modern

MARK PURDY, PURSOFT

the modern Unix server floor can be a diverse universe of hardware from several vendors and software from several sources. Often, the personnel needed to resolve server floor performance issues are not available or, for security reasons, not allowed to be present at the very moment of occurrence. Even when, as luck might have it, the right personnel are actually present to witness a performance “event,” the tools to measure and analyze the performance of the hardware and software have traditionally been sparse and vendor-specific. Because few real Unix cross-platform toolkits exist, there is no standard method for administrators to accurately view an event, record it for later analysis, or share it with additional qualified personnel for an efficient resolution.

TRACKING DOWN LATENCY

The first step in studying the performance of a Unix server environment is to track down any application’s transaction latency from the user’s request event to the ultimate screen paint. The next step is to diagram the server topology. To do this, many elements come into question: the

network topology, database update (and locks), disk arrays, process scheduling, CPU performance, memory affinity, and driver/interrupt service times. You can then use a set of performance analysis tools to study the hardware and software environment along the complete path of the measured transaction latency. This step-by-step breakdown of each part of a user transaction should reveal the elements where latencies are experienced. A performance analysis toolkit should measure these elements and the performance degradation associated with them.

The user thread, as it relates to a transaction and the path required to resolve the user’s query, should be studied in detail. There are two steps to follow when tuning the latency of a system: First, define each step a user transaction must perform to satisfy the user request; then time each step. Though seemingly simple, this can be quite difficult in a multinode transaction environment. One problem is finding an accurate time base. NTP (Network Time Protocol) is generally the best for timing internodal network latencies. Once on board a machine, the CPU’s time base (nanosleep) should be adequate. Every step must be defined as accurately as possible. Every step includes every packet migration, every packet to process switch, every disk device interaction, and every

performance monitoring

**Today’s diverse and decentralized computer world demands
new thinking about performance monitoring and analysis.**

modern performance monitoring

CHOOSING THE
RIGHT TOOLKIT—
THEN AND NOW

The common Unix performance tools that have traditionally been used are

network interaction. Timestamps must be recorded at every step and saved for analysis. These timestamps from the latency study will identify areas of the server environment to examine with a performance toolkit.

WHAT SHOULD YOU MONITOR? EVERYTHING THE OS WILL ALLOW!

You must decide where and what you want to monitor. This can be gathered from a study of the system's transaction latencies, as just defined. You can also monitor and analyze each of the components that service a user transaction and simply make each one as fast as possible. In the end you have a list of devices and processes, and you must then find the appropriate means to meter the items of interest.

Briefly, a Unix computer has five categories of measurable data objects of interest to the performance analyzer: global, CPU, network interfaces, disks, and processes. The first four describe the physical attributes of the user's Unix computer. The global category describes memory, paging, and swap characteristics; global file system memory usage; and other items such as the time, uptime, and load averages. The CPU category contains physical items such as usage, interrupt, trap, cross calls, and other Unix items such as device reads/writes and process migration. The network category encompasses the physical interface layer and its components, plus the logical TCP/IP layer, including socket usage items. The disk category includes the physical disk device metered data items, the interconnect from the CPU, and channel data items. Underlying all of this is the topology of the disk arrays, generally hidden from the CPU's point of view. This adds another layer of abstraction that must be accounted for when measuring disk I/O performance from a CPU-based performance metering tool. These four categories describe the Unix view of the hardware world.

The last category is the process layer, where most of the need to monitor is first felt. For example, users might complain about slow response time. Whether a CPU is running hot or a database CPU "over there" is slower than normal, the first point of incident is the process that notices the bottleneck. Then it becomes a question of where, what, and how to meter the objects registering the latency incident.

well known and rather limited. Most software engineers are familiar with *sar*, *vmstat*, *iostat*, and *netstat*, in addition to other vendor-specific tools of a similar format. A useful tool originating from the Sun world is *SymbEL*, created by Adrian Cockcroft and Rich Pettit. In general, however, smooth response time from all assets in a parallel, well-threaded application system requires a more comprehensive performance tool than is available in the traditional Unix tool set.

The first great concern for performance monitoring is sample time. Most third-party tools are geared toward server farms and gathering *coarse-grain* performance data for capacity planning and server load hot spotting. A sample time useful to the person measuring the performance of an application in today's distributed environment is not the same as that needed for a capacity planner. Successful performance monitoring and analysis requires *fine-grain* timing software and hardware engineering tools. These are much harder to find. This is one reason why in the past I have relied so heavily on the *SymbEL* tool to "hot-rod" a realtime database environment.

The question then becomes: What is coarse and what is fine? We define coarse as samples taken and stored or displayed at five to 30 seconds or greater. Fine refers to samples of one-tenth of a second to five seconds. For the software engineer worried about transaction latencies, one second or less is needed—especially in large, parallel SMP (symmetric multiprocessing) systems. These systems could have tens of thousands of transactions per second and may require latencies of eight milliseconds or less from input packet arrival to user data delivery. As processors continue into the gigahertz domain, useful subsecond samples can be done on the server. The only downside to a monitoring system that can support such a fine-grain sample time is when you decide to gather all this performance data into a log format and store it to a disk. Logs of this resolution can be large. (Later in the article I will discuss the sheer power that intelligent logging provides and why it's an acceptable trade-off.) My recent efforts in logging fine-grain Unix performance data to disk at one-second intervals have shown the following: All performance data for a 72-processor machine with 16 network interfaces and 128 disk drives running 4,096 processes requires log writes at four megabytes per second.

A SAMPLE PROBLEM SET IN A SERVER ENVIRONMENT

As an example, let's examine a generic problem set faced by many Unix engineers. It revolves around the hardware involved in defining a user file system on a disk array that may be in a SAN (storage area network) environment. There are several pieces to this puzzle on the server floor. In its simplest form, a file system is defined across several physical disk assets in a disk array or set of disk arrays. The file system's disks may be seen from the host in three ways: the interconnect from the host to the disk array(s); the disk array itself; and the disk slice (or disk-array-defined stripe) as presented to the host system as a physical unit. From the host, we can see the physical element we may use as an element in a stripe set. Implicit in that element is generally the path to that physical disk unit, but some systems use multipath drivers to complicate matters. Most Unix tools can see a whole file system or the individual disk, but few tools can display the performance of the file system *and* the elements used to build that file system. This is vital because performance degradations can occur at any point in the signal chain. There may be channel failure, degradations, or switch issues in the SAN environment. Any element will be affected differently depending on the type of problem.

The most useful meter for a disk is average service time. Intermittent faults on channel paths can be caused by dislodged SCSI connectors, twisted fiber cables, cache turned off by accident (usually by Unix administrators or field service personnel), along with numerous SAN and fiber switch problems. They usually manifest themselves in long disk latencies as measured by service time. The real issue is to find the physical element in a file system that is performing badly and isolate the condition causing the latency. Off the shelf, the RICHpse SymbEL tool zoom.se can allow the engineer to spot the physical disk unit. The RulesEngine displays the icon on the physical disk with different colors indicating the severity of the service time. Obviously, red is bad (i.e., a high service time); green is OK; and white means there is no activity on the disk.

A BETTER SOLUTION TO THIS PROBLEM

Users can write additional RICHpse scripts to help improve the ability to define a stripe set in a performance tool. The basics have been available in the RICHpse install set for some time. Basically, it allows for the time series measurement of the byte read/write, and for the service time measurement for a metadevice and its physical disk elements. This is all viewable by host-based stripe management software such as Veritas or DiskSuite. Writ-

ing a further script that enables users to select *all* of the individual elements of a stripe and view the reads/writes and service times and present the data in a time series format allows users to spot performance anomalies with ease. This can then quickly point users to the cache array, channels, SAN environment, switches, shared disk-array resources, etc. Further, the same stripe set and metadevice definitions of this script can allow users to perform similar operations for CPUs (processor set-intensive applications) and network interfaces (multiIP and IP trunks). Users can then capture an event such as this in the field, log it, and read the log later in a GUI that causes the logged server to appear as if it were in realtime.

In the example shown in figure 1, a latency study determines that within a user transaction a particular database interaction is not timely or a particular logical file system and its physical elements are not performing as required. A more comprehensive performance toolkit should find the problem hardware or array configuration error. The software view of the database transaction latency shows reasonable performance often but not consistently. In this example, when the multihost shared disk array was built, the physical disks were cut into 10 slices—five disks with two slices each. Seven slices are allocated to build the file system of server A. Server B's file system uses three slices that span two physical disks. Server B's sporadic use of the one shared disk causes performance degradation experienced on server A's file

FIG 1

modern performance monitoring

system. This is the symptom experienced in the latency study. The performance symptoms are shown by the toolkit as disk I/O rate and service time irregularities and, upon proper data interpretation, lead to the problem shared disk.

Analysis of the incident “searching for the disk element of the crossed stripe sets” will flag this malady anytime, anywhere. A toolkit with this ability allows users to model a stripe set or disk array in any physical viewpoint. The user may need to check channel topology for load balancing or two stripe sets for physical device interactions in the array or SAN topology. Generalized server floor disk array and SAN tools are still evolving as people define the topology. This requires equally evolved performance analysis tools.

A PROCESS PROBLEM IN A SERVER ENVIRONMENT

Another problem I’ve encountered dealt with a suspect database update thread (figure 2). The thread in question was running with several hundred other heavy and lightweight threads in a realtime database. Like many applications, there was a peak hour or so of activity every business day. The thread examined the database state and incoming data to formulate database updates for other database environments on the server floor. This thread was one of several hundred on a database server calculating a particular class of data. There were three sources of generated updates within this thread. The first source of updates was the current database state. The second source was the realtime updates. These were received from external sources via a set of front-end communications computers. The third source of updates consisted of examinations from several user-transaction processing machines and their local databases. All of the data needed was examined. Updates were performed to the local database environment, and these updates were queued for broadcast to several other database server machines.

The first user concern was registered from one of the consumers of the updates from this thread during sustained peak activity. The software engineer responsible for the process was called in to analyze this concern. When this thread became slow, the user database threads would show degradation in performance. Although there were several database servers, only the one with this process

affected server (process on) and a nonaffected server (process off).

It was immediately obvious there was an issue with the service times of the file system’s disks for the server with the process in update mode. There was also a marked difference in the graphical representation of the I/O rates to the arrays. The good server had a smooth graph, whereas the bad server was choppy, indicating *rubbernecking*. Disk array rubbernecking occurs when the back-end of the disk array cannot destage cache to the physical media as fast as the server can update the cache of the disk array. Generally, most disk arrays have high-/low-water mark parameters governing the array’s cache. When the high-water mark is reached, the disk array’s CPUs stop incoming I/O requests from the server and destage I/O blocks from cache to physical media until the low-water mark is reached. There is then a burst of I/O rates until the high-water mark is reached. This performance looks like rubbernecking on a time-series chart of the I/O rates and transfer service times. Upon closer examination, the disk

turned on had performance issues. While the software engineer checked out the code, we started to profile the performance of the disk arrays for the

FIG 2

array's statistics showed many more cache destages on the poorly performing server.

We then started to study the I/O rates of the threads touching that file system and discovered that the update process initially indicated would write a large chunk of memory to disk *occasionally*. This was the problem. The software performed as designed, and every so many updates would checkpoint their process to disk. When the system got busy, the checkpoint activity increased until there were just too many dirty pages in the disk array cache. Once the disk array cache got flooded, many other processes slowed down.

The fix was to redesign the checkpoint logic for the update thread. We formulated several fixes to reduce the impact of the checkpoint operation. One fix was to switch from a number-of-events threshold that would trigger the checkpoint to a time-interval checkpoint trigger. The second fix was to eliminate some of the checkpoint data. As the overall database environment evolved, the global database checkpoint and restart with resync became more robust, thus removing some of the need for the local per-process checkpoint logic in a 15-year-old application. High-resolution profiling of both server disk arrays was the critical information that led to this fix.

MODERN SUPPORT NETWORK USING PERFORMANCE MONITORING TOOLKIT

The previous two examples have a common overall topology that contributes to their inherent complexity. The current state of the typical large server floor is such that assets are commonly found in various locations, sometimes around the world, for tangible reasons. The support diagram shown in figure 3, though generic, describes the typical commercial server floor. In my recent professional experience, I had two server floors with several servers each. The support and programming people I managed were located many miles from the servers. Their workstations are represented at the top of the figure. I also had a local server for “source code NFS plus compile&link” and disk space for core files and performance logs of current interest to my group. This is all represented by the Support Server box in the figure. The LogHost servers on each floor store all data from any server having some form of performance incident. This includes core files and performance logs, both of which are quite large and should not be kept on the local server. Unix administrators generally manage the LogHost and control access to its data. This is necessary to manage incident tracking and ensure that the proper parties are provided the data required to resolve a performance incident. In a secure environment

the LogHost server can allow access to core files and logs by support people without necessarily having to touch the server LAN.

The support diagram in figure 3 is a fairly typical corporate server model. It shows the basic elements of a support matrix and can be scaled up for more support nodes, more server floors, or more servers per floor. The basic structure, however, remains the same. Many Unix administrators and software/hardware engineers tailor their “wares” to provide the performance metrics needed to satisfy the users of the hardware and software in this environment.

FIG 3

modern performance monitoring

ONE TOOLKIT FOR ALL SERVER ENVIRONMENTS

A full-featured toolkit must satisfy the many varied environments that engineers are operating in today. As the diagram in figure 3 shows, most commercial installations are generally separate subnets with the engineers sometimes separated from the physical machines. There may be several sets of physical machines to support. I had 17 E10K Sun machines plus development servers on two separate server floors. My direct application represented only a fraction of the physical assets of the company's server environments. The computer server-floor buildings were separated by 40 miles for multipower grid access and redundancy. The support environment (programmers and Unix support) was in another building many miles from the computer floors. There were also support incidents with various vendors providing hardware and software to the environment. Back then and even today, very few tools could provide the view of the performance metrics required to keep all these machines functioning properly. I quickly built a set of RICHPS tools to work with the large Sun E10K server times 17. These were still single-server-oriented tools. I had no general set of tools to manage and troubleshoot several clouds of servers.

I went on to write and have recently completed a tool set, the PurSoft Analyst, with the goal of resolving all of the performance monitoring issues I have faced over the years. It's my hope that varied Unix environments may now gather the benefit of my years of frustration. When writing this performance analysis toolkit, I felt it was particularly important to allow the data that is presented to the GUI to look exactly the same, whether from a realtime thread or from a disk log file. The GUI is able to display the data in text tables, where the layout features a user-configurable data display, or a graphic presentation as a time series display. This allows users a more detailed presentation and examination of particular user-selected data items. This is similar in structure to several capacity planning and management software packages on the market, but far more responsive.

One major component within this toolkit is a CLI (command-line interface)-style logging binary that can be remotely run on any server. All of a machine's metrics are logged for deferred analysis. Problems and issues have a bad habit of manifesting themselves at a time when you

least expect, randomly or only when certain conditions are met. It is not practical to be at the computer 24/7 to work with many problems in real-

time. The logging facility can capture a system crash, be scheduled for a particular time, or snapshot the machine activity when certain user-settable conditions are met. The logs can be accumulated and cataloged somewhere for support center analysis or for select vendor system engineer analysis. This fine-grain timing performance logging tool presents the same look and feel to the primary commercial Unix vendors as the Unix engineer on the floor. Thus, all participants—engineers, consultants, and administrators alike—involved in this server incident can share the same view of the incident's metrics. Each participant can read the same log file of the server's incident using a GUI that presents the log as if it were in realtime. Because the GUI log reader tool operates with tape-recorder-like transport control, it has the ability to "play" the performance of a logged server from the log file. Troubleshooters can pause, rewind, fast forward, and loop areas within the log to focus quickly on problem areas.

Another feature of this performance toolkit is a profiling analysis tool to help the engineer find any item in the sampled Unix data that is out of variance with any user-defined baseline. This profiler can define a RulesEngine that can listen to the computer and report when some parameter is out of bounds. An example of this concept is the RulesEngine in the SymbEL toolkit. The RICHPS library, which is written in SymbEL, has a facility, *liverules.se*, that implements a RulesEngine for all RICHPS SymbEL application scripts, such as *zoom.se*. The SymbEL language is very similar to C with several extensions and limitations, but any C programmer can easily write or modify the RICHPS tools. One of the best features of this tool set is the Rules facility. The user can modify the *liverules.se* library to implement a profile for the particular hardware environment.

The source code for these tools is too large to properly illustrate here but is freely available on the Web. RICHPS is an interpreter; thus, on older machines with many CPUs, it consumes considerable overhead. The ideal solution is a binary coded with efficiency in mind. This binary should be "barely seen and not heard" except for a RulesEngine exception. The disk logging binary I've developed can be invoked according to the profiling rules set. Logging is then turned on when an exception condition is detected by the RulesEngine. A support engineer

can then, as before, analyze logged performance data at a later time and place.

THE FUTURE IS NOW

Looking toward the future, artificially intelligent-style profiling will get us closer to realtime problem determination in that the profiler should be able to detect a hardware or software variance with a user-settable threshold, scan the process data, and determine the processes that caused the incident. For example, an I/O hot spot is detected and scanned to determine that certain user application threads were active doing I/O at that time. Action can then be taken directly toward those threads executing the I/O calls. This is a process we have all done manually, but it should be one that can be automated. With the system I propose, the Unix administrator of a server farm would have a toolkit that allows locating the incident using AI incident trapping software and capturing all needed data about the incident. When we augment the logging of a server with a highly adjustable profiler such as this, we will have the ability to gather a log at just the right time. This will make logs especially powerful because effective analysis can begin immediately upon inspection. Unix administrators need not search for the suspicious event; it's caught right at the beginning

of the log. They can then forward the instantly viewable problem to the parties best suited to fix the incident. This may be anyone, from the CPU and subsystem vendor engineers to the application software designers. With this kind of multiplatform interoperability, system administrators and troubleshooters everywhere will have, for the very first time, a standardized set of tools to resolve performance issues quickly and effectively. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MARK D. PURDY is the original computer ticker system architect of Bloomberg LP. Along with building and managing the ticker-system environment the Bloomberg organization became known for during his 18-year tenure, his legacy is complemented by the implementation of the Sun Microsystems E15K-based realtime, time-series database engine capable of 60,000 updates per second and 5,000 user transactions per second with realtime monitoring streams. Now a semiretired partner of Bloomberg LP, his efforts at PurSoft Inc. have led to the creation of complete server floor system monitoring and analysis tools designed to deliver peak global performance in diverse, large-scale computer environments.
© 2006 ACM 1542-7730/06/0200 \$5.00

acm
queue
architecting tomorrow's computing

Coming in March

Workflow Systems

From ad-hoc to automation

Business transformation with BPM

Process change—dealing with humans