MEASUREMENT AND TUNING OF COMPUTER SYSTEMS USING STATISTICAL DATA AND HEURISTICS

[1]Khalil Shihab, [2]Haider Ramadhan
[1]Assist. Prof., Department of Computer Science, Box 36, Sultan Qaboos University, Oman
[2]Assoc. Prof., Department of Computer Science, Box 36, Sultan Qaboos University, Oman
Email: kshihab@squ.edu.om , haiderr@squ.edu.om

ABSTRACT

Satisfactory system performance requires a continuous adjustment of tunable system parameters. These parameters are typically used to minimize the execution time. However, by the coupling of a performance model with an application, system parameters can be determined without user intervention. In this work presented here, a novel performance prediction system has been used to provide suitable performance models which can determine application mapping parameters, code execution decisions, and system choices on-the-fly. The method uses heuristics and system performance tools to the diagnosis of bottlenecks and provides the necessary remedies to achieve acceptable computer performance. The work introduces a parameter prioritizing tool to focus on those performance critical parameters. It also demonstrates how to reduce the time when tuning a large system with many tunable parameters. The search space can be reduced by checking the relations among parameters to avoid unnecessary search.

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, 3, 4]. 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 [4, 5] 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 [6, 7, 8].

The existing operating systems and the UNIX systems in particular contain a number of measurement tools available [9, 10]. 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 [11, 12, 13]. 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. This requires characterization of the system loads, see the following (section 4) for more details.

3.1. Workload Characterization and Parameterization

This stage involves the collection, classification and implementation of a suitable set of representative examples for different types of workload.

The accounting software tool on the UNIX system at the University was used to collect the required data for workload modeling. Our first intention was to find the most important traffic measures and their roles in the selection of a subset of workload components, and in the classification of these components. Also we were investigating the effect of the groups to which the users belonged and of the source of the traffic in order to assign an appropriate weight for each group and for each source. A piece of software was designed and implemented to extract the data from the daily job accounting records, based on our characterization scheme.

Our classification scheme is based on the following classification principles:

- Classes should consist of components that reveal similar performance measures. (For example, I/O bound components should not be grouped in a single class that also contains CPU-bound components).
- Classes must distinguish workload components of special interest. (For example, if we are only interested in the response time of the interactive components, then we must have one class representing the interactive components and one or more classes representing the rest).
- Classes must distinguish different workload types (transaction, terminal and batch).
- Classes must not be limited to a small number. (For example we may have several classes representing batch services: One class for short jobs, one class for medium-sized jobs, and one for all others).
- Classes must distinguish different user groups and locations.

3.2. Methodology

The objective, therefore, is to produce a grouping of workload components based on conceptual clustering techniques. The results were compared with the results of the self-organizing map (SOM). Therefore, we developed a clusteruing technique that uses a weighted inter-cluster criterion to select the better clustering result. The method is outlined as follows:

Given:

1. A selected sample of data extracted from a daily accounting routine, see Table 1. (The standard measure or Z score was used [11] for normalization of data.)

2. The classification principles that are mentioned in the progress report (see Workload Characterisation and Parameterisation).

3. An extensive memory E of special cases and their meaning.

4. A distance measure (inter-cluster criterion) which is given as follows: Let Z_1 and Z_2 be any two components (vectors) such that,

$$Z_1 = \{z_{11}, z_{12}, z_{13}, \cdot\ \cdot\ \cdot, z_{1n}\}; \quad Z_2 = \{z_{21}, z_{22}, z_{23}, \cdot\ \cdot\ \cdot, z_{2n}\}$$

$$D_{ij} = \sqrt{\frac{\sum_{k=1}^{n} w_k \left(z_{ik} - z_{jk}\right)^2}{\sum_{k=1}^{n} w_k}}$$

D_{ij} = the weighted Euclidean distance between component i and component j

Q_{ij} = the weighted qualitative distance between component i and component j

T_{ij} = the total distance measure between component i and component j

W_k = the weights that are assigned to workload parameters. Initially, these values are based on the objectives of the study and assigned to each parameter to reflect its relative importance. They are subsequently changed during clustering according to the following heuristic:

$$W_k = W_k(1 - W_0); \qquad W_0 = \frac{1}{N+1}$$

Where N is the number of cycles.

Goal:

Distinguish clusters C1, C2, . . ., Cn, using the following method, and then select the better result according to the weighted inter-cluster criterion (ie. when inter-cluster similarity tends to be minimized and intra-cluster similarity tends to be maximized).

Method

The major steps of this method are as follows Stepp and Michalski [14].

1. Select k distinct components from the sample based on the given dissimilarity measure and the classification criteria - k may be found by using minimum spanning tree (MST), when a sudden change in the linkage distance has been reached.
2. Produce a category matrix. As an example, using categories: 0, 1, 2, 3 and 4, if the observed natural workload features vector is

V_p = (150, 100, 50, 20, 0), then its representation is
V_r = (4, 3, 1, 1, 0).

This is accomplished by dividing Vp by 1.5 and using the following scale:

The digit 0 represents the interval [0,20], the interval (20, 40] is represented by the digit 1 and, the interval (80, 100] is represented by the digit 4.

By the same way we treat Table 1 that is produced by the accounting tool on Solaris system. If the table contains nominal data, it should be categorized in the same way as we treated the numerical data.

3. Reduce the number of columns, if possible, in order to derive concepts. Columns that have similar data across the majority of components should be deleted. These columns (attributes) do not add any useful information for producing different clusters.
4. Set $W_k = 0$ for the corresponding deleted columns.
5. Use the derived concepts for filtering the workload components. The filtering technique is based on the majority of a simple matching method. For example, suppose that the following concepts have been derived.

$CS_1 = \{1, 2, 3\}$
$CS_2 = \{3, 1, 2\}$
The following workload component $WK_1 = \{5, 2, 3\}$ should be, therefore, placed into the list of components that are associated with the CS_1 concept.

6. If a component belongs to two or more concepts, then this component should be placed on an exception list.
7. Use the given formula above on the exception list to find a place for each component.

Table 1: A summary report produced by the accounting tool on Solaris.

COMMAND NAME	NUMBER CMDS	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	CHARACTER TRANSFER	BLOCKS READ
sendmail	463	1.47	170.52	57.99	17056656	1438
elm	296	0.68	296.78	66.69	10591064	1887
sh	465	0.47	279.03	79.85	56579	233
vi	268	3.10	641.18	11.70	10820442	1160
mail	226	0.45	10.62	71.58	4266504	835
finger	223	0.72	34.59	44.29	13201744	11
ls	364	0.44	0.48	70.45	536915	47
pc0	82	0.55	0.80	39.28	8004808	272
cc1	49	0.42	0.50	36.90	1478240	51
ld	62	0.38	0.47	41.14	49133	119
quota	96	0.23	0.42	62.33	2183776	3
cpp	135	0.21	0.31	64.65	4559334	188
pc	86	0.19	2.05	64.57	71036	262
in.comsa	99	0.18	210.42	65.60	1129072	7
tty	138	0.19	0.21	56.32	195848	30
in.ident	57	0.22	0.40	46.89	15378304	2
rm	158	0.14	2.16	71.29	1030	259
rquotad	77	0.15	154.11	64.26	793408	1
date	119	0.10	0.12	88.89	92127	2
uudemon.	113	0.11	0.33	81.63	67694	20
cat	108	0.08	0.09	98.51	130005	6
pt_chmod	85	0.09	0.14	78.35	91885	176
w	60	6.43	0.13	50.81	153823	3
as	38	6.37	0.17	50.03	520103	41
cp	87	6.31	0.18	74.11	816246	179
un	18	6.24	238.95	2.24	16664128	54
sed	80	0.08	0.17	76.17	125186	2
ufsdump	55	0.54	21.01	10.39	201138400	480
tset	64	0.11	1.18	49.53	1799680	3
in.finge	45	0.33	4.68	15.63	5820984	12
a.out	61	0.08	2.64	63.89	36521	1
in.rlogi	30	3.83	726.68	1.24	3132856	51
uuxqt	48	0.06	0.10	76.93	855936	3
df	50	0.08	0.10	54.10	66500	3
gcc	51	0.07	1.31	56.27	77316	149

talk	23	0.99	764.00	3.88	1893752	0
expr	62	0.06	0.14	63.18	15556	0
uusched	48	0.06	0.15	65.01	36096	260
fbe	22	0.14	0.17	24.88	844383	24
in.talkd	26	0.06	20.63	50.10	1082362	0
awk	27	0.04	0.07	75.14	15377	6
sadc	44	0.06	0.14	45.20	84392	11
more	27	0.04	5.88	61.81	121055	3
cg	22	0.16	0.18	15.02	4338408	25
ps	26	0.12	0.16	19.48	291880	9

After removing the outliers and smoothing the data if necessary (using a logarithmic function), the resulted category table is as follows:

Table 2: The resulted category table after the discretization of table 1

COMMAND NAME	NUMBER CMDS	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	CHARACTER TRANSFER	BLOCKS READ
sendmail	5	4	5	5	4	5
elm	5	3	5	5	4	5
sh	5	2	5	5	1	4
vi	4	5	5	3	4	5
mail	4	2	3	5	4	5
finger	4	3	4	4	4	1
ls	5	2	1	5	3	3
pc0	2	3	1	4	4	4
cc1	1	2	1	4	3	3
ld	2	2	1	4	1	3
quota	3	2	1	5	3	0
cpp	3	1	1	5	4	4
pc	2	1	2	5	2	4
in.comsa	3	1	5	5	3	1
tty	3	1	0	5	2	2
in.ident	2	2	1	5	4	0
rm	3	1	2	5	0	4
rquotad	2	1	5	5	3	0
date	3	1	0	5	2	0
uudemon.	3	1	1	5	2	2
cat	3	0	0	5	2	1
pt_chmod	2	0	0	5	2	4
w	2	5	0	5	2	0
as	1	5	0	5	3	2
cp	2	5	0	5	3	4
un	0	5	5	0	4	3
sed	2	0	0	5	2	0
ufsdump	1	3	3	2	5	4
tset	2	1	1	5	3	0
in.finge	1	2	2	3	4	1
a.out	2	0	2	5	1	0
in.rlogi	0	5	5	0	3	3
uuxqt	1	0	0	5	3	0
df	1	0	0	5	2	0
gcc	1	0	1	5	2	3
talk	0	1	5	1	3	0
expr	2	0	0	5	1	0

uusched	1	0	0	5	1	4
fbe	0	1	0	0	3	2
in.talkd	0	0	3	5	3	0
awk	0	0	0	5	1	1
sadc	1	0	0	4	2	1
more	0	0	3	5	2	0
cg	0	1	0	3	4	2
ps	0	1	0	3	2	1

The hierarchical representation produced by the program is as follows:

```
Resulting Tree =_|-> sendmail
                 |-| |-> elm
               |-| |-> vi
              |-| |-> mail
            |-| |-> finger
            | |_|--> ls
           |-|   |--> pc0
           | |     __|----> quota
         |-| | |--|  |----> in.ident
         | | |-|  |--> w
        |-| |   |--> cp
        | | |-> ufsdump
      |-| |_|----> un
      | |   |----> in.rlogi
      | |_|----> as
      |   |_____|----> tty
      |       |_____|------> fbe
      |            |------> cg
     -|     __|--> sh
      | |--|  |--> ld
      | |  |__|--> rm
      | |    |__|--> pc
      | |       |--> rquotad
      |-|  |----> in.co
        | |           _____|----> uudemon.
        | |          |    |_____|----> cpp
        | |   |------|         |----> tset
        |--|    |      |    |----> cc1
          |    |      |----|    |------> gcc
          |    |          |----|      |------> uusched
          |    |                |------|       _____|------> a.out
          |----|                       |------|    |------> expr
              |                                |_____|----> talk
              |                                     |_____|------> in.talkd
              |                                          |------> more
              |      |----> in.finge
              |      |     _____|----> pt_chmod
              |------|    |    |_____|----> cat
                   |    |        |_____|----> date
                   |----|            |_____|------> sed
                       |                 |_______|--------> uuxqt
                       |                       |--------> df
                       |_____|------> sadc
                           |_______|------> awk
                                |------> ps
```

We noticed that if a finer resolution is used, then there is no significant change in the resulting clusters but there is a little change in the performance of the program.

3.3. Workload Clustering Using the Self-Organizing Map (SOM)

Neural networks are being used in data classification, and satisfactory results have recently emerged. Therefore, we also used used Kohonen's Self-Organising Map (SOM) for this problem of workload characterisation .

SOM uses an unsupervised approach to learning. It defines a mapping from the input data space (input layer) Rn onto a regular two-dimensional array (output layer) of nodes. Each of these nodes is a composite profile or typical representative of all cases that fit that profile. The problem lies in which node should represent a case (input vector), the solution is to let the nodes compete for the right to represent a class of input vector. The winner will adjust its weights so that it becomes more like the input vector. Therefore, the winner is closest to the input vector. The neighbors of the winning node also adjust their weights to become closer to the input. This means that neighboring SOM nodes should be similar or should represent similar SOM input vectors.

Repeating the corresponding columns (attributes) has reflected the objectives of the workload characterization. If, for example, the interest is in the CPU performance rather than the other components, then the CPU characterization should be repeated to emphasize this importance. This modification is to ensure the increase of the importance of high-order attributes. Also, the columns reduction step that is used in method 1 is used here to make sure that the network does not organizes the components based on the most frequently occurring attributes that do not reflect the important features. Hiotis [15], and Caudill [16]I have noticed that the network self-organizes on lower-order attributes which are not the most critical for classification.

3.4. Workload Clustering

As presented in section 3 that the experimental data were obtained from daily reports that were collected on Sun Blade 100 at the Sultan Qaboos University, running under the Solaris 9 operating system. The reports, which relate to approximately 50 commands, form the input to the conceptual clustering and the SOM. They contain a command label and nine dimensions of data, such as the CPU time consumed, the memory space required, the number of disk I/O blocks transferred.

Our first intention was, therefore, to find the most important traffic measures and their roles in the selection of a subset of workload components, and in the classification of these components. In addition, we were investigating the effect of the groups to which the users belonged and of the source of the traffic in order to assign an appropriate weight for each group and for each source.

The workload dataset that was produced by the UNIX accounting system at our University, shown in Table 1, consists of 45 commands (senmail, elm, sh, vi, etc.), each represented by six features, namely number of commands (frequency), CPU time, real time, memory size, character transfer, and block read. In practice, these features are the most relevant and sufficient for characterizing the load on a UNIX system. It is worth mentioning here that workload characterization is the first and the most important step in computer performance evaluation. Normally, it involves the following:

- data collection (using monitoring software or an accounting routine),
- identification of the important components for performance study,
- partitioning these components into workload types, and
- performing cluster analysis on these types [24, 25].

The clusters of UNIX commands found by SOM are quite poor [21, 22]. It produces many small clusters. When we used the visualization programs that are included in the SOM package, such as Sammon mapping and Planes [23], to visualize the reference map, to identify distinct clusters was difficult for us because the boundaries between these clusters were vague. Clusters correspond to clear zones separated by dark hexagons, see Figure 1. Therefore, we used a data histogram technique that shows how many data vectors (UNIX commands) belong to a cluster defined by each unit and its neighbors by counting the number of hits. This number was obtained by using a trained SOM and a dataset. After calibration, some of the units in the map have labels showing an area in the map that corresponds to some of the UNIX commands. The resulted map and the dataset were then used as two input files to the visual program, which were included in the SOM package, to generate a list of coordinates corresponding to the best matching units (BMU) in the map for each data vector in the dataset (see Table 1 and Table 3.)

It has been reported that the number of neurons should usually be selected as large as possible, with the neighborhood size controlling the smoothness and generalization of the mapping. The mapping does not considerably suffer, even when the number of neurons exceeds the number of input vectors, given that the neighborhood size is selected appropriately. Nevertheless, as the size of the map increases—e.g. to tens of thousands of neurons—the training phase becomes computationally and impractically too heavy for most applications.

We conducted the experiments and clustered the workload data using both the SOM and our clustering algorithm. We also used a manual partitioning of data as a reference to facilitate an entropy calculation and comparison.

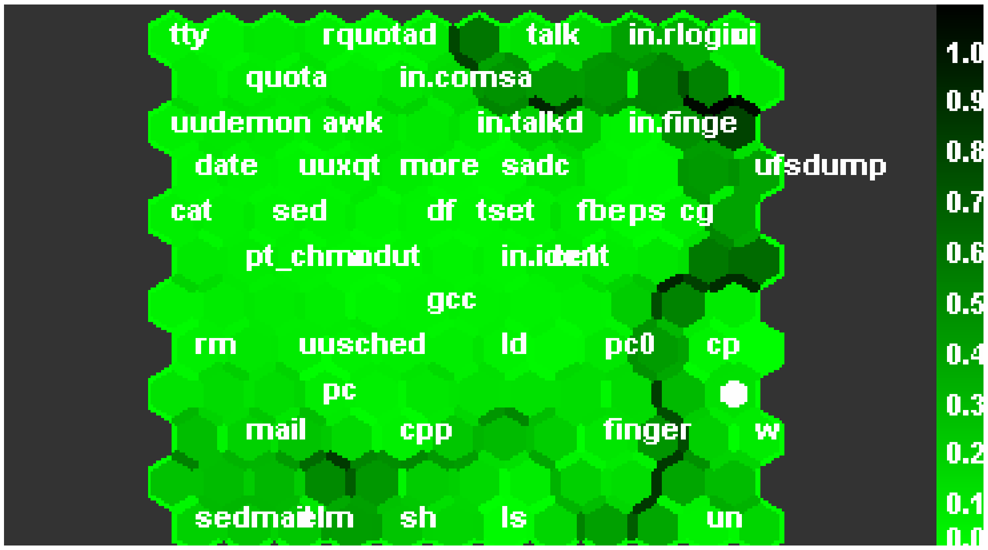

Figure 1**:** The u-matrix (unified distance matrix) visualization of the SOM for the UNIX workload dataset. The map is 12 × 12 neurons. The map has also been labeled. Clusters correspond to clear zones separated by dark hexagons.

The manual partitioning process has some advantages, including division along natural boundaries, but the process requires engineering, time and insight. With the help of the technical staff and the computer system manager in our department, we analyzed and characterized the reports that are generated by the accounting tool on the UNIX system in the department. Our first intention was to find the most important traffic measures and their roles in the selection of a subset of workload components, and in the classification of these components. Additionally, we were investigating the effect of the groups to which the users belonged and the source of the traffic to assign an appropriate weight for each group and for each source. The characterization scheme that we used for grouping workload components is based on the identified classification principles, see section 3.1. Table 4 shows the experimental results for the eight classes that were identified and produced by SOM, manual grouping, and our clustering technique. The percentages represent the clustering accuracy relative to the manual clustering.

Table 3: UNIX commands cluster produced by SOM

Cluster	Commands			
1	sedmail	elm		
2	sh	mail		
3	vi	in.rlogi	talk	
4	finger			
5	ls			
6	pc0			
7	cc1	ld	sadc	
8	quota	pc	in.comsa	
9	cpp	rm		
10	tty			
11	in.indent	tset		
12	rquotad			
13	date	uudemo	cat	pt_chmod
14	w	as	cp	Un
15	sed	uuxqt	awk	
16	ufsdump			
17	in.fing	fbe	cg	Ps
18	a.out	expr	uushed	
19	df	intalk		
20	gcc	more		

Table 4: The experimental results for the eight classes that were identified. The percentages represent the clustering accuracy relative to a manual classification.

	Clusters							
	1	2	3	4	5	6	7	8
Manual Grouping	18 entries	2 entries	9 entries	2 entrie s	1 entries	14 entries	1 entries	3 entries
Conceptual Clustering	88%	100%	77%	100%	100%	100%	100%	66.67%
SOM	89.89%	100%	44.44%	50%	100%	78.57%	100%	0%

4. SELF-TUNING SYSTEMS

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. However, the process of workload characterization and clustering should be first carried out. As we presented in sections ***, it enables us to understand the behavior of the system and allows us to effectively and efficiently use of the system measurement tools. Moreover, workload characterization and clustering enable us to identify and formulate performance tuning heurists.

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. BOTTLENECKS DETECTION

A bottleneck is a limitation of system performance due to inadequacy of 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 [17, 18, 19]. 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:

$$T = U_i/D_i \qquad (1)$$

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

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

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 [6, 20]. 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, 7].

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

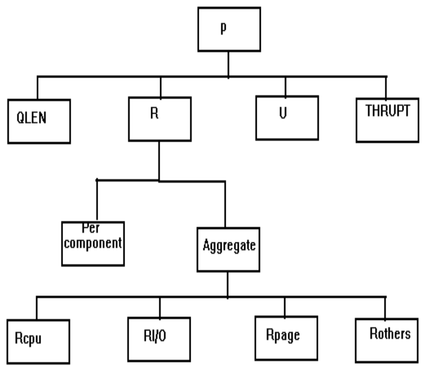

Figure 2: MNG (management navigation graph)

Figure 2 represents 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., selecting 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. CACHE PRINCIPLES

Caches work on two basic principles. The first is that if we spend a long time getting something that we think we may need again soon, you keep it nearby. The contents of our cache make up our working set. The second principle is that when we get something, we can save time by also getting the extra items we suspect we will need in the near future.

The first principle is called "temporal locality" and involves reusing the same things over time. The second principle is called "spatial locality" and depends on the simultaneous use of things that are located near each other. Caches only work well if there is good locality in what you are doing. Some sequences of behavior work very efficiently with a cache, and others make little or no use of the cache. In some cases, cache-busting behavior can be fixed by changing the system to provide support for special operations. In most cases, avoiding cache-busting behavior in the workload's access pattern will lead to a dramatic improvement in performance.

A cache works well if there are a lot more reads than writes, and if the reads or writes of the same or nearby data occur close together in time. An efficient cache has a low reference rate (it doesn't make unnecessary lookups), a very short cache hit time, a high hit ratio, the minimum possible cache miss time, and an efficient way of handling writes and purges.

10. 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.

11. 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.

11.1. 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.

11.2. 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.

11.3. 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 [6, 26].

Frank Waters [6] in his book "AIX Performance Tuning" reported a number of AIX tunable parameters.

12. CONCLUSION

A computer system tuning model and a computer program are developed. The underlying technique is based on heuristics and sysetm performance tools 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.

Designing and building a software tool for construction of reliable workload models of given real workloads is the first step in the process of tuning and evaluation of computer performance. The main advantage of the used method is its capability of producing accaeptable clusters of data. Using this approach, the workload models that are produced will have a general applicability to system performance evaluation, system tuning and capacity planning.

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.

13. REFERENCES

[1] Harrison PG & NM Patel (1993), "Performance Modelling of Communication Networks and Commuter Architectures", Addison-Wesley.

[2] Lazowska ED at al. (1984) "Quantitative System Performance: Computer System Analysis Using Queuing Network Models", Prentice-Hall, Inc.

[3] R. Bryant and D. O'Hallaron, Computer Systems: A Programmers' Perspective, First Edition, Prentice Hall, 2003;

[4] Daniel P. Bovet and Marco Cesati, Understanding the Linux Kernel, 1st Edition, O'Reilly and Associates Inc., 2001.

[5] Khalil Shihab and Haider A. Ramadhan. Automatic Detection of Performance Bottlenecks Using a Case-Based Reasoning Approach, *Journal of Intelligent Systems*, Vol. 11, No 6, pp 385-407, 2001.

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

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

[10]] Jain, R. The Art of Computer System Performance Analysis, John Wiley & Sons, Ltd., 1991.

[11] 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.

[12] Haverkort, B.R., Performance of Computer Communication Systems, John Wiley & Sons, Ltd., 1998.

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

[14] Stepp RE and RS Michalski (1986), "Conceptual Clustering: Inventing goal-directed classifications of structured objects", in RS Michalski at al. (eds.), Machine Learning: An artificial intelligence approach, vol. 2, Morgan Kaufman.

[15] Hiotis A (1993) "Inside a Self-Organising Map", AI EXPERT, pp 38-43, April 1993.

[16] Caudill M (1993) "A Little Knowledge is a Dangerous Thing", AI EXPERT, pp 16-20, June 1993.

[17] Cady J and B Howarth (1990) "Computer Systems Performance Measurement and Capacity Planning", Prentice-Hall, Inc.

[18] Terplan K (1992), "Communication Networks Management", Prentice-Hall, Inc .

[19] Kant, K. Introduction to Computer System Performance Evaluation, Mc Graw-Hill Inc., 1992.

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

[21] Khalil Shihab. Improving Clustering Performance by Using Feature Selection and Extraction Techniques, *Journal of Intelligent Systems*, Vol. 13, No. 3, pp 249-273, 2004.

[22] Shihab, K. I. and Ramadhan, H., A Clustering Technique Using Dynamic Filtering Concepts and its Application to Computer Workload Modeling, *Journal of Intelligent Systems*, Vol. 10, 4, pp. 321-344, 2000.

[23] Kohonen T, at al (1992) "Self-Organising Map", Program Package Ver. 1.2. Helsinki University of Technology, Finland.

[24] Daniel A. Menasce, et al. Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems,., Prentice-Hall, 1994

[25] Bolch, G., Greiner, S., de Meer, H. and Trivedi, K. Queueing Networks and Markov chains, 2nd Ed., John Wiley & Sons, Ltd., 2006.

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

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;
}
```