

Practical Performance Models for Complex, Popular Applications

Eno Thereska
Microsoft Research

Bjoern Doebel^{*}
TU Dresden

Alice X. Zheng
Microsoft Research

Peter Nobel
Microsoft

ABSTRACT

Perhaps surprisingly, no practical performance models exist for popular (and complex) client applications such as Adobe's Creative Suite, Microsoft's Office and Visual Studio, Mozilla, Halo 3, etc. There is currently no tool that automatically answers program developers', IT administrators' and end-users' simple what-if questions like "what happens to the performance of my favorite application X if I upgrade from Windows Vista to Windows 7?". This paper describes our approach towards constructing practical, versatile performance models to address this problem. The goal is to have these models be useful for application developers to help expand application testing coverage and for IT administrators to assist with understanding the performance consequences of a software, hardware or configuration change.

This paper's main contributions are in system building and performance modeling. We believe we have built applications that are easier to model because we have proactively instrumented them to export their state and associated metrics. This application-specific monitoring is always on and interesting data is collected from real, "in-the-wild" deployments. The models we are experimenting with are based on statistical techniques. They require no modifications to the OS or applications beyond the above instrumentation, and no explicit *a priori* model on how an OS or application should behave. We are in the process of learning from models we have constructed for several Microsoft products, including the Office suite, Visual Studio and Media Player. This paper presents preliminary findings from a large user deployment (several hundred thousand user sessions) of these applications that show the coverage and limitations of such models. These findings pushed us to move beyond averages/means and go into some depth into why client application performance has an inherently large variance.

Categories and Subject Descriptors

C.4 [Performance of Systems]: [Measurement techniques, Modeling techniques, Design studies]

^{*}Work done while interning at Microsoft Research, Cambridge, UK.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMETRICS'10, June 14–18, 2010, New York, New York, USA.
Copyright 2010 ACM 978-1-4503-0038-4/10/06 ...\$10.00.

General Terms

Design, Management, Measurement, Performance

Keywords

Performance variance, what-if, developers, IT administrators

1. INTRODUCTION

Understanding tradeoffs between application performance and hardware, software, and environment configurations is important for developers and IT administrators. These tradeoffs are known as a "performance model". Developers, for example, need performance models to anticipate and iron out performance anomalies for applications that run on diverse configurations. Developers currently rely on in-the-lab tests to obtain these models. For popular applications, which are often used by thousands of clients, data on how the application is behaving "in-the-wild" is not channeled back to them. IT administrators, too, need models to anticipate any performance effects of an upgrade (e.g., moving from one operating system to another, or moving from SCSI disks to SATA disks). Currently, they use their experience, rules of thumb, benchmarks (with well-known limitations [24]) and staged rollouts to make educated guesses on any performance impact. Global experience from other administrators upgrading similar systems is at best passed through word-of-mouth, e.g., from colleagues. Its collection is not automated in any way.

The lack of a global "tradeoff database" is a wasted opportunity that leads to multifaceted frustration: end-users and IT administrators have to resort to vague description of their performance concerns on support web sites; developers miss crucial information about application usage that could guide them towards better testing and improved performance.

This paper focuses on performance models for popular, and often complex, client applications, for example Adobe's Designer suite, Microsoft's Office suite and Visual Studio, Mozilla, etc. Unlike enterprise applications, the applications we target do not have explicit service-level agreements (SLAs). Their performance is usually judged relative to how other peers are performing, as long as they remain responsive to the user. Thousands to millions of people use popular applications, thus the hypothesis of this paper is that the law of large numbers helps in their modeling. We postulate that these applications can be modeled implicitly, by a tool that (carefully) observes their behavior while deployed. We believe addressing this problem is both important (impact would be felt by millions of users) and technically challenging.

The modeling infrastructure we are working on, called App-Model, monitors the behavior of an application on a machine, and how the same application behaves under different configurations

on different machines. AppModel collects performance signatures for each application, which include not only OS-specific metrics (usually collected in state-of-art approaches like [4, 9]), but also application-specific performance counters and characteristics of the workloads the application is processing, e.g., file size, file type, image resolution, etc. We show the necessary (small) changes that can be released as a patch to enable these applications to export these metrics. AppModel then builds performance models automatically, using regression techniques, and helps with their interpretation.

1.1 Usage scenarios

Here we illustrate how AppModel could be used by application developers and system administrators.

Application Developers: A developer might ask AppModel “what happens to performance of application X if I use configuration C ?” C might be a set of configuration attributes the developer is interested in, e.g., $C = \{\#CPUs = 4, CPUarchitecture = 64bit, OS = Windows7, \dots\}$. The developer might have tested C in the lab, but wants to know the performance experienced by real users. Or perhaps the testing has not been done yet, e.g., C might be expensive to set up and the developer wants to know if there is a problem before making the purchase.

AppModel takes C , iterates efficiently over all existing configurations in the field similar to C , and provides a distribution on how users with C experience performance (end-to-end, or for individual application *states*, e.g., for application startup). If the results are unexpected, the developer might proceed to replicate them. For certain configurations that exhibit large performance variance, the developer might want AppModel to do a delta analysis of performance metrics from the best and worst performing scenarios to understand why. Sometimes the performance variance might be explainable, e.g., high contention from other applications. If not, the developer might want to initiate a performance debugging session.

IT administrators: An administrator might ask AppModel “what happens to the performance of the top-10 popular applications I support if I migrate to a new release of an OS?”. The administrator might also ask what-if questions to understand the effect of hardware upgrade decisions (e.g., “What happens to performance if I double the amount of memory?”) or to understand the effect of configuration changes (e.g., “What happens to overall performance if I use a particular virus scanner?”). The administrator simply poses these questions without having to purchase the new hardware or software.

AppModel is designed to continuously collect local historical data, before the question is asked, on how users this administrator is responsible for interact with the top-K applications. This data could include which buttons they click on most, or which states dominate end-to-end performance. AppModel takes that data together with the desired new configuration C' , which includes all existing attributes plus the new OS version number and consults global data on how users with configurations similar to C' perceive their application performance. AppModel then makes a prediction to the administrator. The prediction takes the form “average end-to-end performance will not degrade or improve, but performance variance will increase by 10%”. The IT administrator would then decide on the upgrade.

1.2 Contributions

This paper describes the challenges and results from making such a modeling framework a reality. We make the following contributions: first, we develop a performance modeling framework for popular client applications. This framework does not rely on complex understanding of the applications or operating system. Sec-

ond, we show how we have instrumented several popular applications to export necessary state metrics and workload characteristics. Third, we perform a feasibility study for AppModel using two approaches. Initially, we present preliminary results from a large user deployment (hundreds of thousands of users), but with a limited set of performance metrics collected. The data from this deployment is in active use by real developers and testers of the applications under consideration. For sensitivity studies and validation we use synthetic tests from a handful of users with a much larger set of metrics collected.

2. MOTIVATION AND RELATED WORK

Understanding application performance is important for developers in our company. We also wanted to understand how our modeling infrastructure could potentially help IT administrators. Hence, we conducted a small survey (the raw survey results can be found at [1], below we distill what is relevant for this paper).

2.1 IT administrator survey

We asked members of the Usenix SAGE and System Administrators LinkedIn mailing lists to answer some questions on system performance. We received replies from 38 system administrators, 24 of whom maintain large system setups with more than 40 machines. 28 of the participants have 5 or more years of experience. When asked to rank their main concerns, understanding and fixing performance problems ranked as high as all other concerns, including keeping software up-to-date, applying security patches and backing up the data.

Figure 1 shows the results for three performance-related questions to the administrators. When asked whether they knew the potential performance impact of a software or hardware upgrade beforehand, 16 of the 38 administrators answered “no”. We then asked the administrators about how they perceived and understood performance. Most of them said they were using some form of rules of thumb, based on their own or their colleagues’ experience, when trying to assess performance. Many of them (18) also use simple benchmarks (e.g., *bonnie++*, *httperf*, *lmbench*) and monitoring tools to assess system behavior. It is well-known that benchmarks are not good for addressing application-specific performance concerns [24]. Monitoring tools (e.g. *top*, *iostat*) and tracing tools (e.g., *rrdtool*, OS performance counters, *SystemTap*) provide measurements and visualization of the *current* system configuration and performance. They cannot be used to flag anomalous behavior among computers with the same configuration. They can also not be used to predict performance under different configurations. Several administrators’ understanding of performance was based on end-user experiences, which referred to complaints aggregated by email and web-forms.

The results confirmed that performance is important, and that the administrators made extensive use of local knowledge. The administrators did not have any framework or tools to automatically aggregate global knowledge beyond complaints on emails and web-forms. It is important to note that this is a small survey and it may be affected by self-selection, bias of the volunteers and coverage problems. Nonetheless, we believe the results complement those of other administrator surveys (e.g. [11] on failures during upgrades), though our focus is on performance.

2.2 Contrast with related work

We contrast our work to related work along five axis.

Performance modeling vs. performance debugging: The primary contribution of this paper is on novel, practical ways to do performance modeling, with a secondary focus on performance de-

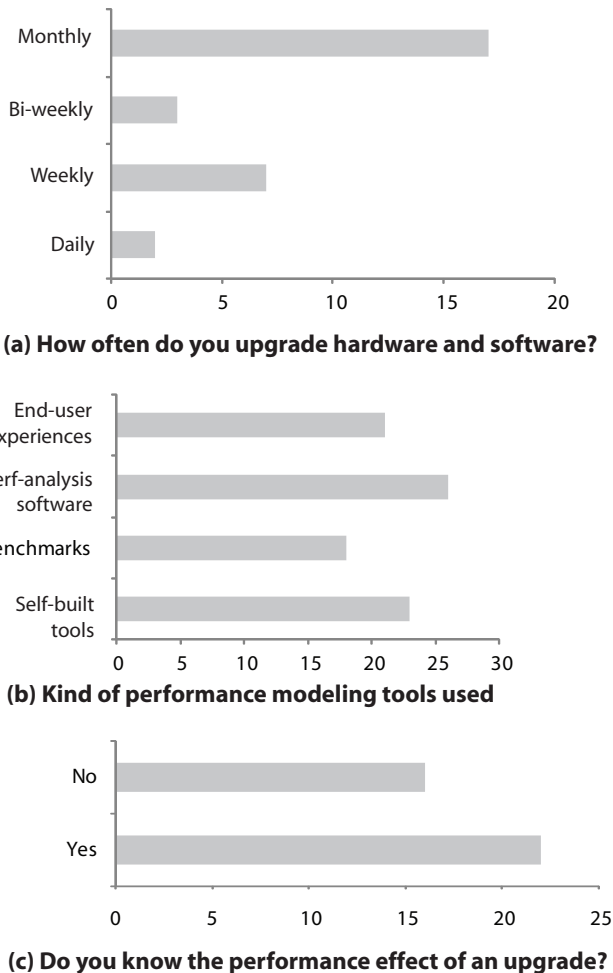


Figure 1: A survey of IT administrators.

bugging. Performance modeling answers what-if questions like the ones above. Performance debugging analyzes performance discrepancies. Most related work has focused on performance debugging [2, 3, 4, 5, 8, 9, 10, 22, 23, 25, 26]. Performance modeling helps performance debugging by providing a notion of "expected" behavior. After a what-if question is answered and the recommendation enacted upon, performance might not be what it was expected. To help understand why, AppModel matches the state-of-art [4, 9, 10] by performing a delta analysis between application deployments that perform well and those that do not.

Black-box vs. white-box modeling: There has been considerable recent work done on performance modeling (e.g., see [14, 18, 27]) which augments a well-known body of literature [15, 17]. The spectrum of modeling approaches ranges from "white box" to "black box". Our approach falls in-between. We use developer knowledge to understand what parts of an application to instrument (we have full control over our applications), and black-box, statistical algorithms to answer what-if questions. Some of these algorithms involve, for example, regression and linear least squares fits. We do not make any new contributions in statistical methods but we report on subtleties in making them work in our context.

System building vs. measuring: This paper is interested in how to build applications that export relevant state and performance metrics. The systems we consider can be legacy systems, but we

expect them to be actively maintained, i.e., patches can be issued at any time. We will require some minor changes to the applications. Indeed, all the real applications we consider in this paper made the changes required incrementally. Hence, we go beyond merely measuring what is there (usually metrics exposed by the OS [4, 9, 10]) to actively building applications to provide more insights.

Methodology: Most of the above performance debugging papers establish a level of normalcy by observing *one* system over time. AppModel observes *many* systems over time. Hence, AppModel can explore how an application would run under different configurations. The approach of using observations from many systems is most closely related to PeerPressure [29] and Clarify [13]. Both share our method of leveraging deployed machines for statistical analysis, but both have an orthogonal goal of troubleshooting computer misconfigurations and errors. Performance modeling has other challenges. Performance is time-varying, and depends on interactions among many configurations (hardware and software) as well as workload characteristics.

There are existing performance tradeoff exploration techniques, which range from in-the-lab scenario-based tests and benchmarks to staged upgrades. Staged upgrades are used to gradually roll out a new upgrade with minimal impact to the users [11]. Fundamentally, staged upgrades require the new hardware or software to be purchased and do not allow administrators to explore answers to what-if questions *before* committing to buying anything.

Targeted systems (and non-targets): We target popular client applications with no particular service level agreements. Applications that can be exhaustively tested in-the-lab will not benefit from AppModel. Our approach will not work well for highly-customized applications or those with few deployed instances, for example financial applications for large banks with only tens of deployments, or applications which only run in a handful of data centers [5, 27]. This is because no statistical confidence can be obtained from observing only a handful of deployments and configurations.

3. APPROACH

3.1 Overview and challenges

Figure 2 illustrates the architecture of AppModel. From a top-down point of view, an administrator or developer poses what-if questions to AppModel. These questions take the form "What is P for application A , for configuration C ?" P is a performance metric (throughput, latency, full distribution or average), A stands for the name of the application, and C is a set of computer hardware and software configuration attributes/parameters. AppModel returns P .

From a bottom-up point of view, AppModel has several logical steps. First, a tool continuously monitors the local performance of each popular application. There are two challenges here: deciding how to define an application's performance and deciding what parts of the application to monitor. Unlike simplistic benchmarks, an application might have hundreds of states which consume different resources. For interactive applications like Microsoft Office, Adobe Photoshop, etc. the state is dictated by the history of user's button clicks. AppModel monitors the time spent in that state, resources consumed and characteristics¹ of the workload, e.g., the size of the file being worked on. In addition, for each hardware resource, its utilization by other applications is also recorded.

Second, the local information is sent to a central database where it is aggregated. Our current implementation places this database with the company that produces the application and cares about un-

¹In this paper we use the terms workload "characteristics", "attributes" and "parameters" interchangeably.

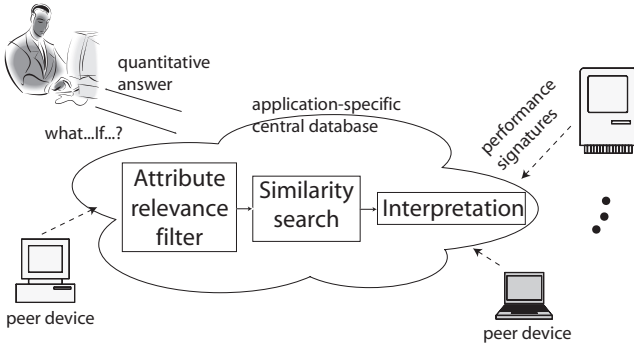


Figure 2: A developer or administrator poses what-if questions to AppModel. AppModel continuously collects performance signatures on how peer devices are performing. AppModel then answers what-if questions quantitatively and provides fidelity metrics with them.

derstanding its performance (e.g., Adobe, Google, Microsoft, etc.) If the administrator cares about many applications, s/he will have to send separate what-if queries to the respective databases. The main challenge is to maintain privacy. Users can choose to opt-in and periodically transmit performance signatures. The exact privacy agreement for our deployment is shown here [20]. Performance signatures do not contain any user-identifiable information, as will become clear in this section.

Third, a statistical model is built for each what-if question. Logically, the modeling component has several parts, each with subtle complexities that do not have simple "off-the-shelf" solutions. We only give an informal overview here. A first challenge is to determine for each what-if question the set of most relevant configuration attributes that have traditionally had strong correlation with performance. This set is likely to be different for different what-if questions. We construct an attribute relevance filter that filters out the irrelevant attributes. A similarity search module then searches for machines with configuration most similar to the initiator's machine. The intuition here is that it is very likely someone already has the configuration the initiator desires. To perform the similarity search it only considers the relevant resources indicated by the attribute relevance filter. The behavior of the application under consideration is analyzed for each of the most similar machines. The average performance, as well as the entire probability distribution, is returned to the initiator.

A final challenge is interpreting the results and differentiating between expected and unexpected variance in performance. The interpretation module is triggered if the initiator wants to perform a delta analysis between the worst and best-performing machines for a given what-if question. The interpretation module performs the analysis automatically and returns a set of attributes for the initiator to examine, together with annotations on whether the delta indicates an anomaly.

3.2 Performance signatures

We borrow the term *performance signature* from Cohen et al. [10]. In this paper we extend it to mean not only a collection of operating system (OS) performance counters, but also a collection of application metrics, workload attributes and general computer configuration attributes. AppModel collects signatures during an application *state*. A state can be defined at different semantic levels. A single method can be thought of as a state. For interactive applications that require, for example, the clicking of a button

on the application menu, a state can be defined as the collection of methods invoked between the time a button is clicked to when the action completes. Different states might have different resource consumption needs, e.g., one state might be CPU-bound (e.g., when performing an arithmetic calculation), another might be disk-bound (e.g., when saving a file to disk). Ignoring these distinct states and treating an application as an "average" of its states provides little insight on what the performance model means.

AppModel requires developers to expose the time it takes to be in a state. Developers can do so by recording a timestamp at the entrance and exit points of a state. Figure 3 illustrates an example instrumentation for a popular spreadsheet application (Microsoft Excel) for the "File Save" operation. The library that collects the data is straightforward and can be released as a small patch to a legacy application. As described in Section 4, many user-visible states in our applications are instrumented using this approach.

The application should also expose relevant workload attributes. For example, the time to complete a "File Save" operation probably depends on the type and size of the file. Exposing the right workload attributes can be difficult and requires developer experience. This process cannot be automated and is iterative, but AppModel can help. After the developer exposes several attributes, AppModel automatically correlates them with performance and gives feedback to the developer on the attributes' relevance. Developers remove attributes that are found irrelevant in the long term.

Most operating systems provide basic utilization performance counters per application, per resource, such as "disk writes/second" or "disk queue length". We call these counters "dynamic attributes" throughout the paper because they are time-varying. We also collect general computer configuration attributes, e.g., "disk type" or "OS version". We call these attributes "static".

3.3 Formalizing the modeling problem

For modeling purposes, we define an application session as a sequence of states i (with possible repetition of states, e.g., *File Open, File Open, File Close, File Close*) from the time a user starts the application until the user quits it. Let L_i be a random variable representing the latency incurred in state i . Overall application performance P could thus be written as $P = \sum_{i=1}^n L_i$. There are two details however. State i might be partially concurrent with state j and the overlap time needs to be subtracted. Also, L_i might depend on L_j , i.e., in practice states might not be independent. The data we collect allows for a conditional probability table (with entries $L_i|L_j$) to be populated.

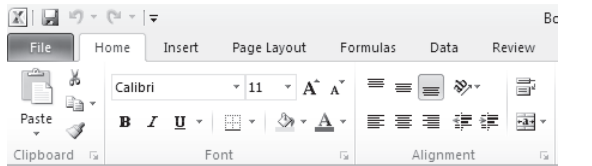
Answering a what-if question means providing P for an application A for a given configuration C . The next sections show how L_i , and thus P , is related to C . The applications we model are interactive, thus the focus on latency. The same approach would work for modeling throughput, where L_i would represent throughput.

3.3.1 Attribute relevance filter

The relevance of an attribute a_i in the configuration C depends on the what-if question. For example, the answer to a question concerned with upgrading hard drives is more likely to be correlated with attributes such as "memory size" and "average disk queue length" than with attributes such as "CPU speed" and "network queue length". Thus, L_i is a function of the performance characteristics, or attributes, of that application and its resources:

$$L_i = F(a_1, \dots, a_n) \quad (1)$$

It is this function F that AppModel approximates globally using peer machines. The data available to make the approximation is like the table part of Figure 3. The table is large however, as it con-



Static attributes		Dynamic attributes		Application attributes		
OS	#CPUs...	CPU util	disk util ...	save time	file type	file size
Win 7	4	10%	20%	100ms	.xls	4MB
Win 7	4	0%	10%	150ms	.xls	8MB
Win 7	4	60%	80%	180ms	.xls	1MB
...

Figure 3: Three possible performance signatures (captured in three rows) when users click three times on Microsoft Excel’s File Save button. File save time is what is being modeled as a function of all other attributes (columns).

tains data from all reporting users. AppModel uses a classification and regression tree (CART) [7] to do the approximation. CART is similar to the TAN models used by Cohen et al. [9], and we believe either can be used, but we have experimented with CART only. CART assigns each attribute an entropy-based weight that determines how correlated that attribute is with performance. We provide a brief sketch below of how this is done.

CART induces a model as follows. For each attribute a_i in the data collected, CART determines the entropy of the observed data when that attribute is chosen to split the data. CART chooses the attribute that leads to the lowest entropy. Intuitively this means that data is partitioned into bins where each bin has as little variance in its values as possible (in the best case, all the values in the bin are the same). CART continues this process recursively, until all attributes have been placed in a tree. This process implicitly associates a weight v_i with each attribute a_i . The default weight is defined as the normalized information gain of that attribute [21](pages 55-60). We have found working with CART is an iterative process, far from the ideal of being fully automatic. It involves a trial-and-error process with adding or removing some attributes that, in our context, stems from the standard correlation vs. causality problem (an example is given in Section 5.2). Mitchell describes several such open questions in his book [21](chapter 3). We do not discuss them further in this paper.

Figure 4 shows a portion of a tree that models high-level application throughput as a function of several attributes. The application itself is described in Section 4.2. The top-3 attributes CART has chosen are “Write bytes/s”, “Read bytes/s” and “CPU utilization” of the process. Hence, the disk resource and CPU resource are most related to throughput, and the other 100+ attributes can be ignored. Overfitting of regression trees is a concern [21](page 66). Ideally, the regression trees should not be too deep. Pruning and cross-validation, which are standard techniques against overfitting, work well, and as Section 5.4 shows, the depth of trees can be small (less than 10 levels deep).

Training time for CART is $O(t \cdot \text{Height}(t))$ and predictions take $O(\text{Height}(t))$ time, where t is the number of observations (i.e., the number of rows in the global table in Figure 3) and $\text{Height}(t)$ is the depth of the regression tree. CART can handle mixed-type attributes (e.g., continuous and categorical values) and can learn incrementally from new data. Furthermore, the structure of a CART tree can often be human-interpretable, although in practice we find that the human has to be a domain expert.

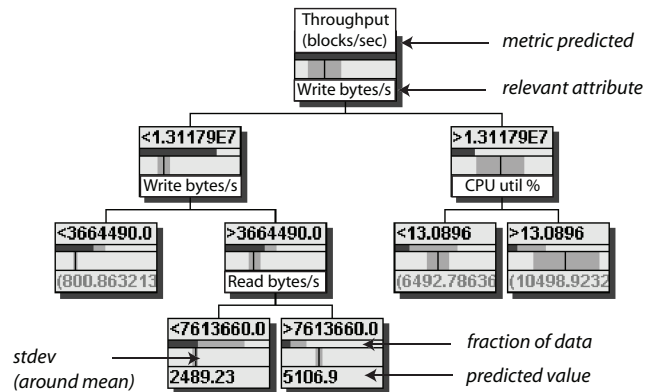


Figure 4: Example CART-based attribute filter. Visualization is done with a regression tree viewer [6].

3.3.2 Similarity search

After filtering irrelevant attributes, several subtle steps take place next. First, *dynamic* attributes are replaced by their *static* counterpart. A dynamic attribute usually has many static attributes associated with it. For example, if CART picks the dynamic attribute “CPU utilization” as relevant, AppModel converts that to the static attribute(s) “CPU type” and “CPU speed”. To avoid introducing new notation, we call static attributes also a_k .

Dynamic attributes are not used in this step, because comparing them requires a further normalization function. For example, should a machine with a 2GHz CPU and 10% utilization be equivalent (with respect to application latency) to a machine with a 1GHz CPU and 20% utilization? Theoretically, this question can be answered if one has much training data covering many resource utilization scenarios. With the current deployment we have good coverage of static attributes, but not dynamic attributes. Hence, it remains future work to verify whether such normalization is possible. We exploit dynamic attributes only for the interpretation phase (Section 3.4) and use static attributes for similarity search.

Second, the top- K most relevant static attributes are selected and their local weights (calculated as the maximum weight of the dynamic attributes) are averaged to assign a global weight. To avoid introducing new notation, we call this global weight also v_i , but the context will make it clear whether the weight is local or global. K is determined empirically. Sensitivity studies indicate that K is usually less than 20.

After selecting the most relevant static attributes and assigning a weight to them, we define the distance between two computer configurations C_i and C_j as:

$$D(C_i, C_j) \equiv \sqrt{\sum_{k=1}^n v_k d(a_k(C_i), a_k(C_j))^2} \quad (2)$$

where $a_k(C_i)$ denotes the value of the k^{th} attribute of configuration C_i . The function d defines a (normalized) distance metric among attributes. The choice of a distance metric is subjective. For example, what should be the distance between two operating system versions (the “version” attribute)? One possibility is to set the difference to 0 if the versions are identical and 1 otherwise. Another possibility is to have earlier versions incur a larger distance from the current version than the latter versions. In theory, cross-validation could automatically select the best distance function among several candidates [21](page 235). The mileage varies however, and is a function of how many test cases are available. For some resources

we do the obvious subtraction (e.g., CPU frequency), for others we subtract the version number (e.g., OS version).

We are also experimenting with having a set of standard tests assign a score to certain resources. These tests benchmark resources either at OS install time or whenever a user explicitly instructs the OS to do so. The exact method to do that currently ships with the Windows OS, and forms part of the Windows Experience Index [30]. The benchmarks represent common workloads the hardware devices are expected to handle well. For example, to normalize two hypothetical disks, one from Seagate and one from Hitachi, a benchmark that reads and writes sequentially from the disk is run on the disks and the throughput is measured. The throughput is then normalized to a score between 1 and 7.9 (the score range might change with subsequent releases of Windows). It is important to note that these benchmarks are written by domain experts. For example, storage experts know that to characterize a disk one needs to measure at least streaming and random-access throughput and latency. Other measurements might be important too (e.g., the above metrics as a function on number of requests in the disk queue) and might be added in future releases.

In theory, these tests are not needed for our modeling framework, since thousands of users “test” their hardware by virtue of running their applications. However, in practice, these tests establish a notion of normalcy and expected behavior with higher confidence than statistical methods. These tests are used to bootstrap the similarity search problem by getting started with a reasonable function d . After the score is assigned, the distance between two resources is:

$$d(a_k(C_i), a_k(C_j)) \equiv score(a_k(C_i)) - score(a_k(C_j)) \quad (3)$$

Third, the latency distribution from a number of similar configurations is returned to the user. This is fundamentally a nearest-neighbor problem. We do not have any new insights on selecting the right number of neighbors to look at beyond what Mitchell describes [21](chapter 8). AppModel can help visualize relationships between latency and each of the configuration attributes by performing linear least squares fits, for example.

3.4 Interpretation

After a prediction is made, AppModel can provide interpretation for the developer to better understand the data. The interpretation step uses dynamic attributes, which capture how applications use the available resources over time. In addition to visualization over time on many machines, the interpretation step can perform a delta analysis of relevant attributes for best and worst-performing configurations. For example, Table 1 shows two performance signatures for one application (described in Section 4.2), whose performance suffers under configuration C_i . The difference between the two signatures is the total disk utilization. The developer should interpret this delta as a result of contention for disk by another application (i.e., nothing is inherently wrong with C_i , but the user has too many applications open).

Section 5.3 describes our findings of several performance signatures and deltas for common sources of performance variance, ranging from contention with other applications to garbage collection by the common runtime (such as .NET). The goal is to have the deltas be annotated once and the annotation then would be made available to developers experiencing similar problems.

3.5 Summary and limitations of approach

Answering a developer or administrator what-if question, for example, “what happens to performance of application X if I use configuration C ?” first requires the attribute relevance filter to deter-

$Sig_i : < \text{Total disk util} = \mathbf{80\%}, \text{App X disk util} = 20\%, \dots >$
 $Sig_j : < \text{Total disk util} = \mathbf{20\%}, \text{App X disk util} = 20\%, \dots >$

Table 1: Two simplified performance signatures. The difference between them is highlighted in bold.

mine which configuration attributes are relevant. Then, all computer configurations similar to C are examined and the latency distribution is returned as an answer. Enabling this technique to work requires two subtle modeling approaches: partitioning of the attributes into static and dynamic ones and assigning normalization scores to hardware resources by running a set of standardized tests. The interpretation step is designed to help differentiate between expected and unexpected variance in performance and is crucial for deeper understanding of performance. The important takeaway is that annotations made by one developer to get a semantic understanding of the performance discrepancies can be used by others.

All the above modeling and interpretation requires novel application instrumentation. For interactive applications, we have identified the right points of instrumentation as being the entry and exit from a user-induced action (e.g., clicking of a button). Developers instrument application states iteratively. The information learned from AppModel is designed to feed back into a refinement of the iterative instrumentation.

A limitation of this modeling methodology is that its success and fidelity depends on the application being popular and used on many computers with diverse configurations. AppModel cannot answer what-if questions on configuration that have never been seen in the field. The use of the term “model” usually implies capability to calculate performance estimates for any configuration; as such we use this term in this paper loosely. Another current limitation is that we do not explore ways applications might interact² with one another, but only focuses on individual states in individual applications.

It is important to note that delta analysis in the interpretation step is not root-cause analysis. Root-cause analysis might require human understanding at various semantic levels of the system. However, delta analysis can help guide the developer’s attention to resources that exhibit the largest performance discrepancies and is often the first step in a performance debugging session. A possible second step in the debugging process would be to collect detailed system call and kernel traces from the user and send them to the developer whenever the current performance signature matches an annotated one. This is beyond the scope of this paper, but is discussed further in Section 6.

4. MEASUREMENT METHODOLOGY

This section describes the sources of measurements. These measurements are inputs to the models.

4.1 Real data collected

This subsection describes the two sets of real data we collected.

In-the-wild performance data: The first set of data captures information about several popular applications whose user-facing buttons have been instrumented over several months. They include a popular word processing, spreadsheet and presentation software package (Microsoft Office), a media player, an email client (Outlook) and a code editing and debugging software package (Visual Studio). All these applications are in use by thousands of users.

²We differentiate “interaction” (e.g., one application induces a window to open in another) from simple resource contention among applications which we do model.

There are more than 7,000 instrumentation points in these applications. The data collected is available to our developers through a database’s SQL interface.

The data is obtained from users who have opted-in to make it available by agreeing to the privacy statement [20]. Device and OS-specific attributes collected include hardware configuration (memory size, number of CPUs, speed of CPUs, etc.) and software configuration (operating system version, software versions, etc.) Section 5.1 explains data coverage and limitations. A local instrumentation library buffers data every time a user clicks on a button and the performance signature is recorded locally. Periodically the data is transmitted to a centralized database. AppModel makes predictions by using the data stored in the database.

There are several limitations to this data set. First, the data is biased towards users who choose to opt-in. These users might not be representative of the overall user population. Second, the attributes from the OS are not correlated with the attributes collected from an application. For example, an application session might be experiencing high latency because the OS resources (e.g., disk) are heavily used by the kernel or by another application. Because of this limitation, this first data set will not provide interpretations for sources of performance variance. Third, currently not all available OS performance counters are collected. A reason for the above limitations is that this feasibility study is part of ongoing research. Many of these limitations will be addressed with the next few versions of the software, but the timing is dependent on release cycles. Despite the above limitations this data will still be interesting for understanding trends and thus answering what-if questions.

In-the-lab data: The second set of real data is shared by the real developers and testers of Visual Studio. Several hundred scenario-based tests are performed in the lab and latency is reported. The scenarios include start up time, project debugging, project building, etc. Each test is run four times on *one* machine (thus this data is not useful for what-if predictions). Several OS-specific performance counters are collected. This data is helpful for interpreting sources of performance variance.

4.2 Synthetic data for validation

It is difficult to perform sensitivity studies and understand root-causes of performance variance without having direct access to the population’s computers. Hence, we use a synthetic application and a handful of local computers to validate the interpretation of sources of performance variance and explain the results better, albeit with very simple scenarios. The synthetic application has three states after startup: a CPU, disk and a network-intensive states. During the CPU state the application performs mathematical tasks that keep the CPU close to fully utilized. During the disk and network states the application writes out a 1.2GB file in 4KB chunks and then performs random-access 4KB read and write requests within it, with a read:write ratio of 1:2. In the network state, the file is written sequentially to a secondary computer over a 55 Mbps wireless network.

The latency of each state is monitored, together with many OS performance counters exported by Windows. There are two users of this application with machine configurations as shown in Table 2. This application is sometimes run in isolation and other times together with common desktop applications such as browsers, code editing, LaTeX, music player, virus scanner, etc.

5. EVALUATION

This section is formed as a series of hypotheses and evidence in their support. The setup for each case study makes it clear which deployment scenario is used.

Attribute	Machine 1	Machine 2
#CPUs	2	2
CPU architecture	32 bit	32 bit
CPU score	5.1	5.3
Memory score	4.8	5.5
Disk score	5.9	5.5
Graphics score	4.7	4.6
Gaming score	5.3	4.2

Table 2: Relevant machine attributes for synthetic application. The distance metric D between these computers can only be defined as a function of the what-if questions asked, and we will show an example in Section 5.3.

5.1 Measurement coverage and limitations

This subsection provides a visual glimpse into what is collected in-the-wild during a period of 90 days. Below we discuss the coverage and limitations of these measurements.

Hardware configuration: Figure 5(a) shows the number of samples collected as a function of CPU speed and memory size. Other hardware configuration collected (but not shown) includes storage configuration, graphic card configuration, network card configuration etc. Figure 5(d) shows the normalization score assigned to the graphic card as a function of several tests performed on it, as described first in Section 3.3.2. The score is a single number that is representative of the capabilities of that device for common tasks. Currently, running the necessary tests to obtain a score needs to be done manually, and hence not every user reports them. We are working on automating that part.

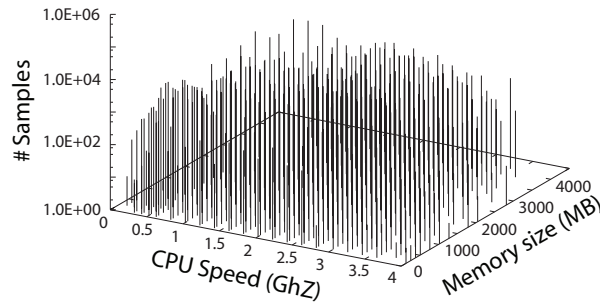
Software configuration: Figure 5(c) shows an example software configuration, the OS version number and application version for a popular code editing software package (Visual Studio). We are currently not making use of software configuration files (e.g., XML files that describe configuration data for an application or registry entries), although collecting this data would be straightforward with our infrastructure. In general, we find that collecting software configuration is more involved for developers and users of our infrastructure than collecting hardware configuration. It requires collecting a list of obvious (e.g., registry and version ID) and non-obvious (e.g., hard-coded parameter in code) configuration entries. Hence, in practice this step is iterative.

Workload characteristics: Capturing hardware and software configurations is only part of the data needed for modeling. The other part involves understanding the inherent demands that a workload places on the system. Figure 5(b) shows two example workload characteristics captured, the file type and bitrate for files played through a popular media player package. Of course, the performance of the media player depends on these characteristics. We find that identifying and collecting the right workload characteristics is an iterative process.

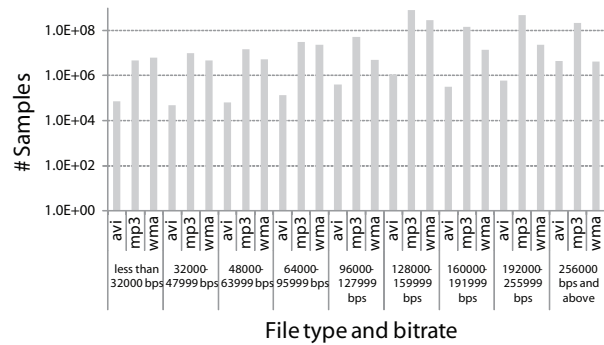
It is worth keeping in mind that the representation of the configuration space is a cross-product of all the above graphs and cannot be visualized in two dimensions. What works to our advantage is that, for the kinds of what-if questions we have looked at, only a few dimensions mattered (usually less than 20).

5.2 Modeling scenarios

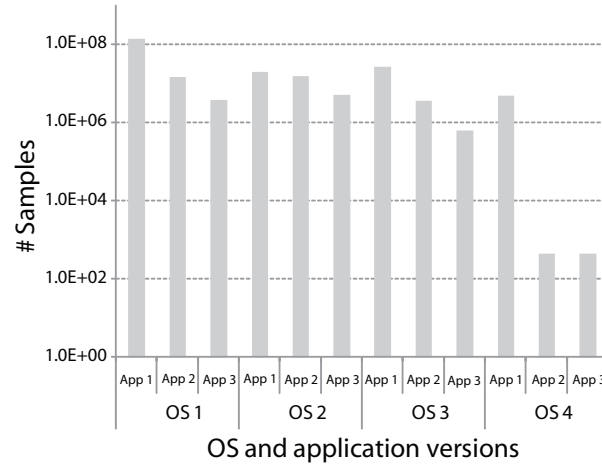
This section uses the in-the-wild performance data to model several aspects of the client applications, for developers and IT administrators. The hypothesis in this subsection is that AppModel can create meaningful trends.



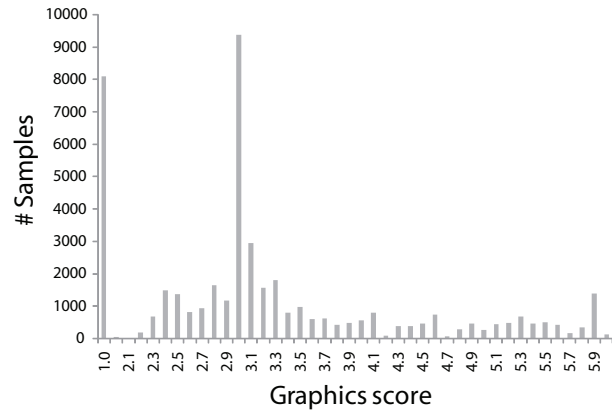
(a) Hardware config: CPU speed and memory size.



(b) Application workload characteristics.



(c) Software config: OS and application version (anonymized).



(d) Baseline performance.

Figure 5: Configuration space for a subset of 4,924,467 user sessions (periods between a user starting an application until the user quits it). A single user might have many sessions, but to preserve privacy no user-identifiable data is kept. Hence, we cannot report on the number of unique users these sessions correspond to.

Modeling startup time: Suppose a developer wants to speed up the startup time (i.e., only one state) of a popular email client (Microsoft Outlook). After tests in the lab, the application is eventually deployed and the developer wants a model of the startup time in the field. The developer asks AppModel to give a distribution of startup time as a function of the top-3 relevant attributes. We show several of the steps involved through this simple example.

First, from the instrumentation step, every time Outlook starts up several attributes are collected (OS metrics indicating resource utilization, application-specific metrics like size of inbox, etc). Second, the attribute relevance filter uses a CART model to rank the top-K attributes relevant for predicting start up time. In our example, the model might indicate that the top-3 attributes are amount of available memory (RAM), CPU frequency and number of emails in the inbox. Third, given that the developer only wants to see a broad distribution of startup time (rather than for a particular computer configuration), there is no need to do the similarity search step.

Figure 6 shows the results. A linear least squares fit is overlaid on the median startup times. We make several observations. First, the number of emails in the mailbox is likely to make the biggest difference in application startup time. Second, memory size and CPU speed appear to both be correlated with startup time, though the correlation is less strong than that with number of emails. The developer runs into a standard correlation vs. causality problem here, since AppModel also reports that memory size and CPU speed

are correlated with one-another (intuitively a machine with a lot of memory most likely has faster CPUs as well). It is up to the developer to interpret causality. Most likely the developer would consider memory size to be more relevant (intuitively startup time can be reduced if parts of the application code and data reside in memory). Third, the variance in performance can be large. We will analyze potential sources of variance in Section 5.3.

At this point the developer could work on reducing the impact of number of emails on startup time. One option would be to reduce the number of emails in the inbox by enabling aggressive auto-archival features. Another would be to introduce a more efficient email index data structure. Whatever the developer chooses to do, the main takeaway is that AppModel helped with the understanding of the tradeoffs experienced in the field.

Note that an IT administrator might ask a similar questions to AppModel when making an upgrade decision (e.g., to buy more memory for client machines). The steps involved would be similar, but the administrators’ computer configuration would have to be examined in the similarity search step (i.e., the administrator is not interested in all computers out there, just the ones s/he maintains).

Modeling a multiple-state application: The next scenario is similar to the previous one, but we will tell the story from the point of view of a hypothetical IT administrator. The administrator could ask the question “What happens if I upgrade to the latest release of an OS?” In particular, the administrator might be interested in

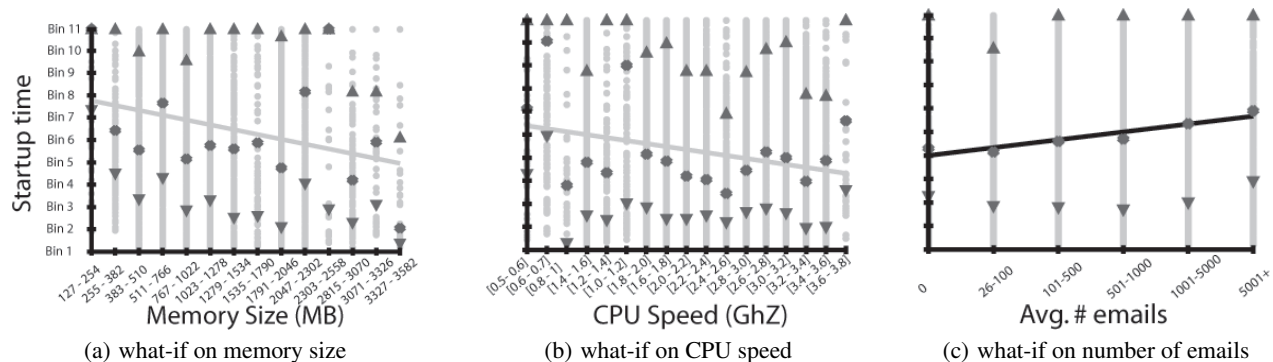


Figure 6: Results when modeling an email client’s startup time as function of memory size, processor speed and number of emails in the folder. Note that, for the purposes of this paper, the startup time has been masked into bins. The bins match across the three graphs. The 25%, median, and 75% quartile markers are shown. This experiment uses data from 105,821 user sessions.

an application most of the employees use. Say that application is Microsoft Excel. The administrator wants to minimize client complaints, thus the performance of the application must not degrade after the upgrade. We model a rather simplistic scenario with three states to make the steps clear: starting the application, opening a file, appending more data to it and then saving it.

The instrumentation step is identical to the previous scenario and the startup state, file open button and file save button have all been instrumented. Figure 7 shows one of the three correlations for the three application states from 11,220,340 user sessions, for the save state only (the other two states have similar trends). We make several observations. First, the correlation between OS version and save time is weak. This is also reflected in the attribute filter tree itself (not shown), which does not give a high weight to OS version. The task of the IT administrator ends here. The upgrade can go ahead.

Second, the range of latencies might be large, and the developer might have to get involved to understand why. Although the numbers are masked, we can confirm that the 75% quartile latencies are well in the “normal behavior” range where the user does not perceive any difference. The next two bins have values in the “noticeably annoying” range. These annotations are not arbitrary, and the values are chosen according to studies of human perception while interacting with computers. For the troubleshooter of performance bugs, the “noticeably annoying” range includes performance problems that require fixing. Many of these could be correctness errors that could benefit from correlation with crash reports [12] and detailed tracing. We discuss this in Section 6.

Other real usage: In addition to the above rather simplistic cases, many developers and testers in most of our applications are actively using the measurements to improve performance. Broadly speaking, there are three kinds of problems being addressed. The first revolves around understanding how people interact with applications and then improving usability and testing. For example, by observing how users interacted with Microsoft Office (the performance signatures capture sequences of button clicks), decisions were made to change much of the button layout to allow frequent operations to be done with fewer clicks. This resulted in improved usability (as measured by user satisfaction surveys). Also, more testing effort is placed in lab on application interactions that are frequent in the field. The second usage revolves around understanding performance for emerging platforms, for example 64-bit architectures, multiple cores and solid-state drives. A third usage model is to help in performance debugging by building models of expected

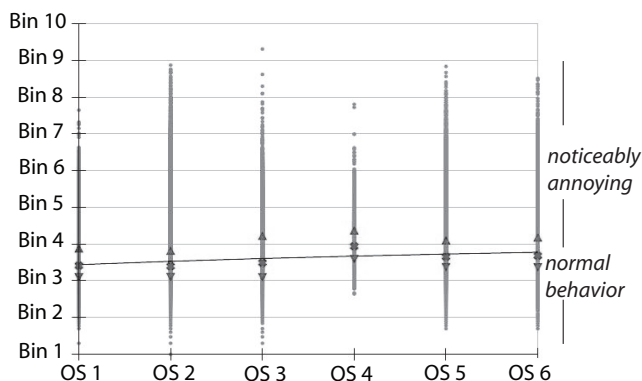


Figure 7: Spreadsheet file save time as a function of the OS version.

and unexpected variance. All these usage models have justified non-negligible costs incurred by the infrastructure, as mentioned in Section 5.4.

5.3 Interpretation and validation

As Figures 6 and 7 show, many times performance can vary significantly. A developer is often interested in normal (or expected) variance vs. unexpected performance variance. The hypothesis of this sub-section is that AppModel can help the developer interpret sources of variance. In this section we will also describe some details on the subtler approaches of using dynamic and static attributes as first mentioned in Section 3.3.2.

Because of the data collection limitations mentioned in Section 4.1 we can only use the code editing and debugging program (in-the-lab test). We have data from several hundred tests of it running in isolation, i.e., there is no resource sharing by other foreground applications. These tests are part of a real nightly regression suite. Figure 8 shows the average, min and max latencies from running each test four times. We observe that even in this controlled environment, performance might vary by two orders of magnitude. We do not attempt to provide an exhaustive list of performance signatures for expected performance variance. However, we list several common ones we found from this experience. Note the discussion in Section 3.5 on root cause analysis limitations. AppModel provides performance counters values that are worth investigating, but a human must interpret their values.

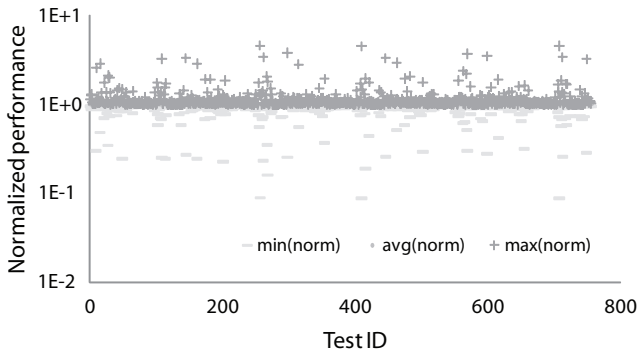


Figure 8: Inherent variance in application performance. The y-axis is in log scale. + indicates the maximum latency seen, - indicates the minimum latency seen, whereas . indicates the average latency. All numbers are normalized to the average.

Variance associated with memory and storage sub-system: The two main sources of variance from the above tests came from the CPU and disk resources. Because disks are mechanical devices, the positioning latency of the disk head can take from 0 to several milliseconds. The exact latency depends on the previous position of the disk head. In general, disk access times are expected to vary by tens of milliseconds and our infrastructure considers this to be normal behavior. In addition, depending on the contents of the storage buffer cache, sometimes requests hit in the cache, sometimes they miss. The other layers in the memory hierarchy (e.g., L2 and L1 CPU caches) influence performance as well. The interpretation step performs a delta analysis among cases of good and bad performance and results in a visualization as first shown in Section 3.4.

This next set of experiments are performed using the data from the synthetic application described in Section 4.2. We ran the synthetic application continuously for a week on Machine 1 and Machine 2 to understand other sources of performance variance.

Variance associated with OS background activities: We observed variance in latency or throughput may result from background activities. This may be the OS performing maintenance, such as checkpointing, lazy write-backs, or defragmentation. In these cases, we observed that performance counters show resources being used by the kernel. There are also user-level maintenance activities, such as garbage collection done by the runtime environment or a virus scanner performing its duties. AppModel’s interpretation step gets a delta of the counters for the runtime environment (that describe the rate of garbage collection, among other things). The virus scanner is just another process so its process counters will show high activity — disk or CPU — when the scanner is active.

A closer look at variance associated with resource sharing: Here we analyze deeper sources of variance when an application contends for resources with other foreground applications. We run three scenarios on two different computers (the computers’ configurations can be found in Table 2). We only show one graph, a case with a mix of accurate and inaccurate predictions; the other graphs have accurate predictions. First, we run the application on each computer with the disk-intensive state only. AppModel builds an attribute filter using the data from one of the computers and predicts the relevant attributes of the second computer’s run. Second, we run one instance of the application with the disk-intensive state only. We then start a CPU-intensive application on

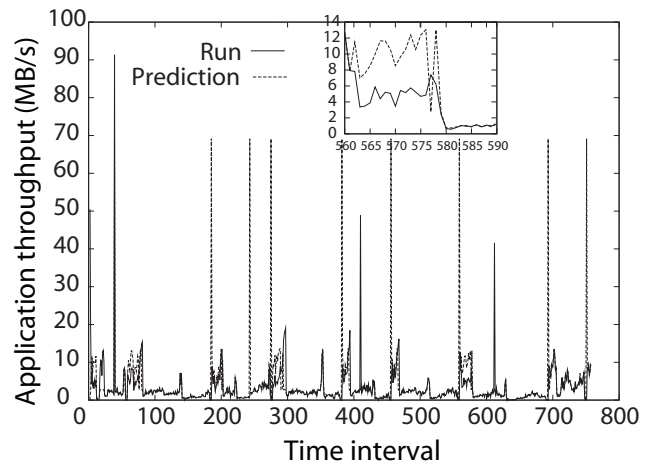


Figure 9: Cases of accurate and inaccurate predictions.

the same computer to understand the performance signature of resource sharing among unrelated resources. We observe that the attribute filter successfully correlates application throughput with the disk attributes and in the second case does not get perturbed by the CPU-bound application. Intuitively, this is expected, since the application is disk-bound and CPU sharing should not significantly affect it. For this what-if question, the distance metric depends only on the disk score and equals $D((C_i), (C_j)) \equiv diskscore((C_i)) - diskscore((C_j)) = 0.4$. The distance is small, hence it is meaningful to compare the performance between the two computers.

Third, we repeat the second experiment, but this time the two instances both run the disk-intensive state (thus heavily contending for the disk). AppModel builds an attribute filter using data from the single instance run, and predicts the relevant attributes for the run with contention. We make several observations: First, the attribute filter successfully correlates application throughput with the static disk performance counters in both runs. Second, Figure 9 shows that there are cases when the attribute filter does not correctly predict the second run’s performance, i.e., the dynamic attributes change between the two runs. This is OK. As mentioned in Section 3.3.2, the dynamic attributes might be different across runs. What is important for the similarity search is that, for both runs, the correct (disk-related) *static* attributes are chosen. Then, the interpretation phase exposes the contention (to the human) through delta analysis of the *dynamic* attributes. This analysis shows that the process “disk utilization” attribute is smaller in the second run, while “total disk utilization” remains the same. This indicates contention of the disk by another process.

In summary, performance variance depends on resource contention. The signature for a resource under contention contains a high total resource utilization and only a fraction of it is used by the application under consideration. AppModel does not do deep characterization of contention; when performance is bad it merely answers the binary question “was there contention of resources or not?” Periods of contention are not flagged as anomalous.

Variance associated with external dependencies: From a test that exercised the network bound state of the synthetic application (writing on a network share, instead of the local disk) we observed that performance on a single computer might depend on external resources on different computers (e.g., email stored on a server). In these cases, AppModel gets a delta of the rate of bytes being sent on the network interface and reports the network resource as needing most attention.

Experiment	Runtime	Std. Dev.
w/o counters	61.6 s	2.58
+ counters (137)	63.5 s	2.24
+ counters (18)	60.7 s	2.27

Table 3: Overhead for producing and collecting performance signatures. The average and std. dev. over 40 runs is reported.

5.4 Cost and efficiency

How hard is it to build and maintain the infrastructure?:

The infrastructure used to instrument applications, collect the data and make it queryable has taken time to build and is still ongoing in that new application states are being continuously instrumented. As costs change, we are only able to provide rough numbers for the costs involved. The hardware costs (servers to collect and store the measurements with standard load balancing front-ends and database back-ends) currently range from several thousand to 100,000+ US dollars, depending on the application (each application maintains its own set of servers). Tens of people maintain the servers and provide training to developers for adding new instrumentation points and interpreting them. Most of our applications have been instrumented over a period of more than 3 years.

Does the monitoring degrade performance?: For users that have opted-in to provide the data (in-the-wild), the instrumentation is on at all times with no negative perceived performance reported from the field. Nonetheless, we provide some theoretical overheads here for completion, using the synthetic application. We set the frequency of state changes to be once every 3 seconds (in practice users click less frequently for real applications). We ran the application described in Section 4.2 in four different setups. The resulting overheads are given in Table 3. In the first experiment, we turned off all monitoring and event generation. For the second run, we collected a set of 132 OS and 5 application metrics. This case resulted in a 3% runtime overhead. The third run is described in the next sub-section. When collecting all 137 metrics, the raw data consumed less than 20MB per day.

It is conceivable that the monitoring overhead might increase with future releases. When that happens, we could benefit from using existing optimization mechanisms to further reduce monitoring overhead, such as the ones presented by Bhatia et al. [4], Verbowski et al. [28] (e.g., on collecting metric deltas) and Kiciman et al. [16] (e.g., collect different sets of performance metrics from different users).

Details on attribute filter efficiency: Not all attributes collected are used for every what-if question. For example, we created an attribute filter used to predict the relevant attributes for the synthetic application’s disk-intensive state. This filter was created in under 2 minutes using 40,000 data points in training mode (on the first computer) and can subsequently make 4,000 attribute filter predictions/second. The filter is small in size, less than 110kB. Figure 10 shows its accuracy as a function of attribute filter tree depth. We observe that there is no gain in accuracy for depths larger than 8, a point where the tree only contains 18 attributes. When restricting data collection to these attributes in a third run, we measured a negligible monitoring overhead as shown in the last row of Table 3.

All the algorithms we described in this paper are embarrassingly parallel. For example, different applications have different unrelated models. Attribute filters from different users and data collection and processing can all be done in parallel.

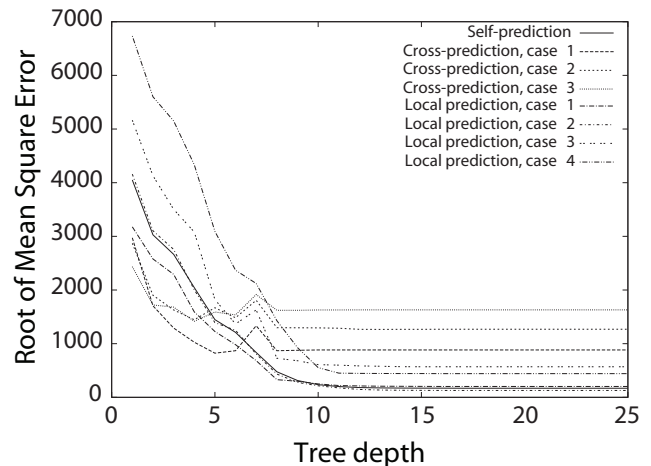


Figure 10: Attribute filter prediction error. *Self-prediction* means the model was used to predict its own training data. *Cross-predictions* are predictions made for 3 runs on the second computer. *Local predictions* are predictions for 4 runs on the same computer but on different test data. RMS values have little absolute meaning, but are used here to indicate when prediction stability is reached and the point of diminished returns.

6. ONGOING WORK AND DISCUSSION

More detailed tracing: There are several avenues for future work. First, performance reports could be correlated with crash reports, e.g., like the ones collected by Windows [12]. This would help understand the relationship between program correctness and good performance. Second, when latency is entering the “noticeably annoying” category as shown in Figure 7, detailed system call and kernel traces could be transmitted to the developer. The mechanism to collect these traces is part of Windows [19].

What other applications could be modeled?: We hypothesize on how other popular applications could be modeled, by discussing a few cases. All these hypotheses need verifying. A graphics suite such as the Adobe Creative Suite has many similarities to the interactive applications we used. Each action performed on an object (such as color changes, rasterizing, blurring) can be instrumented, together with characteristics of the workload (e.g., file size and resolution determines blurring time). It might also be possible to model popular games, such as Quake and Halo, as a sequence of states. Certainly, rendering time could be modeled as a function of the hardware configuration. Response time could be modeled as a function of the scene’s complexity. Web 2.0 applications (the client side) have similarities to the interactive applications we used, in that several of the (often JavaScript) functions run on thousands of computers. The instrumentation mechanism might look like AjaxScope [16], but the modeling might be similar to ours.

Should end-users bother?: When we embarked on this project, exposing the performance models to end-users was high on our list. We had two hypotheses. First, we thought that end-users would want a tool that helped them with upgrade questions. Second, we thought that end-users might benefit from a “Red button” which they could press each time an application is perceived to be slow, an idea first explored by Basu et al. [3]. We have not done a formal user study, but from interactions with users we realized both ideas were implausible. End-users will usually buy the best device they can afford and they do not want to tinker with its hardware (e.g., buying faster disks or adding CPUs). From simple experi-

ments with a “Red button” we implemented we also realized that asking users to make extra clicks does not add to a positive usage experience, because they do not get an immediate reward for it — fixing a performance problem might take time.

7. SUMMARY

AppModel is a step towards modeling popular, complex client applications. This novel modeling framework relies on global aggregation of performance signatures collected from real application deployments. It does not require deep instrumentation of the operating system or application and it does not need *a priori* knowledge on how the application should behave. AppModel is designed to be versatile. It could help developers expand their testing coverage, to improve program usability by providing them with information on how an application is behaving in the field, and to annotate previously-seen performance anomalies. AppModel has the potential to help IT administrators, too, in understanding consequences of an upgrade decision.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Murray Woodside for helping us improve this paper. We thank Paul Barham, Tim Harris, Rebecca Isaacs, Joe Hellerstein, Joseph Joy, Dongmei Zhang for valuable discussions on this project. This work would not have been possible without the efforts of the Microsoft Customer Experience Improvement team and Microsoft Office, Windows, Visual Studio and Media Player teams.

9. REFERENCES

- [1] <http://research.microsoft.com/~etheres/research/Sigmetrics2010Survey.xls>.
- [2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating System Principles*, pages 74–89, 2003.
- [3] S. Basu, J. Dunagan, and G. Smith. Why did my PC suddenly slow down? In *SYSMML'07: Proceedings of the 2nd USENIX workshop on tackling computer systems problems with machine learning techniques*, pages 1–6, 2007.
- [4] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Symposium on Operating Systems Design and Implementation*, pages 103–116, 2008.
- [5] P. Bodík, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Eurosys*, Paris, France, 2010.
- [6] C. Borgelt. DTX - decision tree induction and execution. <http://www.borgelt.net/dtree.html>.
- [7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and regression trees*. Chapman and Hall/CRC, 1998.
- [8] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Symposium on Operating Systems Design and Implementation*, pages 309–322, 2004.
- [9] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Symposium on Operating Systems Design and Implementation*, pages 231–244, 2004.
- [10] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing clustering and retrieving system history. In *ACM Symposium on Operating Systems Principles*, 2005.
- [11] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *ACM Symposium on Operating Systems Principles*, pages 221–236, 2007.
- [12] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the Symposium on Operating systems principles*, pages 103–116, 2009.
- [13] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. *SIGPLAN Not.*, 42(6):101–111, 2007.
- [14] Q. He, C. Dovrolis, and M. Ammar. On the predictability of large transfer TCP throughput. In *ACM SIGCOMM Conference*, pages 145–156, 2005.
- [15] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 1991.
- [16] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *ACM Symposium on Operating Systems Principles*, pages 17–30, 2007.
- [17] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queuing network models*. Prentice Hall, 1984.
- [18] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–48, 2007.
- [19] Microsoft. Event tracing. <http://msdn.microsoft.com/>.
- [20] Microsoft. Microsoft customer experience improvement program. <http://www.microsoft.com/products/ceip>.
- [21] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [22] S. E. Perl and W. E. Weihl. Performance assertion checking. In *ACM Symposium on Operating System Principles*, pages 134–145, 1993.
- [23] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Symposium on Networked Systems Design and Implementation*, pages 115–128, 2006.
- [24] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The case for application-specific benchmarking. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, 1999.
- [25] K. Shen, M. Zhong, and C. Li. I/O system performance debugging using model-driven anomaly characterization. In *Conference on File and Storage Technologies*, pages 309–322, 2005.
- [26] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Symposium on Networked Systems Design and Implementation*, 2005.
- [27] E. Thereska and G. R. Ganger. IRONModel: robust performance models in the wild. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2008.
- [28] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: monitoring persistent-state interactions to improve systems management. In *Symposium on Operating Systems Design and Implementation*, pages 117–130, 2006.
- [29] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Symposium on Operating Systems Design and Implementation*, 2004.
- [30] Wikipedia. Windows system assessment tool. http://en.wikipedia.org/wiki/Windows_System_Assessment_Tool.