Gathering Measurements
CS 239
Experimental Methodologies for System Software
Peter Reiher
April 26, 2007

Outline

- Monitors
- Tools for measurement
- Applying workloads to systems
- Common mistakes in benchmarking

Monitors

- A monitor is a tool used to observe system activity
- Proper use of monitors is key to performance analysis
- Also useful for other system observation purposes

Classifications of Monitors

- Hardware vs. software monitors
- Event-driven vs. sampling monitors
- On-line vs. batch monitors

Hardware Vs. Software Monitors

- Hardware monitors used primarily by hardware designers
  - Requires substantial knowledge of hardware details
  - VLSI limits monitoring possibilities
- Software monitors used (mostly) by everyone else

Event-Driven Vs. Sampling Monitors

- Event-driven monitors notice every time a particular type of event occurs
  - Ideal for rare events
  - Require low per-invocation overheads
- Sampling monitors check the state of the system periodically
  - Good for frequent events
  - Can afford higher overheads

On-Line Vs. Batch Monitors

- On-line monitors can display their information continuously
  – Or, at least, frequently
- Batch monitors save it for later
  – Usually using separate analysis procedures

Issues in Monitor Design

- Activation mechanism
- Buffer issues
- Data compression/analysis
- Enabling/disabling monitors
- Priority issues
- Abnormal events monitoring

Activation Mechanism

- When do you collect the data?

– When an interesting event occurs, trap to data collection routine

– Analyze every step taken by system

– Go to data collection routine when timer expires

Buffer Issues

- Buffer size
  – Big enough to avoid frequent disk writes
  – Small enough to make disk writes cheap
- Number of buffers
  – At least two, typically
  – One to fill up, one to record
- Buffer overflow
  – Overwrite old data you haven't recorded
  – Or lose new data you don't have room for

Data Compression or Analysis

- Data can be literally compressed
- Or can be reduced to a summary form
- Both methods save space for holding data
- But at the cost of extra overhead in gathering it
- Sometimes can use idle time for this
  – But might be better spent dumping data to disk

Enabling/Disabling Monitors

- Most system monitors have some overhead
- So users should be able to turn them off, if high performance is required
- Not necessary if overhead is truly trivial
- Or if purpose of system is primarily gathering data
  – As is case with many research systems

Priority of Monitor

- How high a priority should the monitor's operations have?
- Again, trading off performance impact against timely and complete data gathering
- Not always a simple question

Monitoring Abnormal Events

- Sometimes, failures and errors are most important thing to observe
- Can requires special attention
  – System may not be operating very well at the time of the failure

Monitoring Distributed Systems

- Monitoring a distributed system is like designing a distributed system
- Must deal with
  – Distributed state
  – Unsynchronized clocks
  – Partial failures

Layered View of Distributed Monitor

Management	Make system changes, as necessary
Console	Control the overall system
Interpretation	Decide what the results mean
Presentation	Present your results
Analysis	Analyze what you've stored
Collection	Store what you've seen for later
Observation	Watch what happens

The Observation Layer

- Layer that actually gathers the data
- Implicit spying - watching what other sites do without disturbing the activity
- Explicit instrumentation - inserting code to monitor activities
- Probing - making feeler requests into system to discover what's happening

The Collection Layer

- Data can be collected at one or several points in the distributed system
- How does the data get from observer to collector (if not co-located)?
  – Advertising - observers send it out, collectors listen and grab it
  – Soliciting - collectors ask observers to send it
- Clock issues can be key, here

The Analysis Layer

- In distributed system, may be more feasible to analyze on the fly
- Can sometimes dedicate one (or more) machines to analysis
- Often requires gathering all data to one point, though

Tools and Methods For Software Measurement

- OK, so how do I actually measure a piece of software?
- What are the practical tools and methods available to me?
- How do I get my damn project done?

Tools For Software Measurement

- Code instrumentation
- Tracing packages
- System-provided metrics and utilities
- Profiling

Code Instrumentation

- Adding monitoring code to the system under study
- Basically, just add the code that does what you want

Advantages of Code Instrumentation

+ Usually the most direct way to gather data
+ Complete flexibility of where to insert monitoring code
+ Strong control over costs of monitoring
+ Resulting measurements always available

Disadvantages of Instrumenting the Code

– Requires access to the source
– Requires strong knowledge of the design and many details of the code
– Requires recompilation to change monitoring facility
– If overdone, strong potential to affect performance

Typical Types of Instrumentation

- Counters and accumulators
  - Cheap and fast
  - But low level of detail
- Logs
  - More detail
  - But more costly
  - Require occasional dumping or digesting
- Timers
  - To determine elapsed time for operations
  - Typically using OS-provided system calls

Counters

- Useful if number of times an event occurs is of interest
- Can be used to accumulate totals
  - E.g., total bytes read by file system
- In modern systems, make them wide enough so they won't overflow

Examples of Counters

- Count number of times a network protocol transmits packets
- Count number of times programs are swapped out due to exceeding their time slices
- Count number of incoming requests to a Web server

Logs

- Can log arbitrarily complex data about an event
- But more complex data takes more space
- Typically, log data into a reserved buffer
- When full, request for buffer to be written to disk
  - Often want a second buffer to gather data while awaiting disk write

Designing a Log Entry

- What form should a log entry take?
- Designing for compactness vs. human readability
  - Former better for most purposes
  - Latter useful for system debugging
  - Make sure no important information is lost in compacting the log entry

Timers

- Many OSs provide system calls that start and stop timers
- Allowing you to time how long things took
- Usually, only elapsed time measurable
  - Not necessarily time spent running particular process
- So care required to capture real meaning of timings

Tracing Packages

- Allow dynamic monitoring of code that doesn't have built-in monitors
- Basically, augment the code to call monitoring routines when desired
- Akin to debuggers
- Typically allow counters and some forms of logging

How Do Tracing Packages Work?

- Much like debuggers -
  – Attach them to running programs
  – Use commands in the tracing packages to associate data gathering with particular points in the programs
  – Replace normal code at that point in program with preliminary calls to data gathering code

Advantages of Tracing Packages

+ Allows pretty arbitrary insertion of monitoring code
+ Doesn't require recompilation in order to instrument code
+ Tremendous flexibility at measurement time
+ No instrumentation overhead when you're not using it

Disadvantages of Tracing Packages

– Somewhat higher overheads than building instrumentation into code
– Really requires access to source for effective use
– And requires deep understanding of the internals of the code
– Only produces data when special package is used
– Usually specific to particular systems

System Provided Metrics and Utilities

- Many operating systems provide users access to some metrics
- Most operating systems also keep some form of accounting logs
- Lots of information can be gathered this way

What a Typical System Provides

- Timing tools
- Process state tools
- System state tools
- OS accounting logs
- Logs for important systems programs

Timing Tools

- Tools that time the execution of a process
- Often several different times are provided
- E.g., Unix `time` command provides system time, user time, and elapsed time
- Various components of the times provided may depend on other system activities
  - So just calling `time` on a command may not tell the whole story

Process State Tools

- Many systems have ways for users to find out about the state of their processes
  - E.g., `ps` in Unix/Linux systems
- Typically provide information about:
  - Time spent running process so far
  - Size of process
  - Status of process
  - Priority of process
  - I/O history of process

Using Process State Tools

- Typically, you can't monitor process state continuously
  - Updates not provided every time things change
- You get snapshots on demand
  - So most useful for sampling monitors

System State Tools

- Many systems allow some users to examine their internal state
  - E.g., virtual memory statistics
  - Or length of various queues
- Often available only to privileged users
- Typically, understanding them requires substantial expertise
  - And they are only useful for specific purposes

OS Accounting Logs

- Many operating systems maintain logs of significant events
  - Based either on event-driven or sampling monitors
- Examples:
  - logins
  - full file systems
  - device failures

System Accounting Logs

- Often, non-OS systems programs keep logs
- E.g., mail programs
- Usually only useful for monitoring those programs
- But sometimes can provide indirect information
  - E.g., notice of failure to open connection to name server suggests machine failure

Applying Test Loads to Systems

- Designing test loads
- Tools for applying test loads

Test Load Design

- As discussed earlier, most experiments require applying test loads to the system
- General characteristics of test loads already discussed
- How do we design test loads?

Types of Test Loads

- Real users
- Traces
- Load generation programs

Loads Caused by Real Users

- Put real people in front of your system
- Two choices:
  - Have them run pre-arranged set of tasks
  - Have them do what they'd normally do
- Always difficult to test this way
  - Labor-intensive
  - Impossible to reproduce a given load
  - Load is subject to many external influences
- Highly realistic, though

Traces

- Collect a set of commands/accesses issued to the system under test (or a similar system)
- Replay them against your system
- Some traces of common activities are available from others (e.g., file accesses)
  - But often don't contain everything you need

Running Traces

- Need process that:
  - reads trace
  - keeps track of progress through trace
  - issues commands from the trace when appropriate
- Process must be reasonably accurate in timing
  - But must also have little performance impact
- If trace is large, can't keep it all in main memory
  - But be careful of disk overheads

Load Generation Programs

- Create a model for the load that you want to apply
- Write a program implementing that model
- The program also issues commands/requests synthesized from the model
  - E.g., if the model says open a file, the program builds the appropriate `open()` command

Using the Model

- May require creation of test files or processes or network connections
  - Include how they should be created in the model
- Again, shoot for minimum performance impact of program running the model

Applying Test Loads

- Most experiments will need multiple repetitions
- Most accurate results are gotten if each repetition runs in identical conditions
- So your test software should work hard to duplicate conditions on each run

Example of Applying Test Loads

- For the Ficus experiments actually conducted
  - To examine performance impact of update propagation for multiple replicas
- Test load is a set of benchmarks
  - Involving file access, among other activities
- Must apply test load for varying numbers of replicas

Factors in Designing This Experiment

- Setting up volumes and replicas
- Network traffic
- Other load on the test machines
- Caching effects
- Automation of the experiment
  - Very painful to have to start each run by hand

Experiment Setup

- In this case, we need volumes to read and write
- And replicas of each volume on various machines
- Must be certain that our setup completes **before** we start running the experiment

Network Traffic Issues

- If your experiment is distributed (as ours is), how is it affected by other traffic on the network?
- Is the traffic you see on the network used in the test like the traffic you expect on the network you would actually use?
- If not, do you need to run on an isolated network?
- And/or generate appropriate network load?

Controlling Other Load

- Generally, you want to have as much control over other processes running on the test machines as possible
- Ideally, use dedicated machines
- But also be careful about background and periodic jobs
  - In Unix context, check carefully on cron and network-related daemons

Caching Effects

- Many types of jobs run much faster if things are in the cache
  - Other things also change
- Is the caching effect part of what you're measuring?
  - If not, do something to clean out caches between runs
  - Or arrange experiment so caching doesn't help
- But sometimes you should measure caching

Automating Experiment

- For all but very small experiments, it pays to automate
  - So you don't have to start each run by hand
- But automation must be done with care
  - Make sure previous run is really complete
  - Make sure you completely reset your state
  - Make sure that the data is really collected

Common Mistakes in Benchmarking

- Many people have made these
- You will make some of them, too
- But watch for them, so you don't make too many

Only Testing Average Behavior

- Test workload should usually include divergence from average workload
- Since few workloads always remain at their average
- And behavior at extreme points is often very different
- Particularly bad if only average behavior is used

Ignoring Skewness

- Generally not including skewness of any component
  - E.g., distribution of file accesses among a set of users
- Leads to unrealistic conclusions about how system behaves

Loading Levels Controlled Inappropriately

- Not all methods of controlling the load are equivalent
- Choose methods that capture the effect you are testing for
- Prefer methods allowing more flexibility in control over those allowing less

Caching Effects Ignored

- Caching occurs many places in modern systems
- Performance on a given request usually very different depending on cache hit or miss
- Must understand how the cache works
- And design experiment to use it realistically

Sampling Inaccuracies Ignored

- Remember your samples are random events
- Use statistical methods to analyze them
- Beware of sampling techniques whose periodicity interacts with what you’re looking for

Ignoring Monitoring Overhead

- Primarily important in the design phase
  - Must minimize overhead to the point where it is not relevant, if possible
- But also important to consider it in the analysis

Not Validating Measurements

- Just because your measurement says something is so isn’t necessarily true
- Extremely easy to make mistakes in experimentation
- So check whatever you can
- And treat surprising measurements especially carefully

Not Ensuring Constant Initial Conditions

- Repeated runs are only comparable if the initial conditions are the same
- Not always easy to undo everything the previous run did
  - E.g., same state of disk fragmentation as before
- But do your best
  - And understand where you don't have control in important cases

Not Measuring Transient Performance

- Many systems behave differently at steady state than at startup (or shutdown)
- That's not always everything we care about
- Understand whether you should care
- If you should, measure the transients, too
- Not all transients due to startup or shutdown

Performance Comparison Using Device Utilizations

- Sometimes this is the right thing to do
- But only if device utilization is the metric of interest
- Remember, faster processors will have a lower utilization on the same load
  - And that isn't bad

Lots of Data, Little Analysis

- **The data isn't the product!**
- **The analysis is!**
- So design experiment to leave time for sufficient analysis
- If things go wrong, alter experiments to still leave analysis time