Proceedings of the 5th WSEAS Int. Conf. on APPLIED INFORMATICS and COMMUNICATIONS, Malta, September 15-17, 2005 (pp148-155)

Using Existing Instrumentation For System Performance Tuning

KHALIL SHIHAB
Department of Computer Science, Box 36, Sultan Qaboos University, Al-Khod, 123 Oman

Abstract: - This work presents a new measurement methodology especially designed to improve the performance of interactive systems as perceived by the user (user-perceived performance). It uses heuristics and system performance tools to the diagnosis of bottlenecks and provides the necessary remedies to achieve acceptable computer performance. The technique relies on a high level functional model of the interaction between application workloads, the UNIX operating system, and system hardware. Current performance measurement and tuning techniques suffer from a multitude of problems when applied to interactive systems. Our reliance on these techniques for interactive system performance tuning has caused the systems to be tuned in a suboptimal manner with systems often failing to provide predictable performance

Keywords: Heuristics, Computer System Tuning; Bottleneck Detection; System Management.

1. Introduction

Satisfactory computer services depend greatly on the choice of configurations and capacity in the computer systems. Performance evaluation of computer and communication systems helps not only in determining how well they are performing and whether any improvements need to be made, but also in understanding their behavior in order to plan and to design the systems of the future. As the hardware cost of these systems is decreasing, their complexity and the demands being placed upon them are increasing dramatically. Therefore, considerable theoretical research and applied development have been focused on improving computer system performance.

Literatures in system performance and engineering reported many factors that affect system performance [1, 2]. Usage patterns, I/O configuration, CPU configuration, cache size, and system and user software are examples of these factors. Changing any of these variables can lead to different system behavior. However, we should regularly monitor our system and analyze the values of these variables before any changes we might consider. Based on the outcomes of the analysis, necessary actions can be taken in order to reach a well-configured system that has an acceptable computer performance.

Computer system managers should consider two views: user's view of performance and the computer's view. If users' jobs take a long time to run and complete, the manager should expect a number of complaints from them. On the other hand, if the system hardware resources are not well utilized, then the system is in trouble. This is also the case when the load on the resources is unbalanced or the throughput is low. Therefore, we need to ensure that every user gets a fair share of available resources and in the same time, we should keep maintaining a healthy system.

Therefore, an effective computer program is designed and built to help computer managers in the tuning process of their computers. For detection of bottlenecks, some heuristics and operational laws are also used [3] as a framework for modeling the relationships among the variables of computer performance. In particular, the program encodes the functional model of a computer operating system. The inference method combines expert assessments with the measures that produced the system monitoring tools. These tools are also called system management tools, tuning tools, or system measurement tools (c.f. section 7).

While the system is running, the program predicts the values of observable system counters available from the UNIX performance-monitoring tools. During diagnostic inference, observed performance monitor values are analyzed to find the most probable assignment to the workload parameters.

The tuning problem is considered in this work as two interrelated activities: self-tuning and learning. The following sections provide some background on automated bottleneck detection, describe the structure of the system model, and discuss empirical procedures for implementing these activities.

2. Dynamic System Performance

When a computer system is running, many factors should be considered for evaluation. These contribute to a job's total time. Therefore, we should look at CPU time, I/O time, and network time to find out whether the system is spending more time in the System State (i.e. executing operating system calls) than in the User State – executing users' programs. For instance, to find out whether the system is overloaded, we may need only to investigate the I/O time.

Other important factors should be considered in order to achieve acceptable computer behavior. These are system-related factors and they are as important as user related factors. In any system, there are three fundamental resources CPU, memory, and I/O subsystems (e.g. disks and networks). Each resource has its own particular problems. The job of a manager is, therefore, to determine which subsystem is causing his/her system to slow down (i.e. a bottleneck). For example, CPU contention and CPU utilization provide good understanding of the status of the CPU and its limitations. Memory contention arises when the memory requirements of the active processes exceed the physical memory that is available on the system. Another good indication of degradation of system performance is when we notice that the system is paging [4].

The existing operating systems and the UNIX systems in particular contain a number of measurement tools available [2]. These tools are good resources that provide sufficient data about general system and per component behavior. The UNIX systems, for example, have a good number of monitoring tools such as uptime, ps, iostat, sar, vmstat, and netstat (c.f. section 7). We can also use the UNIX utility cron that runs specified UNIX commands at regular intervals and collect the relevant data to system performance. Necessary changes to the computer configuration should be taken based on the analysis to of the collected data.

3. Understanding System's Workloads

The principal aim of performance tuning is to analyze the behavior of the configuration of a computer system to the existing workload [5, 6]. Understanding our system workload is therefore necessary to be able to determine the necessary hardware that supports it. The workload definition must include not only the type and rate of each component but also the identification of both the typical and peak request rates.

After a complete definition of the system's workload, we will be left with many courses of actions that can be taken to enhance the performance of our computer system. These actions include eliminating unnecessary daemons and other system processes, giving the highest priority to the most important jobs, and shifting some jobs to run at another time.

Analyzing the workload enables us to determine some of its major characteristics, for example, whether it is I/O-bound, CPU-bound, or both, and so on.

4. Self-Tuning Systems

Other objectives of this work include the dynamic tuning of an operating system. LINUX is used for this purpose because its source codes are accessible.

In order to achieve a satisfactory level of performance for a live system, the used method should be fast and its overhead should be negligible. These restrictions cannot be achieved if a detailed analysis of a real workload is required. Therefore, an alternative method suggested here is based on system measurement tools, such as iostat, vmstat, and ps.

If the above-mentioned restrictions are taken into account, then the dynamic tuning can be achieved by an adjustment of the system's parameters. However, these parameters are dependent on the used operating system and the hardware capacity and configuration. In particular, the number of these tunable parameters differs from one operating system to another, and it also differs from one version of an operating system to another. Furthermore, in order to change the values of these parameters, each operating system has built-in commands that can be used for this purpose. These commands are also operating system dependent. Therefore, a general dynamic tuning technique cannot be achieved. However, the method can easily be adapted if it is required for a different platform.

The system management tools, such as iostat, vmstat, renice, ps, time, kill, and netstat, that are provided with almost all operating systems are not only being used for assessing the current state of system performance but they are also used successfully for tracking the changes in workloads and system performance. Systems' managers, for their daily management tasks, use these tools and their demands are negligible.

Therefore, the on-line tuning should be based on a quick analysis of the results that are produced by these system management tools.

5. Detection of System Bottlenecks

A bottleneck is a limitation of system performance due to inadequacy of a hardware or a software component. It is also the result of bad system organization. Once a particular component is identified as the bottleneck, a number of remedies exist. Theses include running big jobs at lower priority, terminating the jobs with largest memory requirement, distributing I/O workload more evenly, or eliminating unnecessary daemon processes. Other actions require some changes to the parameters of the operating system. These include reducing the size of buffer cache if the system reveals of having a memory problem or increasing the size of memory cache if the system has a disk I/O problem. These and other necessary actions will resolve the bottleneck by reducing the time spent using the component that is causing it.

Management tools play an important role in the process of bottleneck detection of a live computer system [6, 7, 8]. For example, response times can be inferred from both the throughput and the utilization measures that are produced by these tools. The throughput itself enables us to identify the bottleneck and its causes. Clearly, the system component that saturates at the lowest rate is the bottleneck. This component can be characterized by having the largest service demands. The key to determining this result is the consistency law.

Let Di and Ui denote the demand and the utilization of hardware center i. The Throughput Law states:

$$T = Ui/Di \qquad (1)$$

Where T is the system throughput. When any of the hardware components becomes saturated, that is when its utilization = 1, the whole system becomes saturated. Let max be the index of the bottleneck center. The maximum throughput for any resource i is

$$Tmax = 1/Di \qquad (2)$$

Therefore, the center with the smallest T in the system will determine the maximum throughput the system can achieve. This computer center is the bottleneck.

6. The UNIX Systems

AIX is the only operating system of the UNIX family that allows us to tune its parameters without need to rebuild the kernel and reboot the machine [4, 9, 10]. Other UNIX systems, such as Solaris, need to redesign its kernel so that they accept the automatic and dynamic tuning. Otherwise, the tuning should be carried out when the system is doing almost nothing, at night for example. In this case, the anticipated load during the next day has to be considered.

LINUX and MINIX have no system management tools, and you also need to rebuild the kernel after each change of the values of their tunable parameters. It is not difficult to add these tools to the kernel. However, it is hard to capture the reaction of these systems, after changing their parameters, to a real workload in order to fulfill the first activity, namely the self-tuning activity.

Dynamic tuning cannot be carried out on a live system unless the used method is fast and its overhead is negligible. These restrictions cannot be achieved if a detailed analysis of a real workload is required. Therefore, our alternative method, that is described here, is based on system measurement tools, such as iostat, vmstat, and ps.

The tuning problem is considered in this work as two interrelated activities: self-tuning and learning (c.f. section 10).

7. System-Management Tools

They are efficient commands that periodically collect and record performance data. Other features of these tools include the following:

- They can provide system-performance reports at a fixed interval indefinitely.
- They report on activity that varies with different types of workload.
- They report on activity since the last previous report, so changes in activity are easy to detect.

Examples of these system-management tools are:

iostat provides a picture of the state of the system every certain unit time.

vmstat provides a picture of overall memory use, and supplies data on I/O, and CPU. It can be used to find out whether the system is memory-limited or I/O, or both.

ps reports the actives processes. It is a good tool for identifying the programs that are running in the system and the resources they are using.

sar displays statistics on operating system activities such as directory access, read and write system calls, forks, paging activity.

uptime reports the average number of jobs in the run queue over a given period of time.

ab is apache bench which simulates multiple web browsers. A good networking and application server test.

Therefore, the system's parameters can be adjusted based on an overall assessment of the system behavior that is reported by the system-management tools. For example, if it is found that the disk service time is greater than 50ms, then the inode cache size should be increased by 20%. This quantity, i.e., 20%, is obtained by the off-line training method (section 10 elaborates on this point).

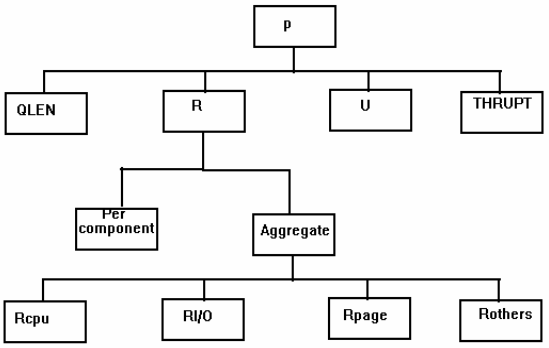
8. Heuristic Rules

Heuristics, a form of cognitive strategy, have been studied in discplines such as cognitive psychology, social psychology and social cognition. Heuristics are rules of thumb for reasoning, a simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike formal structures like algorithms, heuristics do not guarantee optimal, or even feasible, solutions and are often used with no theoretical guarantee.

The use of heuristics is often contrasted with probalistic, statistical, or rationalistic reasoning, according to which people use rationalistic and systematic ways to solve problems and generally seek the optimal results.

From the results of the measurement tools, an overall assessment of system performance can be initiated and that would lead to assign the best values for system tunable parameters [2, 10].

The heuristic rules assist in the traversal of MNG (management navigation graph).



F ig. 1: MNG (management navigation graph)

Figure 1 represent a management navigation graph, where P denotes system performance; R denotes response time; U denotes utilization; THRUPT denotes system throughput; QLEN denotes queue length; Rcpu denotes the CPU time; RI/O denotes the I/O time; Rpage denotes the time spent in the paging activities.

Examples of the implemented heuristics are as follows:

Rule 1: If any paging-space I/O is taken place, then the workload is approaching the system memory limits, i.e. there is a memory problem.

Rule 2: If the sum of user and system CPU utilization is greater than 80%, then the workload is approaching the CPU limits, i.e. there is a CPU problem.

Rule 3: If the I/O-wait percentage is non-zero, a significant amount of time is being spent waiting on I/O, and some part of the workload is I/O-bound, i.e. there is a disk problem.

Rule 4: If the number of blocked processes approaches or exceeds the queue length, then there is a disk problem (bottleneck).

Rule 5: If there is more system time than user time and the machine is not an NFS server, then there is a system problem.

Rule 6: If the idle time and the load average are both high, then we have a memory problem

Rule 7: If the average arrival rate is increasing, then select QLEN.

Rule 8: If the service time is greater than 50ms, then increase the inode cache by 20%.

Rule 9: If the queue length is more than four times the number of CPUs, then it is long, i.e., select QLEN.

Rule 10: If the utilization of CPU is greater than 80% or the utilization of a disk is greater than 35%, then there is a utilization problem, i.e. select U.

Rule 11: If vmstat.swap is greater than 4000k, then increase the swap area.

Rule 12: If sar,ufs.lpf is less than or equal to 100% and greater than zero, then double the inode area.

Rule 13: If we have a disk problem (busy or a slow disk), then we have a throughput problem.

Rules 14: If we have a throughput problem, use the formula (2) to identify the disk that causes this problem.

The conflict between memory performance, disk performance, and processor performance is resolved in favor of memory, and then in favor of disk. This is because the memory problem can cause a disk problem.

9. Implementation and Results

The on-line tuning and the off-line learning were carried out on the same system hardware specifications. The on-line tuning was carried out on the UNIX system running under Solaris operating system. The off-line experimental analysis and learning were conducted on the same system, when the system is idle.

The programs that listed at the end of this paper are selected pieces from our program. The first program is a script written in cshell. It uses some of the UNIX accounting tools for collecting the required data for performance analysis. The second program is written in C++ uses some heuristics and the results of the first program for allocating some possible bottlenecks.

10. Self-Tuning Systems

A self-scaling benchmark is developed (see the following subsection) in order to implement the self-tuning strategy. LINUX is used in this work as a platform for the implementation. This work involves the learning activity, which is the main step in the process of self-tuned operating system. The second activity is for finding the best values of system tunable parameters. The following subsections explain these two activities.

10.1. The Learning Activity

Given:

1. The values of the system measurements, CPU utilization, I/O utilization, response time, throughput, etc.

2. A self-scaling benchmark that produces similar values of the system measurements that are produced during the first activity (see the next section for more details).

Use:

Heuristic rules (thresholds) and management navigation graph (MNG) to learn the best values of the system tunable parameters. Here we should keep changing the values of the system parameters, i.e. moving these values up and down, within their permissible intervals until no more enhancements in the system performance can be achieved.

10.2. Self-Scaling Benchmark

In order to produce the best values of tunable system parameters, a benchmark can be used that automatically scales itself across the computer system under study.

This type of workload model is characterized by having a set of tunable parameters. The number of these parameters depends on the number of performance indexes (measurements) that are indicated by the system measurement tools. During the execution of this model, its parameters can automatically be adjusted to reach a performance state (base state). The base state is the performance assessment of the current system that is close enough to the performance assessment that produced the system measurement tools on the same system.

Adjusting of the benchmark parameters should be guided by a set of heuristic rules instead of using a random or a blind search.

There are a number of self-scaling benchmarks that can be used, after some modifications, for this purpose, such as TPC-B, TPPC, Sdet, and SDM. Otherwise, it is not difficult to design and to build a self-scaling benchmark.

Once the base state has been produced for a particular run, the system should invoke the second activity for finding the best values of system tunable parameters.

10.3. System Tunable Parameters

Almost every operating system has a number of tunable parameters, Solaris for example has around 30 of such parameters, and AIX has around 52 tunable parameters. To change the default value of each parameter, there are many commands that can be used in order to tune these parameters. AIX on PowerPC or RS/6000 has the tuning commands: *fdpr* optimizes executable files; *nfso* changes the values of NFS options; *nice* executes a command at a specific priority; *no* changes the values of network options; *renice* changes the priority of running processes; *schedtune* changes the values of VMM memory load control parameters, the CPU-time-slice duration, and the paging-space-low retry interval; *vmtune* changes the Virtual Memory Manager page replacement algorithm parameters.

Frank Waters in his book "AIX Performance Tuning" reported a number of AIX tunable parameters.

11. Conclusion

A model and a computer program are developed. The underlying technique is based on heuristics and operational laws for detecting computer system bottlenecks. The model and the program are currently being extended and verified in order to implement another set of heuristics and laws. Fortunately, in the realm of computer performance analysis, it is relatively easy to generate the needed data and therefore to automate that data collection effort. The implemented model is effective for dynamic tuning of system operating parameters, such as cache sizes, in response to inferred application loading.

Also, we plan to use similar approaches to predict the effects of changes to application workload parameters. The model can predict throughput and bottlenecks given an increment to application workloads.

References

[1] Accetta et al. Mach: a new kernel for UNIX development. *In Proceedings of USENIX Association Summer Conference,* pages 93--112, Atlanta; 1986.

[2] Loukides, M. 1992; System Performance Tuning, O'Reilly & Associations, Inc. 1992

[3] Buzen, J. Fundamental operational laws of computer system performance. *Acta Informatica,*Vol. 7, 1976, pp. 167—182.

[4] Waters, F. *AIX Performance Tuning,* Prentice Hall, 1996.

[5] Ferrari, D. Workload characterization for tightly-coupled and loosely-coupled systems. *In Proceedings Sigmetrics and Performance '89 International Conference on Measurement and Modeling of Computer Systems*, page 210, Berkeley, California. ACM, 1989.

[6] Domanski, D. A PROLOG- based expert system for tuning MVS/XA. *Performance Evaluation Review,* Vol. 16, 1989, pp. 30—47.

[7] Gian-Paolo D. Musumeci and Mike Loukides, *System Performance Tuning, 2nd Edition.* O'Reilly & Associates, 2002.

[8] Joseph D. Sloan, *Network Troubleshooting Tools.* O'Reilly & Associates, 2001.

[9] Stern H. Mike Eisler, and Ricardo Labiaga, *Managing NFS and NIS, 2nd Edition.* O'Reilly & Associates, 2001.

[10] Wilson, E. and James Naramore, *Network Monitoring and Analysis.* Prentice Hall, 2000.

Appendix

```
#!/bin/csh
# long term performance collection script
if ($#argv != 2) then
echo "usage: monitor interval filename"; exit
else
 echo "Performance Log File Collected By Monitor"
> $2
        echo >> $2
endif
iostat -tDc -l 32 $1 2 > iolog$$ & vmstat $1 2 >
vmlog$$
echo >> $2
echo "performance for" $1 "seconds ending at "
`date`>>$2
wait
head -2 vmlog$$ >> $2
tail -1 vmlog$$ >> $2
rm vmlog$$
head -2 iolog$$ >> $2
tail -1 iolog$$ >> $2
rm iolog$$
uptime >> $2
//**************************************
*****
//        To run program -
```

```
//                  g++ csp.C
//                  a.out
//This program finds the relevent figures from the vmstat,
//iostat and uptime UNIX commands and identifies the possible bottlenecks.
//
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <iomanip.h>

#define in_file "result.txt"
#define PO 53
#define DiskU1 104
#define DiskU2 108
#define DiskU3 112
#define DiskU4 116
#define DiskU5 120
#define DiskU6 124
#define CpuI 129
#define LoadAv 142

struct Values
{
 int PageOut; int CpuIdle;          float LoadAverage;
        int CpuUtil; float DiskUtil[5];
};
void Setvalues(Values &Sysresults, int &counter, ifstream monitorFile);
void OverThirty(float x); void cpu_idle(float a, int b);
void cpu_disks(int CpuUtil, float diskAv);
void Outputfn(Values Sysresults);
main()
{
char z; char quitx; int count; Values Sysresults;
while (quitx != 'Q' && quitx != 'q'){
  count = 0; system("monitor 1 result.txt");
    ifstream monitorFile(in_file); if (!monitorFile){
        cout << "File Result.txt cannot be opened"<< endl; quitx = 'q';}
     else {while (monitorFile.peek() != EOF){
              monitorFile.get(z);
              if (z == ' ') {
               count ++;
          while      (monitorFile.peek()      ==      '      ') monitorFile.get(z);
           Setvalues(Sysresults, count, monitorFile);}
          }
      monitorFile.close(); system("rm result.txt"); Outputfn(Sysresults);
     cout<<endl;
     cout<<"Press C     to continue or Q     to quit"<<endl; cin>>quitx;
    }
   }
}
//*****************************************
***
//          Outputs the results to the screen.
void Outputfn(Values Sysresults)
{
float diskAv = 0; int i;
for (i = 0; i < 6; i++) diskAv = diskAv + Sysresults.DiskUtil[i];
system("clear");
cout<<"************************************"
<<endl;
cout<<"*"<<endl;
cout <<"* Page Out: "<<setw (7)<< Sysresults.PageOut;
if (Sysresults.PageOut > 0) cout<< "  Paging has reached a high level";
cout<<endl;cout<<"*"<<endl;
cout<<"************************************
*"<<endl;
cout<<"*"<<endl;
cout <<"* Disk Utilisation: "<<endl;cout <<"*"<<endl;
for (i=0; i < 6;i++){
  cout <<"* Disk "<<i<<": "<<setw (7)<<Sysresults.DiskUtil[i];
   OverThirty(Sysresults.DiskUtil[i]);}
cout <<"*"<<endl;cout <<"* Average Disk Utilisation: "<<setw (7)<< diskAv<<endl;
cout <<"*"<<endl;
cout<<"*****************************"<<endl;
cout<<"*"<<endl;cout <<"* CpuUtil: "<< Sysresults.CpuUtil;
cpu_disks(Sysresults.CpuUtil, diskAv);cout <<"*"<<endl;
cout<<"*****************************"<<endl;
cout<<"*"<<endl;
cout <<"* CpuIdle: "<<Sysresults.CpuIdle<<endl;
cout <<"*"<<endl;
cout<<"*****************************"<<endl;
cout<<"*"<<endl;
cout<<"* Load Av: "<<Sysresults.LoadAverage;
cpu_idle(Sysresults.LoadAverage, Sysresults.CpuIdle);
cout <<"*"<<endl;
cout<<"*****************************"<<endl;
}
//*******************************
```

```
//        Places the relevant values in the structure
void Setvalues(Values &Sysresults, int &counter, ifstream monitorFile)
{
int p;
switch (counter)
{
case PO: monitorFile >> Sysresults.PageOut; counter ++;break;
case DiskU1:monitorFile >> Sysresults.DiskUtil[0]; counter ++; break;
case DiskU2:monitorFile >> Sysresults.DiskUtil[1]; counter ++; break;
case DiskU3:monitorFile >> Sysresults.DiskUtil[2]; counter ++; break;
case DiskU4:monitorFile >> Sysresults.DiskUtil[3]; counter ++; break;
case DiskU5:monitorFile >> Sysresults.DiskUtil[4]; counter ++; break;
case DiskU6:monitorFile >> Sysresults.DiskUtil[5]; counter ++; break;
case CpuI:monitorFile >> Sysresults.CpuIdle;
                Sysresults.CpuUtil = 100 - Sysresults.CpuIdle;
                  counter ++; break;
case LoadAv :monitorFile >> Sysresults.LoadAverage; counter ++; break;
  }}
//*****************************************
//
//        Determines if the disk figures are over 30%
//
void OverThirty(float x)
{
if (x > 30) cout<<"        The Disk utilization is high"<<endl;
else cout<<endl;
}
//*********************************
//        Determines the state of the paging and memory
void cpu_idle(float a, int b)
{
 if (a > 1 && b > 30)
     cout << "The system is paging and there is not enough memory"<<endl;
        else cout << endl;}
//*****************************************
//        Determines the cpu utilization and disk figures.
void cpu_disks(int CpuUtil, float diskAv){
if (CpuUtil <30 && diskAv >30)
  cout<<"    The system is I/O bound"<<endl;
 else if (CpuUtil > 30 && diskAv < 30)
    cout <<"The system is CPU bound"<<endl;
        else if (CpuUtil < 30 && diskAv < 30)
                cout <<"        The system is underutilized"<<endl;
        else if (CpuUtil > 30 && diskAv > 30)
          cout <<"        The system is over utilized"<<endl;
}
```