ▶ Helgrind (a thread error detector)

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

```
top - 11:05:41 up 3 days, 1:01, 1 user, load average: 0.22, 0.23, 0.38
Tasks: 177 total, 2 running, 166 sleeping, 0 stopped, 9 zombie
Cpu(s): 5.8%us, 1.7%sy, 0.0%ni, 89.9%id, 0.0%wa, 0.0%hi, 2.7%si, 0.0%st
Mem: 1035156k total, 983544k used, 51612k free, 30320k buffers
Swap: 1984016k total, 67020k used, 1916996k free, 365876k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4571	alegrand	15	0	328m	178m	30m	R	7	17.7	162:10.23	firefox-bin
3794	gkrellmd	15	0	11844	1152	936	S	5	0.1	104:25.48	gkrellmd
4225	alegrand	16	0	73936	11m	3344	S	4	1.2	54:02.17	xnms2d
4210	alegrand	16	0	42732	14m	10m	S	2	1.5	20:26.84	gkrellm
3792	root	15	0	236m	176m	8972	S	1	17.5	141:55.30	Xorg
4211	alegrand	15	0	47424	27m	10m	S	0	2.7	3:59.48	esperanza
16982	alegrand	15	0	2360	1200	860	R	0	0.1	0:00.12	top
1	root	15	0	1944	652	556	S	0	0.1	0:01.55	init
2	root	RT	0	0	0	0	S	0	0.0	0:00.02	migration/0
3	root	34	19	0	0	0	S	0	0.0	0:00.03	ksoftirqd/0
6	root	10	-5	0	0	0	S	0	0.0	0:04.12	events/0
8	root	10	-5	0	0	0	S	0	0.0	0:00.00	khelper
9	root	10	-5	0	0	0	S	0	0.0	0:00.00	kthread
13	root	10	-5	0	0	0	S	0	0.0	0:00.04	kblockd/0
15	root	10	-5	0	0	0	S	0	0.0	0:05.86	kacpid
155	root	17	-5	0	0	0	S	0	0.0	0:00.04	kseriod
201	root	15	-5	0	0	0	S	0	0.0	0:07.06	kswapd0
202	root	13	-5	0	0	0	S	0	0.0	0:00.00	ai0/0
358	root	15	0	0	0	0	S	0	0.0	0:00.01	kirqd
704	root	10	-5	0	0	0	S	0	0.0	0:13.58	ata/0
706	root	10	-5	0	0	0	S	0	0.0	0:00.00	ata_aux
709	root	10	-5	0	0	0	S	0	0.0	0:00.04	scsi_ah_0
758	root	10	-5	0	0	0	S	0	0.0	0:00.04	khubd
773	root	10	-5	0	0	0	S	0	0.0	0:26.36	scsi_ah_1
1068	root	10	-5	0	0	0	S	0	0.0	0:05.91	kjournald
1256	root	15	-4	2720	1276	464	S	0	0.1	0:00.80	udev
1720	root	16	-5	0	0	0	S	0	0.0	0:00.00	kpsmouse
1756	root	17	-5	0	0	0	S	0	0.0	0:00.00	pccard
1837	root	10	-5	0	0	0	S	0	0.0	0:00.00	hda_codec
1840	root	10	-5	0	0	0	S	0	0.0	0:06.39	ipw3945/0
1842	root	10	-5	0	0	0	S	0	0.0	0:00.24	ipw3945/0

How does top get all these information?

It uses `/proc`!

```
kanza: $ ldd 'which top'  
linux-gate.so.1 libproc-3.2.7.so.5  
libncurses.so libc.so.6  
libdl.so.2 /lib/ld-linux.so.2
```

`/proc` is a way for the kernel to provide information about the status of entries in its process table.

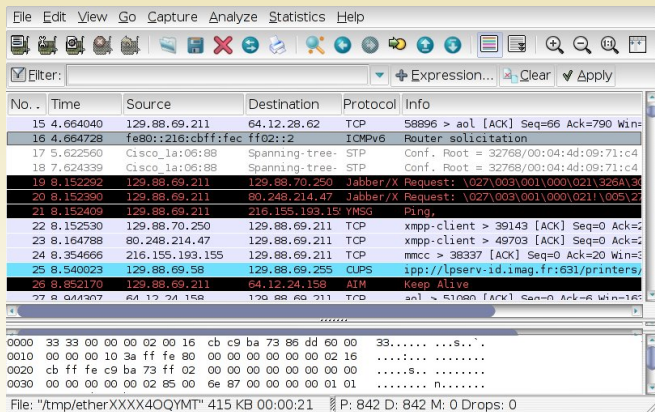
On systems where `/proc` is available, there is no need to instrument the kernel or the application to get these information. Measures are always but it doesn't mean that `top` does not induce perturbations. . . .

Other tools (e.g., `gkrellm`) rely on the same API.



Getting finer information

TCPdump is based on `libpcap` and enables to dump the traffic on a network card.



The screenshot shows the Wireshark interface with a packet capture list and packet details. The packet list table is as follows:

No.	Time	Source	Destination	Protocol	Info
15	4.664040	129.88.69.211	64.12.28.62	TCP	58896 > aol [ACK] Seq=66 Ack=790 Win=...
16	4.664728	fe80::216:cbff:fec	ff02::2	ICMPv6	Router solicitation
17	5.622560	Cisco_la:06:88	Spanning-tree-	STP	Conf. Root = 32768/00:04:4d:09:71:c4
18	7.624339	Cisco_la:06:88	Spanning-tree-	STP	Conf. Root = 32768/00:04:4d:09:71:c4
19	8.152292	129.88.69.211	129.88.70.250	Jabber/X	Request: \027\003\001\000\021\326A\3...
20	8.152390	129.88.69.211	80.248.214.47	Jabber/X	Request: \027\003\001\000\021!\005\27...
21	8.152409	129.88.69.211	216.155.193.15	YMSG	Ping,
22	8.152530	129.88.70.250	129.88.69.211	TCP	xmpp-client > 39143 [ACK] Seq=0 Ack=...
23	8.164788	80.248.214.47	129.88.69.211	TCP	xmpp-client > 49703 [ACK] Seq=0 Ack=...
24	8.354666	216.155.193.155	129.88.69.211	TCP	mmcc > 38337 [ACK] Seq=0 Ack=20 Win=...
25	8.540023	129.88.69.58	129.88.69.255	CUPS	ipp://lpserve-id.imag.fr:631/printers/...
26	8.852170	129.88.69.211	64.12.24.158	AIM	Keep Alive
27	8.944307	64.12.24.158	129.88.69.211	TCP	aol > 51090 [ACK] Seq=0 Ack=6 Win=16...

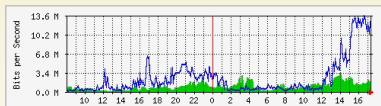
The packet details pane for packet 25 shows the following structure:

```
0000 33 33 00 00 02 00 16 cb c9 ba 73 86 dd 60 00 33.....s..
0010 00 00 00 10 3a ff fe 80 00 00 00 00 00 02 16 .....
0020 cb ff fe c9 ba 73 ff 02 00 00 00 00 00 00 00 .....s..
0030 00 00 00 00 00 02 85 00 6e 87 00 00 00 01 01 .....n.....
```

File: "/tmp/etherXXXX4OQYMT" 415 KB 00:00:21 P: 842 D: 842 M: 0 Drops: 0

`/proc` is rather common but accessing such information requires specific access permissions. Such library does not work on high performance cards such as MyriNet, InfiniBand, ...

- ▶ The Simple Network Management Protocol (SNMP) is an application layer protocol that facilitates the exchange of management information between network devices. It is part of the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol suite. SNMP enables network administrators to manage network performance, find and solve network problems, and plan for network growth.
- ▶ Many tools build upon SNMP to gather information on routers



- ▶ SNMP can even be used to build maps of the network. However, SNMP requires specific access permissions and is often closed for network security reasons.

PAPI Performance Application Programming Interface

PAPI is a tool developed by UTK that provides a portable access to hardware performance counters available on most modern microprocessors (AMD Opteron, CRAY XT Series, IBM POWER, IBM Cell Intel Pentium, Core2, i7, Atom, . . . , recently CUDA).

A few HPC tools

PAPI Performance Application Programming Interface

mpiP MPI profiling

- ▶ mpiP is a link-time library (it gathers MPI information through the MPI profiling layer)
- ▶ It only collects statistical information about MPI functions
- ▶ All the information captured by mpiP is task-local

```
sleeptime = 10;
MPI_Init (&argc, &argv);
MPI_Comm_size (comm, &nprocs);
MPI_Comm_rank (comm, &rank);
MPI_Barrier (comm);
if (rank == 0) sleep (sleeptime);
MPI_Barrier (comm);
MPI_Finalize ();
```

Task	AppTime	MPITime	MPI%
0	10	0.000243	0.00
1	10	10	99.92
2	10	10	99.92
3	10	10	99.92
*	40	30	74.94

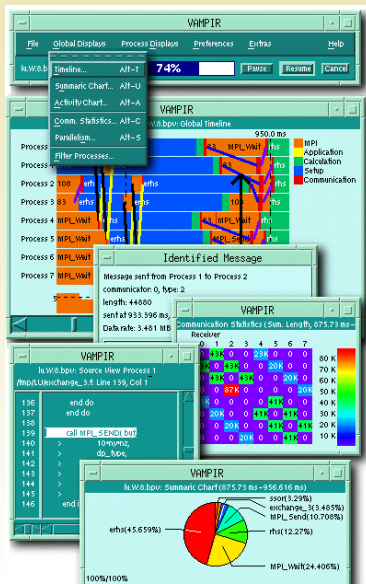
A few HPC tools

PAPI Performance Application Programming Interface

mpiP MPI profiling

vaMPIr MPI tracing

- ▶ generate **traces** (i.e. not just collect statistics) of MPI calls
- ▶ These traces can then be visualized and used in different ways.



- ▶ An API to access monitoring information is often available but not always. . .
- ▶ Even when these monitors are “built-in”, they are generally low-level and building a usable high-level monitoring tool requires a lots of work on:
 - ▶ Sampling
 - ▶ Collection
 - ▶ Analysis and Presentation
- ▶ When such an API is not available you can:
 - ▶ either design a low-level monitor if possible,
 - ▶ to try to evaluate the metric you are interested in an other way.

- 1 Introduction, Definitions, Classifications
 - Performance Metric
 - Finding Bottlenecks
 - Monitors & Measurements
- 2 Monitoring Examples
 - Measuring Time: Practical Considerations
 - Sequential Program Execution Monitoring: Profiling
 - “API-based” Monitoring Examples
 - Indirect Metrics

- ▶ **ATLAS** (Automatically Tuned Linear Algebra Software) is an approach for the automatic generation and optimization of numerical software (BLAS and a subset of the linear algebra routines in the LAPACK library).
- ▶ To produce such kernels, ATLAS needs to know the number of cache levels, their respective sizes, whether the processor has the ability to perform additions and multiplications at the same time, whether it can make use of vector registers or specific instruction sets (e.g., 3dnow, SSE1, or SSE2 extensions). . .
- ▶ There is no portable API providing such information, therefore ATLAS runs some probes to guess these values.

NWS The Network Weather Service is the de facto standard of the emerging Grid community to monitor the system availability. It provides high-level metrics to help applications and schedulers. It also provides trends thanks to a set of statistical forecasters.

- ▶ Available CPU share for a new process $((\text{system} + \text{user}) / (\text{total}))$: due to the process priorities and other scheduling tricks, this value is hard to guess from the `/proc` values without actually probing. As probes are intrusive, NWS uses `/proc` values and uses a correcting factor based on regular probes.
- ▶ Available bandwidth between two hosts: how much bandwidth would get an application using a single standard socket ? Active probes are performed.

Pathchar What about the capacity of network links between two hosts? Pathchar infers the characteristics of links along an Internet path by sending series of probes with varying values of TTL and of size and using statistical analysis.

Peer-to-peer systems:

- ▶ Having good evaluations of the current number of peers is a crucial problem and an active research domain.
- ▶ “Probabilistic games” give good results.

Main Issue

All the previous approaches rely on a model of the system and on parameters estimation based on the expected model prediction. When the model is incorrect, the estimation is likely to be incorrect as well.

- ▶ **Do not reinvent the wheel**: do not build your own monitoring infrastructure unless you are absolutely certain there is no other choice.
Instead learn to use already existing tools.
- ▶ Make sure that you **understand** how such tools work to be aware of their limitations and impact on the measurements you want to perform.



R. K. Jain.

The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.

John Wiley & Sons Canada, Ltd., 1 edition, 1991.



David J. Lilja.

Measuring Computer Performance: A Practitioner's Guide - David J. Lilja - Hardcover.

Cambridge University Press, 2000.