Automatic Bottleneck Detection

Russ Blake John S. Breese
(russbl@microsoft.com,breese@microsoft.com)

October, 1995

Technical Report
MSR-TR-95-10

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

We describe the diagnosis and treatment of bottlenecks in computer systems using decison theoretic techniques. The techniques rely on a high-level functional model of the interaction between application workloads, the operating system (Windows NT), and system hardware. Given a workload description, the model predicts the values of system counters observable with the Windows NT performance monitoring tool. Uncertainty in workloads, predictions, and counter values are characterized with Gaussian distributions. During diagnostic inference, observed performance monitor values are used to find the most probable assignment to the workload parameters.

In this paper we provide some background on performance modeling and automated bottleneck detection, describe the structure of the system model, and discuss empirical procedures for model verification. Our method for inferring the cause of the bottleneck is discussed. Initial results in diagnosing bottlenecks are presented. Based on the diagnosis of the bottleneck we can make both a recommendation of the most cost-effective hardware upgrade, and a prediction of the performance counter values and throughput once the upgrade is installed.

1 Introduction

Computer performance bottlenecks are typified by the overconsumption of some hardware resource or set of resources. Usually this results in the underconsumption of other hardware resources resulting in a delay completing the workload. Once a particular resource set is identified as the bottleneck, a number of remedies exist. These include distributing the load across additional instances of that resource set, installing faster resources in the set, or redesigning the workload to use another set of resources instead. These actions will resolve the bottleneck by reducing the time spent using the bottlenecking resource, possibly even shifting the bottleneck to another component.

Several problems arise which prevent the simplistic detection of bottlenecks by merely observing device utilization. In commodity computer hardware in widespread use today, there is the problem of inadequate system instrumentation. Frequently, resource utilizations are not measured directly. This is partly due to a lack of fast, inexpensive, accurate clocks for timing the usage of resources, and partly due to a lack of computer industry coordination concerning the metering of resource activity and access to that information throughout the hardware hierarchy. Even in a modern system such as Microsoft's Windows NT which supports over 500 different performance metrics [Blake, 1995], inadequate instrumentation remains an impediment to bottleneck detection. Gradual improvement is being made in these areas, but at present there is often a need to infer device utilization indirectly.

Another problem confounding simple bottleneck detection is that some resources are used to satisfy fundamentally different workload requirements. Just because in a certain case we deduce that the disk is the bottleneck, we cannot simply conclude that a faster disk is the correct solution. Modern computer systems use disks and local area networks for both virtual memory and file storage. A shortage of RAM can cause disk activity as easily as file activity can, so the correct solution might be to buy more RAM, not faster disks. Even if the activity is simple file activity, modern systems also use RAM to cache file data, so even if the activity is pure file access the addition of RAM may still be the right solution. Conversely if the activity is one-time sequential file access, it is unlikely that additional RAM will be of assistance, and a faster disk is required.

Which of these causes of the bottleneck prevails in a given case is key to the correct remedy. To answer this question we must infer the bottleneck's cause from existing system metrics. This is an inherently uncertain endeavor. We typically do not know precisely the users' patterns of access of the system's functionality. We do not know the specific performance characteristics of all the concurrently executing software on the particular machine in question. And typically we do not have perfect measurements of internal system states; rather there are some limited set of metered outputs from which we can determine performance.

In general, there has been relatively little work addressing the uncertain aspects of performance analysis in real-world, dynamic environments. To address this problem we use methodologies and techniques from probability and and statistics [Pearl, 1988, Heckerman and Wellman, 1995, DeGroot, 1970]. We construct an analogous probabilisitic model of operating system resource allocation policies. We explicitly represent uncertainties in the workload, the model, and the measurements and use estimates of these to deduce the workload most likely to have caused the bottleneck. Once we know the cause of the bottleneck we can use the model to predict the performance of proposed changes. Eventual applications of this diagnostic capability include support for hardware purchase decision making, advanced software development tools, and dynamic system tuning.

In Section 2, we review some of the literature relevant to performance analysis and bottleneck detection. In Section 3, we present issues in developing, calibrating, and verifying the model of the operating system. In Section 4, we present our method for inferring system behavior from observables, including learning an error model for model predictions, and in Section 5 we present empirical results.

2 Performance Analysis

The literature on computer performance modelling has been largely concerned with answering the following question: Given a planned workload, what selections of possible computer equipment, interconnection schemes, protocols, and algorithms should be made to produce satisfactory performance? This problem is generally attacked by determining the relevant characteristics of the workload to be applied and the relevant characteristics of the computer system performance behavior. A model of the proposed system is constructed, verified against either the actual equipment or a simulation, and used to predict the ability of the system to handle the proposed workload [Dowdy, 1989].

All computer performance models need a description of the workload which will be applied to the systems by the anticipated application. Application workloads are naturally expressed in terms of the use of those facilities the application makes available (e.g., open a document or print a report.) When possible, the original workload used to drive the model can be taken from traces of real system activity on a pre-existing system [Yu et al., 1985]. Considerable literature supports the clustering of application functions into a set of statistically equivalent synthetic activities [Ferrari, 1984, Raghavan and Kalyanakrishnan, 1985, Graf, 1987, Raghavan et al., 1987, Bodnarchuk and Bunt, 1991]. Workloads as described to the model are usually at a lower level of detail than application functions, such as at the level of calls for system services. Methods have been developed for hierarchically decomposing application functional actions into system level calls for input into models [Graf, 1987, Raghavan et al., 1987]. All of this literature assumes the workload is known; the problem is in specifying it economically at the right level of detail for consumption by the model.

Over the past decade there has been a shift away from monolithic applications on dedicated hardware to a client/server environment where inter-communicating applications are thrown together on a mix of servers. Ferrari has noted that "no systematic methodologies are known to reduce a multi-computer workload's description to a more compact and representative model of that workload" [Ferrari, 1989]. The independent construction of the applications and operating systems—combined with their inter-dependent operation—make it very difficult, if not impossible, to specify the workload. Furthermore, much of the workload applied to a system may not be the direct plan of the application designer at all. For example the amount of paging traffic induced by placing a particular application onto a particular system is seldom the intent of the application designer, since typically the application co-exists on the system with other applications, and the cumulative effect determines paging behavior. These very real concerns threaten to render the large body of effort

thus far expended on computer performance modeling irrelevant, because the workload to be supplied to any model cannot be known a priori.

The thrust of our research is to overcome this fundamental problem by using probabilistic methods to infer the most probable workload from the performance measures provided by an existing system. In the situation where a new application is to be added to an existing configuration, the inferred workload can then be combined algebraically with the anticipated application workload derived using the conventional techniques referenced above. If the inter-process interactions are too strong for the anticipated application workload to be characterized in isolation [Ferrari, 1989], a prototype of the new application embedded in the expected environment can be measured to infer the new composite workload. The resulting workload can be used to drive the many sorts of models discussed above, enabling classical analyses such as identifying the bottleneck and its causes, predicting the effect of equipment purchases, or predicting the effects of changes in operating system algorithms.

Previous attempts to automate bottleneck detection expertise have been constructed using rule-based techniques and have focused on performance tuning [Irgon et al., 1988, Domanski, 1989]. These methods have no explicit representation of a workload or a model of the system. While shown to be useful for automating tuning, a rule-based approach cannot be easily manipulated to extrapolate changes in the workload, differences in hardware configuration, or revisions in operating system algorithms. In addition these methods provide no explicit methodology for managing uncertainty in the heuristics or their mapping to the rules.

Hellerstein has recently compared different techniques for bottleneck detection [Hellerstein, 1994]. In his treatment Hellerstein refers to an observed performance metric as "causal" if that metric plays a principle role in the observed delay. This is unrelated to our notion of causality as used in reference to belief networks from decision theory, where the application workload "causes" the bottleneck on the system. In decision theoretic belief networks, the causal relationships fully dictate the structure of the model and are not changed by any particular measurement.

Some attention has been given to the issue of diagnosing transient bottlenecks in computer systems [Hellerstein, 1989, Berry and Hellerstein, 1991]. In our work thus far we have focused on bottlenecks which endure for some period of time. In principle it appears that the automatic bottleneck detection capability we have created could operate in real-time on a network of systems, displaying bottlenecks as they shift about the computer network.

3 A Model of Computer System Performance

Atomic models [Blake, 1979, Gray, 1987] of computer systems are a refinement of operational models [Buzen, 1976, Buzen and Shum, 1987]. Atomic models assign atomic performance values to fundamental system operations. Operational service times at devices can be thought of as molecules of device consumption; atomic models depict these molecules as being made up of atoms of device consumption combined to reflect the structure of the operating system. This permits operational models to reflect operating system policy decisions regarding the allocation of resources, without resorting to the level of detail and consequent long run times of simulation models. Atomic models permit realistic operational models to be constructed and verified for existing or proposed systems, are computationally simple, but like operational models provide only average performance metrics.

The system model we have developed is an atomic model which combines workload attributes with calibrated operating system characteristics and calibrated hardware resources to predict performance monitor counters. To accomplish this it must approximate the algorithm the operating system uses to allocate the workloads to resources. Despite the wealth of services offered by modern operating systems, such a model can be built at a fairly high level of abstraction with reasonable accuracy [Blake, 1979, Gray, 1987].

A high level of abstraction is possible because a surprisingly small number of basic operations comprise the majority of sustained activity inherent in most bottlenecks. This is in part due to the kernel [Ritchie and Thompson, 1974] and microkernel [Accetta et al., 1986] approaches to operating system construction. In this design paradigm, shared in part by the Windows NT operating system [Custer, 1993], system services are built on top of a relatively small number of primitive functions comprising the kernel of the operating system. Knowing the instruction path lengths of those kernel primitives at the root of sustained operations is sufficient to characterize the majority of system activity, since most non-primitive services are constructed using these atomic services as building blocks. The workload specification can therefore be limited to the important subset of calls that repeatedly invoke kernel operating system services.

The difficulty in constructing a model can be further reduced by initially restricting the application domain under consideration. In our case we have deferred constructing a detailed model of system graphical services. This limits detailed bottleneck detection in workstation environments where the graphical user interface is a primary component of sustained activity. It nonetheless leaves open the domain of the server in a network, a system which provides services to other computers over network connections.

Operating system atoms may be instruction path lengths, or they may be memory

consumption requirements of various system components. Operating system atoms are calibrated once each release on an arbitrary base system and do not need to be recalibrated until the next release, although small differences in the operating system code from one processor architecture to the next introduce additional uncertainty into the model.

The portion of the model which determines the amount of pressure on RAM page frames will illustrate its construction. A belief network of the model of paging behavior on Windows NT appears in Figure 1. The figure shows the structure of the model. The values of variables shown in double ovals are deterministic functions of the values of their predecessors. Deterministic nodes with no predecessors are constants based on the calibrated hardware or software, or are functions of variables in other portions of the model.

The application workload parameters are shown in Figure 1 as chance nodes (single ovals) at the upper left. The APP RAM DEMAND is the amount of RAM the application must access at steady state. The LOCAL PAGING AFFINITY is a number between 0 and 1 indicating the fraction of active virtual memory that is on the local server, as opposed to the fraction that is elsewhere on the network. At the bottom of Figure 1 is a chance node denoting a system counter or metric, Pages Input Per Second. As discussed in Section 4 the probability distribution for Pages Input Per Second is a function of the variable Input Page Rate as predicted by the model. This is the rate of input page traffic, a key system performance counter indicating how severely the system is thrashing or moving pages between RAM and disk or network. The other deterministic nodes in Figure 1 denote the internal operating system state variables that are well characterized when their predecessors are known, but are typically unmetered and hence unobservable.

Here are the atomic model formulae corresponding to the belief net fragment shown in Figure 1.

```
// NT characteristics, or measured directly.

SystemPaged: GraphicsRamDemand + SpoolerRamDemand + PoolPaged + System-
CodePaged + SystemNominalAvailableBytes ["megabytes"]


SystemNonPaged: PoolNonPaged + KernelNonPaged + ProtocolNonPaged + Driver-
sNonPaged ["megabytes"]


// Memory usage differs for CISC and RISC processors.

RelativeMemoryUsage: TableLookup ( RelativeMemorySize, InstalledProcessorIndex )
["ratio"]
```

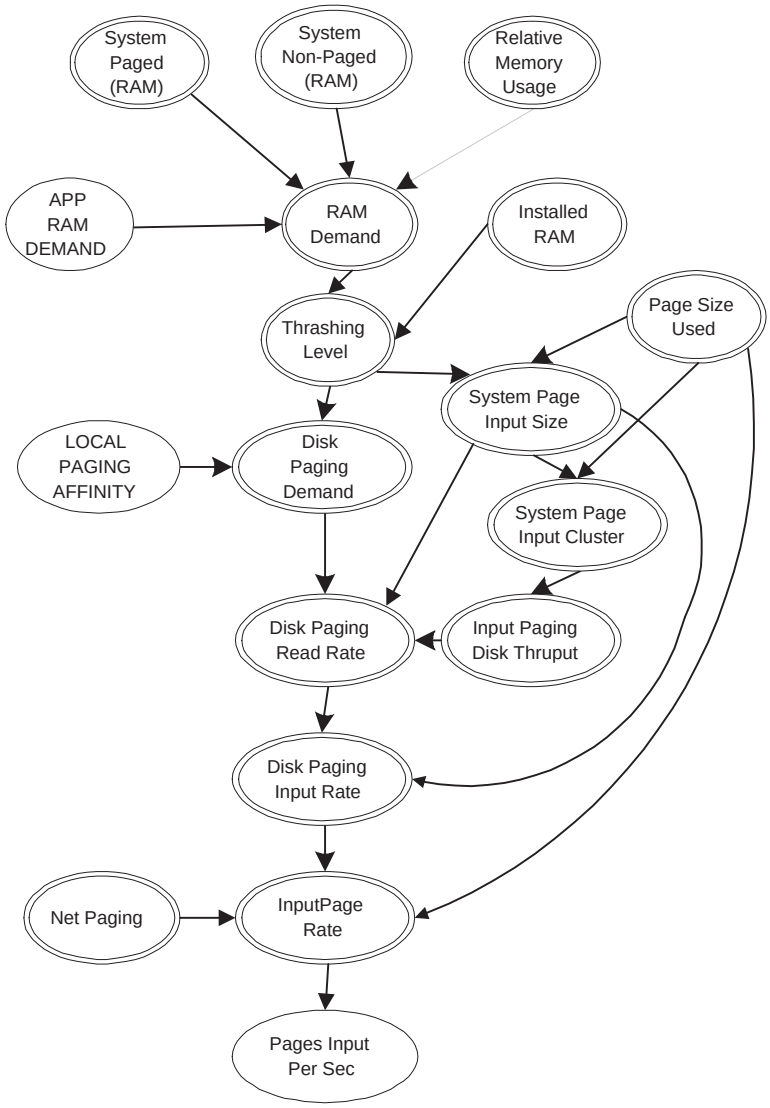

Figure 1: A fragment of the operating system functional model. Variables in double ovals are deterministic functions of their predecessors (perhaps not shown in this diagram) or fixed based on system configuration. Variables shown in single ovals are random variables conditioned on their predecessors.

RamDemand: (RelativeMemoryUsage * (SystemNonPaged + SystemPaged)) + ApplicationRamDemand ["megabytes"]

// The key formula for mapping RAM requirements.

ThrashingLevel: 1 / ((1 + exp (min (500, (InstalledRam − RamDemand)^3)))) ["fraction from 0 to 1"]

// These formulae determine input page cluster size.

SystemPageInputSize: max (PageSizeUsed, 2.5 * PageSizeUsed * ThrashingLevel) ["bytes"]

SystemPageInputCluster: PageSizeUsed * (int ((SystemPageInputSize + PageSizeUsed) / PageSizeUsed)) ["bytes"]

// This formula determines maximum paging rate.

InputPagingDiskThruput: TableLookup (SystemPageInputCluster, RandomDiskThroughput, PagingDiskIndex) ["bytes/second"]

// These formulae determine actual input page rate.

DiskPagingDemand: LocalPagingAffinity * ThrashingLevel ["fraction from 0 to 1"]

DiskPagingReadRate: DiskPagingDemand * (InputPagingDiskThruput / SystemPageInputSize) ["operations/second"]

DiskPageInputRate: DiskPagingReadRate * SystemPageInputSize / PageSizeUsed ["pages/sec"]

InputPageRate: DiskPageInputRate + NetPageInputRate ["pages/sec"]

// The Performance Monitor counter predicted.

Memory.PagesInputPerSec: InputPageRate +
((DiskReadFileByteRate + NetReadFileByteRate) / PageSizeUsed) ["pages/second"]

The complete model as of this writing contains some 300 such formulae.

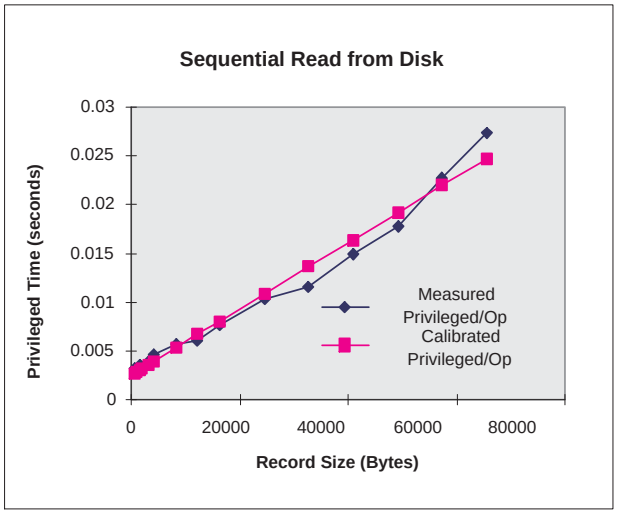

Figure 2: Calibration of sequential reads for a range of record sizes.

3.1 Calibration

In applying the model it is necessary to calibrate the hardware resource maximum bandwidths. This is ideally done on the system under test. Calibration is carried out while the system is otherwise idle.

A synthetic workload generator is used to apply known workloads for calibration. Since workloads of this type are simple and the programs which create them could be quite small loops, the synthetic workload generator is expanded artificially to touch lots of memory in order to exercise the memory cache subsystem as real application workloads do. Neglecting this point would lead to unrealistically high maximum throughput measurements, and incorrect approximation of operating system atoms on the system under test. Nonetheless the deviations between actual workload usage of the memory cache subsystem and that of the synthetic workload generator add uncertainty to the calibration results.

The limits of throughput are collected for each resource over a range of key parameters, and placed into a data base for later extraction. When possible, simple linear regressions are performed to extract parameters for resource characteristics. Figure 2 illustrates one such regression. The processor overhead for disk operations is greatly dependent on the type of disk controller installed in the system. The fit is good, but deviations introduce additional uncertainty into the model.

App Workload	Counters
Inter-operation cpu times:	System.PctPriv
–Sequential Write	System.PctUser
–Sequential Read	System.SystemCallRate
–Random Read	Disk.DiskReadByteRate
–Random Write	Disk.DiskReadRate
Sequential Read Size	Disk.DiskWriteByteRate
Sequential Write Size	Disk.DiskWriteRate
Random Read Size	Cache.CopyReadHitsPct
Random Write Size	Cache.CopyReadsPerSec
Random Read Extent	Cache.LazyWritePgsPerSec
Random Write Extent	Memory.PgFaultsPerSec
RAM Demand	Memory.CacheFaultsPerSec
	Memory.PagesInputPerSec
	Memory.PagesOutputPerSec

Table 1: Workloads and counters that are incorporated into the current model.

3.2 Verification

Before we can use the model we must verify its accuracy. Let $\vec{w}$ be a vector of workload parameters of length m. Let $\vec{c}_p = f(\vec{w})$ be the vector of predicted counter values of length n and $\vec{c}_a$ are the corresponding actual counter values. The application workload parameters ($\vec{w}$) and counters ($\vec{c}_a, \vec{c}_p$) that are in the model are listed in Table 1.

The synthetic workload generator is used to construct a series of one-dimensional workloads. Each generated workload exercises a single system service, varying a key workload parameter w_i, while holding the others fixed. Such a workload might be the sequential reading of a file from disk, with w_i being the size of the record read, and taking on a sequence of increasing record sizes. These synthetic workloads are applied to a system, and the vector of actual performance counters, $\vec{c}_a$, are logged for each value assumed by the key workload parameter w_i.

An identical series of one-dimensional workloads is then applied to the model, and the predicted performance counters $\vec{c}_p$ are recorded for each level of w_i . The results of the model's predictions are compared to the actual performance counters from the real system. Figure 3 illustrates a comparison of a particular actual counter c_a^j to a corresponding model predicted counter value c_p^j over a series of values assumed by a key workload parameter w_i.

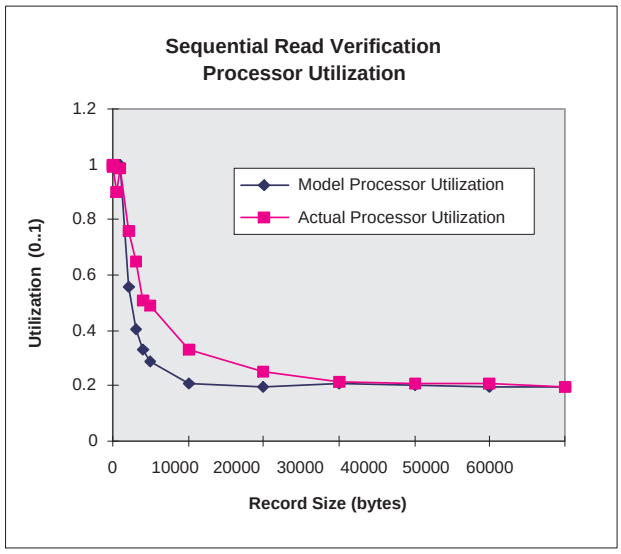

Figure 3: The initial verification for Sequential Reads

3.3 Model Refinement

Verification results can be used to refine the model specification. For example we were concerned about the deviation in Figure 3 of model predicted processor utilization from that observed when the workload was applied to the real system. Examination of the atomic model intermediate values showed that the major component of processor demand was the processor being used by the operating system to read the data from the disk. In the model which produced Figure 3 this was expressed as a linear function of the application's read size in bytes.

A more refined model of this activity can be obtained by first regressing operating system processor usage against the size of a read from disk, as shown in Figure 2. The operating system processor usage for reading from disk during sequential reads can then be determined from this regression by evaluating the regression formula at the size of the read-ahead record used by the operating system. The read-ahead size depends on the operating system kernel in Windows NT and not on the application record size. The operating system overhead for each read-ahead multiplied by the ratio of application read size to system read-ahead size gives the processor overhead for sequential reading.

This new model produces the more accurate verification depicted in Figure 4. Although the refinement is an improvement over that shown in Figure 3, it is not perfect and the discrepancy is a continuing source of uncertainty.

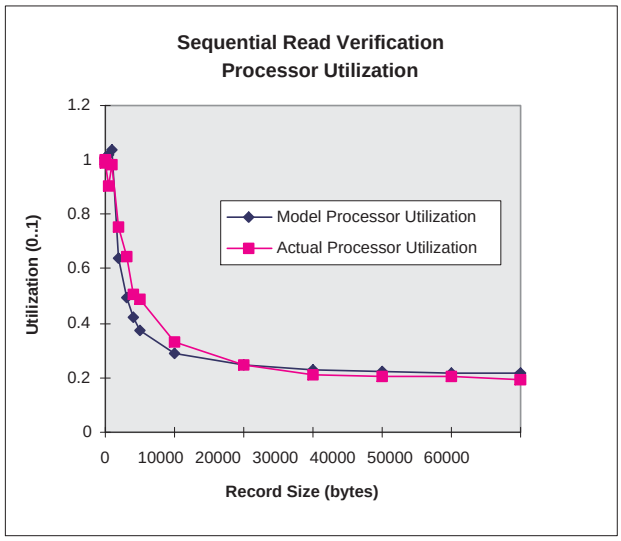

Figure 4: A revised verification for Sequential Reads

One important initial question concerned the robustness of the model with respect to other computer systems than the one on which it is initially tested. As of this writing we have verified the model to acceptable levels across a number of systems, including an Intel 486, an Intel Pentium, and a DEC Alpha, all running Windows NT.

4 Inference

During inference we wish to determine those values of the workload parameters that best explain the observed performance counter values. As before, $\vec{w}$ is a vector of workload parameters of length m and $\vec{c}_p = f(\vec{w})$ is the vector of predicted counter values of length n. The vector $\vec{c}_a$ are the actual counter values corresponding to $\vec{c}_p$. The function f captures the dependence of the counter values on the workload and has been described in Section 3. In this section we describe our method for reasoning with this model, that is finding the value of $\vec{w}$ that best explains $\vec{c}_a$.

From a probabilistic perspective, the best explanation of the observed counters is that workload assignment with maximum probability given the data, that is the assignment that is the most probable explanation (MPE) for the observations. We will use a generalized version of least squares to find this most probable assignment.

The uncertainty structure for this problem is shown in Figure 5 as a belief network. The network represents the conditional independencies we have asserted in this domain

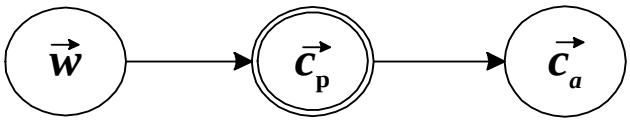

Figure 5: Belief network structure for operating system model.

[Pearl, 1988]. The deterministic functional relationship between workload ($\vec{w}$) and predicted counters ($\vec{c}_p$), the atomic model, is reflected in the double-oval representation in Figure 5. Note that the problem is modelled causally as follows: application workloads "cause" a set of predicted counter values, which in turn "cause" (with noise) the actual counter observations.

Uncertainty in the belief network is characterized by $\Pr(\vec{w}|\xi)$, the prior distribution of workload parameters, and $\Pr(\vec{c}_a|\vec{c}_p,\xi)$ the uncertain relationship between the predicted and actual counter values. In these expressions, ξ is background information including such factors as the installed hardware and the version of the operating system software[1].

The MPE assignment $\vec{w}^*$ is that set of workload parameters $\vec{w}$ that has the maximum probability given the observed counters, that is

$$\Pr(\vec{w}^*|\vec{c}_a,\xi) = \max_{\vec{w}} \Pr(\vec{w}|\vec{c}_a,\xi) \tag{1}$$

Given the belief network model, we can write the joint probability of the variables of interest as a product of conditionals:

$$\Pr(\vec{w},\vec{c}_p,\vec{c}_a|\xi) = \Pr(\vec{c}_a|\vec{c}_p,\xi)\Pr(\vec{c}_p|\vec{w},\xi)\Pr(\vec{w}|\xi)$$

We can integrate over possible predicted counter values as follows.

$$\begin{aligned}\Pr(\vec{c}_a,\vec{w}|\xi) &= \int_{\vec{c}_p} \Pr(\vec{c}_a|\vec{c}_p,\xi)\Pr(\vec{c}_p|\vec{w},\xi)\Pr(\vec{w}|\xi)d\vec{c}_p \\ &= \Pr(\vec{c}_a|f(\vec{w}),\xi)\Pr(\vec{w}|\xi) \end{aligned} \tag{2}$$

since $\Pr(\vec{c}_p|\vec{w},\xi)$ is a unit impulse at $\vec{c}_p = f(\vec{w})$. By the rules of probability and substituting Equation 2 we have

$$\begin{aligned}\Pr(\vec{w}|\vec{c}_a,\xi) &= \frac{\Pr(\vec{c}_a,\vec{w}|\xi)}{\Pr(\vec{c}_a|\xi)} \\ &= \frac{\Pr(\vec{c}_a|f(\vec{w}),\xi)\Pr(\vec{w}|\xi)}{\Pr(\vec{c}_a|\xi)}\end{aligned}$$

[1]Recall that model parameters relating to inherent hardware and software speed on a particular machine and release of the operating system are fixed during calibration.

Since the denominator is a constant in any particular case, Equation 1 becomes:

$$\Pr(\vec{w}^*|\vec{c}_a, \xi) = k \max_{\vec{w}} \Pr(\vec{c}_a|f(\vec{w}), \xi) \Pr(\vec{w}|\xi) \tag{3}$$

where k is a constant.

In evaluating this expression, we made two sets of assumptions. First, we assumed that the workload parameters are marginally independent, implying that $\Pr(\vec{w}|\xi) = \prod_i \Pr(w_i|\xi)$. In various experiments, we have assumed these parameters to be either uniformly, lognormally, or beta distributed, or a mixture of these. For example, with lognormally distributed workloads we have:

$$\Pr(w_i|\xi) = (\sigma_i(w_i - a_i)\sqrt{2\pi})^{-1} e^{-(\ln(w_i - a_i) - \mu_i)^2/2\sigma_i^2} \tag{4}$$

where μ_i, σ_i, and a_i are the logarithmic mean, standard deviation, and minimum value respectively for workload component $w_i, i = 1 \ldots m$. The values of these parameters are provided by direct assessment from an expert. In choosing the distribution we have found it important not to preclude possible workloads by assigning them vanishingly small probabilities.

Second, we assume a multivariate Gaussian error model, that is $\vec{c}_a = f(\vec{w}) + \vec{\epsilon}$ where $\vec{\epsilon} \sim N(\vec{\mu}_\epsilon, \Sigma)$ and $\vec{\mu}_\epsilon$ is the n-dimensional vector of mean errors and $\Sigma = (\sigma_{i,j})$ is a symmetric n by n covariance matrix [DeGroot, 1970]. We estimate the mean errors and covariance from a sample of known workloads, model predictions, and actual counter values on the target system. Using techniques from [DeGroot, 1970], we can update the parameters of the error model by assuming that the distribution for $\vec{\mu}_\epsilon$ is multivariate normal and the distribution for Σ^{-1} is Wishart.

For purposes of this study, we will estimate $\vec{\mu}_\epsilon$ using the sample mean error and estimate Σ with the sample covariance from a set of verification samples. Using procedures similar to those applied during verification, we can run a set of controlled experiments on the target machine to generate model error data. For a set of sampled known workloads, we generate model predictions (using the calibrated model) and collect actual counter values. This yields sample data for estimating the covariance matrix.

Since the actual error given a set of workload parameters is just $\vec{c}_a - f(\vec{w})$, then the probability of the observed counter values, given the model f and $\vec{w}$ is calculated as follows.

$$\Pr(\vec{c}_a|f(\vec{w}), \xi) = ce^{-1/2(\vec{c}_a - f(\vec{w}) - \vec{\mu}_\epsilon)^T \Sigma^{-1} (\vec{c}_a - f(\vec{w}) - \vec{\mu}_\epsilon)} \tag{5}$$

where $c = (2\pi)^{-n/2}|\Sigma|^{-1/2}$. Equations 4 and 5 are used to evaluate Equation 3.

Evaluation of Equation 3 is composed of a prior term $\Pr(\vec{w}|\xi)$ and a liklihood term $\Pr(\vec{c_a}|f(\vec{w}),\xi)$. The prior term captures prior knowledge regarding application workloads, for example the typical program memory footprints for code and data, and sizes of sequential reads.

The liklihood term measures the how well the predictions match the observations. Under the assumption of multivariate normality, the magnitude of the covariance terms reflect the scaling and precision of the various counters. If we restrict the covariance matrix to zeros in the off–diagonal elements ($\sigma_{i,j} = 0, i \neq j$) then the term in the exponent of Equation 5 is the same as in ordinary least squares for the difference between the actual and predicted counters. Maximizing the liklihood is equivalent to minimizing the weighted sum of squares, where the weight is the inverse of the variance for each counter difference. Counter differences that tend to be noisy will get less weight in the probability calculation than more precise counters. The full covariance method is the multidimensional version of least squares accounting for correlated errors.

Unfortunately, we cannot solve for $\vec{w}^*$ analytically due to numerous discontinuities and non-linearities in the model f. The discontinuities arise in discrete shifts in operating system algorithms, such as differences in file system implementation when the record size is a multiple of the page size. Therefore we use numerical methods, such as Newton's Method (see e.g., [Press et al., 1992], to search for the $\vec{w}^*$ which is the most probable explanation for the bottleneck. Although these numerical optimization methods find local maximum, we have found empirically that the numerically derived solutions are very close to the global maximums for problems for which we know the globally maximum configuration. We have not encountered a case where several significantly diverse workloads recieve the same probability score, though this is an area for future research.

5 Implementation and Results

The Windows NT system model and inference procedure have been implemented in Microsoft Excel. We utilize the Excel Solver feature to provide Newton's Method to search for the desired workload vector $\vec{w}^*$. Using Excel Solver, numerical inference is taking less than a couple of minutes. In order to test the inference procedures, we impose a set of known workload parameters on a given platform and collect actual performance monitor counters. The model is then used to infer the workloads. Our test suite consists of the following cases which vary as to the nature of the bottleneck and its cause:

- Sequential Read

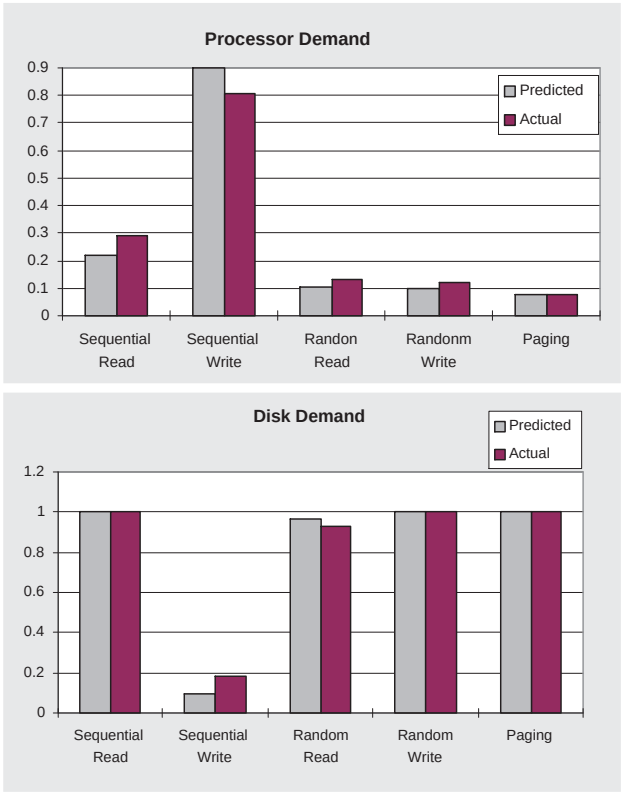

Figure 6: Comparison actual versus inferred processor and disk demands for 5 scenarios.

- Sequential Write
- Random Read
- Random Write
- Paging

We have been able to estimate the error model on a verification set of over 180 cases representing model predictions and actual values for the three different computer systems mentioned. Even with this sparse dataset, performance is encouraging. The Figure 6 shows that for every scenario, the model is finding the correct bottleneck and a good approximation of resource demand.

In Figure 7, we show additional detail regarding resource utilizations for the case of Sequential Reads. Again the actual levels of each variable along with inferred value are shown. Our method is finding the correct solution. We have tested further cases where

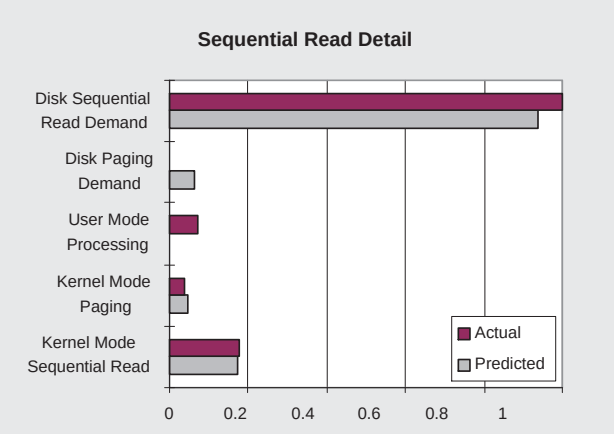

Figure 7: Comparison actual versus inferred device utilizations for Sequential Read.

the workloads are more complex, as well as some real programs, and so far have found the correct solution in each case.

6 Hardware Recommendations

Now that we know the workload which caused the bottleneck, we can apply the model in more conventional ways. The user can supply, for example, the cost and performance characteristics of a variety of hardware upgrades that might be under consideration. The performance characteristics of the anticipated upgrades are characterized using the calibration technique we described earlier. When the equipment has not been purchased and cannot be calibrated, we have provisions for entry of the calibration data estimated by the user from the specifications of the hardware.

The recommendation analysis cycles through the various combinations of hardware upgrade and develops for each one the maximum throughput that will be attained and the cost per transaction for the increase. The unit of transaction is taken as the workload parameter with the highest frequency under the original workload. The user can simply select the upgrade option with the lowest price/performance. Alternatively, the user may have a particular throughput requirement for this upgrade. In this case the user can select the smallest price/performance change which achieves that goal.

Atomic models do not directly compute response times, but they do deliver resource utilization figures. It is possible to apply standard queueing theory to the results of the model to estimate the resulting response times [Blake, 1979].

7 Conclusions and Future Work

There is a natural synergy between the causal Belief Nets of decision theory and atomic models of operating systems. The use of the atomic model permits a simple characterization of operating system resource allocation policies. The atomic models avoid possibly unrealistic assumptions often present in queueing models and are much more economical to build and run than simulation models. That atomic models evaluate rapidly is crucial to our bottleneck detection method because we use numerical search techniques to find the most probable cause of the bottleneck. Decision theory provides a firm conceptual framework in which the model can be used to infer the causes of performance problems. Probablistic characterization of the workload domain combined with the statistics of a learned error model lead to a rapid numerical search for the most probable cause. Knowledge of the cause of the bottleneck, together with the predictive capability of the model, yield a recommendation of the most cost-effective hardware upgrade to resolve the bottleneck.

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. It also can be used by software developers to predict the performance of their application, and to help determine which portions of the program merit additional design and implementation effort.

Finally, since the model includes many variables relating to operating system design and algorithms, this approach can address issues relating to the structure of the operating system itself. This would include off-line design studies, for example, estimating the possible system-wide effects of different paging algorithms. Similar models could also potentially be used for dynamic tuning of system operating parameters, such as cache sizes, in response to inferred application loadings.

Acknowledgments

The authors thank Brian Beckman, Bob Davidson, David Heckerman, Nathan Myhrvold, and Gideon Yuval for useful comments and suggestions.

References

[Accetta et al., 1986] Accetta, M., Baron, R., Bolos ky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. (1986). Mach: a new kernel for UNIX development. In *Proceedings of USENIX Association Summer Conference*, pages 93–112, Atlanta.

[Berry and Hellerstein, 1991] Berry, R. and Hellerstein, J. (1991). An approach to detecting changes in the factors affecting the performance of computer systems. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 39–49, San Diego, California. ACM.

[Blake, 1979] Blake, R. (1979). Tailor: A simple model that works. In *Proceedings Conference on Simulation, Measurement, and Modeling of Computer Systems*, pages 1–11, Boulder, CO. ACM.

[Blake, 1995] Blake, R. (1995). *Optimizing Windows NT*. Microsoft Press, Redmond, WA.

[Bodnarchuk and Bunt, 1991] Bodnarchuk, R. and Bunt, R. (1991). A synthetic workload model for a distributed file server. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 50–59, San Diego, California. ACM.

[Buzen, 1976] Buzen, J. (1976). Fundamental operational laws of computer system performance. *Acta Informatica*, 7:167–182.

[Buzen and Shum, 1987] Buzen, J. and Shum, A. W. (1987). A unified operational treatment of rps reconnect delays. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 78–92, Banff, Alberta, Canada. ACM.

[Custer, 1993] Custer, H. (1993). *Inside Windows NT*. Microsoft Press, Redmond, WA.

[DeGroot, 1970] DeGroot, M. (1970). *Optimal Statistical Decisions*. McGraw-Hill, New York.

[Domanski, 1989] Domanski, D. (1989). A PROLOG-based expert system for tuning MVS/XA. *Performance Evaluation Review*, 16:30–47.

[Dowdy, 1989] Dowdy, L. (1989). Performance prediction modeling: A tutorial. In *Proceedings Sigmetrics and Performance '89 International Conference on Measurement and Modeling of Computer Systems*, page 214, Berkeley, California. ACM.

[Ferrari, 1984] Ferrari, D. (1984). On the foundations of artificial workload design. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 149–157, Cambridge, Massachusettes. ACM.

[Ferrari, 1989] Ferrari, D. (1989). 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.

[Graf, 1987] Graf, I. (1987). Transformation between different levels of workload characterization for capacity planning. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 78–92, Banff, Alberta, Canada. ACM.

[Gray, 1987] Gray, J. (1987). A view of database system performance measures. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 3–4, Banff, Alberta, Canada. ACM.

[Heckerman and Wellman, 1995] Heckerman, D. and Wellman, M. P. (1995). Bayesian networks. *Communications of the ACM*, 38(3).

[Hellerstein, 1989] Hellerstein, J. L. (1989). A statistical approach to diagnosing intermittant performance-problems using monotone relationships. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 20–28, Berkeley, California. ACM.

[Hellerstein, 1994] Hellerstein, J. L. (1994). A comparison of techniques for diagnosing performance problems in information systems. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 278–279, Nashville, TN. ACM.

[Irgon et al., 1988] Irgon, A., Dragoni, A., and Huleatt, T. (1988). Fast: A large scale expert system for application and system software performance tuning. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–156, Sante Fe, New Mexico. ACM.

[Pearl, 1988] Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufman, San Mateo, Ca.

[Press et al., 1992] Press, W., Teulosky, S., Vetterling, W., and Flannery, B. (1992). *Numerical Recipes in C*. Cambridge University Press, Redmond, WA.

[Raghavan and Kalyanakrishnan, 1985] Raghavan, S. and Kalyanakrishnan, R. (1985). On the classification of interactive users based on user behavior indices. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 40–48, Austin, Texas. ACM.

[Raghavan et al., 1987] Raghavan, S., Vasukiammaiyar, D., and Haring, G. (1987). Generative networkload models for a single server environment. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 118–127, Nashville, Tennesee. ACM.

[Ritchie and Thompson, 1974] Ritchie, D. M. and Thompson, K. (1974). The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375.

[Yu et al., 1985] Yu, P., Dias, D., Robinson, J., Iyer, B., and Cornell, B. (1985). Modelling of centralized concurrency control in a multi-system environment. In *Proceedings Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 183–191, Austin, Texas. ACM.