Observer: keeping system models from becoming obsolete

Eno Thereska, Anastassia Ailamaki, Gregory R. Ganger
Carnegie Mellon University
Pittsburgh, PA, USA

Dushyanth Narayanan
Microsoft Research
Cambridge, UK

Abstract

To be effective for automation, in practice, system models used for performance prediction and behavior checking must be robust. They must be able to cope with component upgrades, misconfigurations, and workload-system interactions that were not anticipated. This paper promotes making models self-evolving, such that they continuously evaluate their accuracy and adjust their predictions accordingly. Such self-evaluation also enables confidence values to be provided with predictions, including identification of situations where no trustworthy prediction can be produced. With a combination of expectation-based and observation-based techniques, we believe that such self-evolving models can be achieved and used as a robust foundation for tuning, problem diagnosis, capacity planning, and administration tasks.

1. Introduction

To design and maintain complex systems, it is critical to have a way to reason about them without resorting to full implementation of alternatives. Hence, behavioral models are often built to represent software and hardware components. Such models are consulted for making acquisition, planning and performance tuning decisions as well as for verifying that the system behaves as expected when deployed. For example, a distributed storage system model that checks for compliance to specifications could verify that, when 3-way replication is used, three storage-nodes are contacted on a write. For performance tuning, a storage-node model could predict whether it is best to buy more memory or get faster disks in the cluster.

Whereas it is reasonable to expect system designers to construct good behavioral models for how the system should behave, one cannot expect that the models will account for all workload-system interactions. Systems and workloads evolve, causing them to diverge from the models. Sometimes systems are misconfigured to start with. The models that are built assuming an idealized, predefined workload-system configurations thus become obsolete. When the models do not match reality, it is currently up to the system designer/programmer/administrator (i.e., human being) to discover the root cause of the mismatch and fix the models or system. In distributed systems with hundreds of resources and tens of competing workloads, this task is enormous and time consuming and raises system management costs.

To be useful in practice, system models need to be robust. Robust models discover regions of operation where the prediction confidence is high and regions for which they choose not to predict. They handle deviations from the physical system by localizing the likely cause of a mismatch and continuously refining themselves to account for unforeseen (and hence not programmed-in) workload-system interactions. Thus, in the long run, the models themselves embody the necessary knowledge and rules of thumb that currently humans attempt to learn.

Building self-evolving models will require detailed collection of system statistics (far more detailed than currently done) and efficient algorithms for evolving the models. Hence, a significant amount of spare resources (CPU, network, storage) will need to be used for model evolution. Given that system management costs vastly eclipse hardware costs [7], this is the right time to throw extra hardware towards self-evolving models that are ultimately used to better manage complex systems.

2. Context

System models allow one to reason about the behavior of the system while abstracting away details. Models take as input a vector of workload and system characteristics and output the expected behavior (e.g., performance) of the modeled component. System modeling approaches fall into two categories: *expectation*-based and *observation*-based. Neither is adequate for robust modeling, but each has a role to play.

2.1. Expectation-based models

Expectation-based models use designer and programmer knowledge of how systems behave; thus, the system is considered as "white-box". The models have a built-in, hardwired definition of "normalcy" (e.g., see [8, 9]). Indeed, highly accurate models have been built for disk arrays, network installations, file system caches, etc.

Designers can model both structural and performance properties of a system and workload. For example, a structural expectation in a distributed storage system is that, when RAID-5 encoding is used, five storage-nodes should be contacted on a read. A CPU model might indicate that storage decryption of the read data should use 0.02 ms for 16 KB blocks, and it should take 1.5 ms to send the data from the five storage servers to the client over a 100 Mbps network. A cache model could predict whether the requests will miss in cache and a disk model could predict their service time, and so on.

Expectation-based models are the right starting point for future system designers (some formality in system design is a must). However, the issue remains how to evolve these models over time with minimal burden to humans. We observed that performance models in systems that we traditionally considered to be white-box (since we designed and built them from scratch) exhibited black-box behavior that resulted from 1) unforeseen workload characteristics or system configuration characteristics, 2) unforeseen interaction of white-box components [6] or 3) administrator misconfiguration of the system.

2.2. Observation-based models

Observation-based models do not make *a priori* assumptions on the behavior of the system. Instead, they infer "normalcy" by observing the workload-system interaction space. As such, these models usually rely on statistical techniques (e.g., see [2, 4, 13]). These models are often used when components of the system are "black-box" (i.e., no knowledge is assumed about their internals). For example, Cohen et al. [4] describe a performance model that links sudden performance degradations to the values of performance counters collected throughout the black-box system. If correlations are found, for example, between a drop in performance and a suspiciously high CPU utilization at a server, the administrator investigates the root cause by first starting to look at the CPU utilization on that server. In another example, Wang et al. [13] built a storage server model based on observing historical behavior and correlating workload characteristics, such as inter-arrival time, request locality, etc., with storage system performance.

Observation-based models are the remedy option when pre-existing models are not available. However, they require a large set of training data (i.e., previous observations), an issue which can be a show-stopper even for simple modeling tasks. In particular, observation-based models predict poorly the effects of workload interference in shared systems. Consider a data center, for example. The performance of any workload is strongly correlated with the load placed on the system's resources (e.g., CPU, network, cache, disk) by other interfering workloads. With the "load" attribute taking values from 0-100% for each of the resources, the observation-based model would need to have seen hundreds of *distinct* load configurations to make a reasonable performance prediction (which can be made in a straightforward manner when employing expectation-based models that use queuing analysis).

A simple analogy that illustrates the difference between the two kinds of models comes from the chess world: one can make the next move by knowing nothing about chess rules (i.e., black-box) and only considering an annotated database of board setup images. The other option is to make the move by applying chess rules (i.e., white-box). Systems and workloads change over time (whereas the chess rules remain the same), hence purely black-box observations in systems risk being obsolete every 1-2 years.

2.3. A fusion of models

We have come to believe that a robust modeling architecture will need to augment expectation-based approaches with observation-based approaches, as shown in Figure 1. Known expectations should be continuously observed and verified. Over time, high-confidence suggestions from the observation-based models should be incorporated into the expectation-based models. We are less concerned with what exact tools and algorithms will be used for the models — indeed there are plenty to choose from (e.g., decision trees and Bayes Nets [5] or reinforcement learning [10] from the machine learning community) — and more concerned with how systems should be built so that these tools can be applied in a meaningful way.

3. Components of a solution

We believe that any solution that addresses the problem of model evolution must address two issues: *mismatch localization* and *re-learning*. The former is responsible for localizing mismatches between the model and actual system. The latter is responsible for evolving the model.

3.1. Mismatch localization

A robust, self-evolving model must detect when there is a deviation between the model and current workload-system interactions. Thus, in addition to answering hypothetical

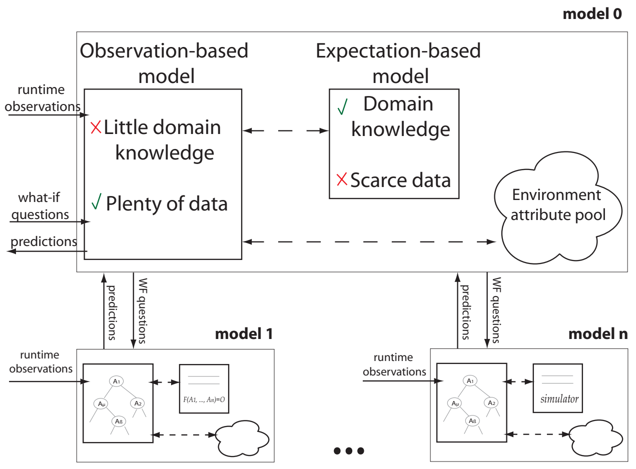

Figure 1: Modeling architecture in a multi-tier system. Each model self-checks and builds confidence for its predictions. New correlations discovered by observation-based models are eventually incorporated into expectation-based models.

what-if questions, individual models should continuously self-check.

From a system's perspective, there are challenges. First, the self-checking must be managed in terms of when it should happen and how frequently. Second, care must be taken to build efficient models that do not consume unreasonable amounts of resources to self-check. Because management is quickly becoming the dominant cost in systems, it may be justified to throw money at dedicated hardware for modeling, but the costs need to be examined.

3.2. Re-learning and model evolution

After a successful mismatch localization, a re-learning component should be responsible for automatically evolving the expectation-based models (or at least making educated suggestions to the model designers). Any algorithm used for evolving must address two issues. First, it should discover new attributes of the workload-system interaction space that should be incorporated into the model. Second, this component should discover regions of operation where the prediction confidence is high and regions where it is not. Hence, any eventual model outputs should have a notion of confidence associated with them. Below, we discuss a high-level approach that could address both issues.

3.2.1 Attributes, and more attributes

An observation we make is that, in many cases, system designers know the workload and system *attributes* that are correlated with the workload's eventual behavior, but not the exact nature of the correlation. Indeed, most model designers build expectation-based models using initial attributes they "feel sure" about (because, perhaps they have a theoretically proven correlation with reality).

For example, disk arrays come with a specification of their expected sequential- and random-access throughput. This is the information on which many disk array models are initially built. However, there are other workload characteristics (e.g., request locality, burstiness, stripe width, etc.) that make a large difference in the eventual workload performance. Storage system designers know that these attributes will have an impact on the prediction, but do not always know 1) which of the attributes are most important, 2) their effect when combined, and 3) their effect on a particular disk array type.

Hence, we believe that expectation-based models should have observation-based parts to them. The observation-based parts should incorporate learning algorithms that continuously sample the workload-system space for new attributes and discover whether these attributes are strongly correlated to the output. Over time, new highly-correlated attributes should be incorporated into the expectation-based model. Observation-based models can also help with deriving confidence values for each prediction. This should be done as part of keeping historical data that over time reflect the model's prediction accuracy. Confidence values can then be used to make policy decisions.

Of course, the issue of having a rich attribute pool to select from is challenging. However, we believe we are close to being able to collect much more detailed system statistics than 10 years ago (mostly due to hardware resources that are now cheap enough to be dedicated to statistics management). For example, in our cluster-based storage system [1] we have started to collect *environmental data* (temperature, humidity, etc), *hardware demands* (per-request, per-machine), *error messages*, *request flow traces* [12], hardware and software *configuration data* (components, topology, users), and annotations of *human activity*.

3.2.2 Active probing and the big, red button

Two approaches can be used to accelerate the process of discovering strongly correlated attributes: active probing and some human involvement. Without acceleration, important correlations may be missed due to infrequent observations (e.g., one in 100 clients uses small stripe units, and their effect might not have been modeled).

First, once an attribute is observed to have some correlation with the model's output, active probing (generating synthetic workloads to test that hypothesis) should be used. The challenges here involve how to have the system itself construct meaningful probes, what kind of physical infrastructure is needed to run the probes onto, and when should these probes run (i.e., a sensitivity study on how much correlation is good enough to justify a probe).

Second, there needs to be a way to involve a human in directing and shaping the algorithms' focus. There are plenty of "false alarm" events that may trigger the system to behave strangely for a while (e.g., power outages, backups, machine re-configuration). In those cases, the human should advise the algorithm to ignore what it learned. The challenge is to have the system designed with a "big, red button" in mind that the administrator can press when such false alarm events happen.

3.3. Early experiences

This section discusses how self-evolving models might help solve some real modeling challenges. We experienced these challenges in Ursa Minor, our cluster-based storage system [1], which currently uses static expectation-based models. The models in this system are used for performance prediction by keeping track of the per-client bottleneck resource (CPU, network, cache, disks). The first example illustrates how individual models would evolve. The second illustrates how a collection of models that work fine individually, might struggle when composed. The third illustrates how self-checking models could be used as a layer to build on for successful problem diagnosis.

Unexpected CPU bottleneck: The CPU model in our system predicts the CPU demand needed to encode/decode and encrypt a block of data when a particular data encoding scheme is used (e.g., replication or erasure coding). A certain workload was getting less than half of its predicted throughput. A manual inspection of the resources consumed revealed a CPU bottleneck on the client machine. The model was significantly under-predicting the amount of CPU consumed and thus did not flag the CPU as a potential bottleneck. It was later discovered that this was because the workload used small block sizes (512 B) and the kernel network stack consumed significant amounts of CPU per-block. Hence, it was impossible to keep the network pipeline full, since the CPU would bottleneck. Our CPU model was built using commonly-used block sizes of 8-16 KB for which the per-block cost is amortized by the per-byte cost. We did not foresee the different behavior from small block sizes.

Using robust models would ideally require no manual diagnosis. All resource models (CPU, network, cache, disks) would self-check and the CPU one would be found the culprit (e.g., it predicted each block needed 1 ms of CPU time; in reality it was taking 2-3 ms). An observation-based model might notice that the attribute "block size" was significantly smaller than in the test cases and would start generating test cases with small block sizes. These probes could run at night on the same physical infrastructure. Eventually the "block size" attribute would be incorporated into the CPU model.

Figure 2 illustrates re-learning results for this example obtained from an early implementation of observation-based models in Ursa Minor. We borrowed a machine learning algorithm, called CART [3] and combined it with our expectation-based models. The initial CART model does not need any training data (the CPU demand as a function of block size can be calculated using expectation-based models, as first described in [11]). Over time, it incorporates a new attribute, "block-size" in its structure. The leaves of the CART tree maintain a notion of confidence in the predictions, which is strongly related to the number of sample data seen in the field. CART picked the attribute "block size" from request flow traces, which are collected online [12].

When striping goes wrong: The network model in our system predicts the network time to read and write a block of data when a particular data encoding scheme is used. A particular workload had larger-than-expected response times when data was read from multiple storage-nodes at once (e.g., when striping data over more than 5 servers). All other workloads that shared servers with the first workload had normal response times. A manual diagnosis of the problem took unreasonably long. Different tests were run, on different machine types, kernels and switches. Using this semi-blind search, the problem was eventually localized at a switch. The switch's buffers were overflowing and packets were getting dropped. That started TCP retransmissions on the storage nodes. The problem is known as TCP-influx, and is rather unique to storage systems that read data from multiple sources synchronously (i.e., all storage-nodes were sending data to the client at the same time).

Ideally, any manual diagnosis would be side-stepped by having the models self-check as the workload is running. For example, the cache model might predict that the workload would get a hit rate of 10% with 256 MB, and indeed that is what the workload would be getting. However, the network model might reveal that remote-procedure-calls (RPCs) are taking 20 ms, when they should only be taking 0.2 ms. Sub-models of the network model, the NIC and switch model, also self-check and the switch model might report structural mismatches (same packet sent multiple times). The high-level data flow model would then incorporate the retransmission probability related to the attribute "number of stripes".

Understanding upgrade effects: We made the decision to upgrade each cluster server from the Linux 2.4 to the 2.6 kernel about a year ago. However, we still have not made the move. Several of our nightly tests performed differently on 2.6 (some performed better, but a third of the tests performed much worse). Ideally, each of the behavioral models would self-check to locate the mismatches with the previous kernel version.

It could be the case that several of the models report a

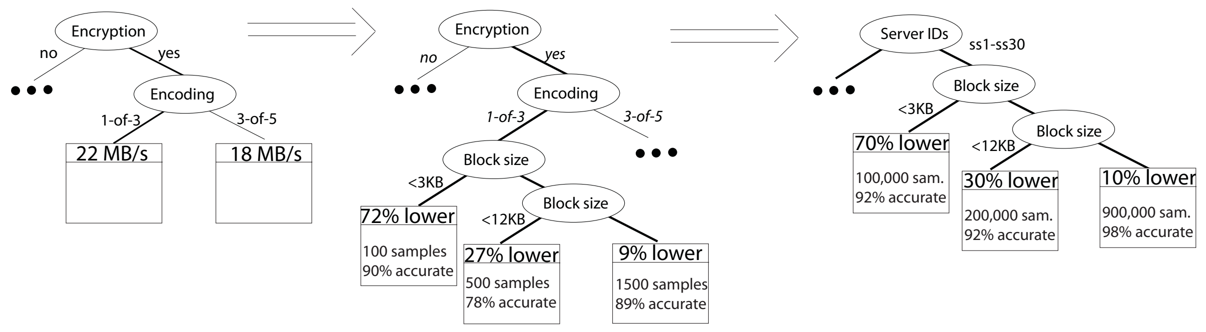

Figure 2: Evolving an encode/decode CPU model in Ursa Minor. The initial CART model is constructed using expectation-based models. It evolves in the field over time. First, it incorporates a new attribute "block-size" in the models (the second tree). Second, it generalizes over a rack of machines with similar CPUs (the third tree). The second and third trees predict how much worse performance is when compared to what the first tree predicts.

discrepancy. For example, a change in the TCP stack processing affects both network transmission times and CPU consumption. Both affect eventual throughput and response time. However, we believe orthogonal diagnostic methods could be built on top of a robust modeling layer that self-checks. Such diagnostic methods could, for example, perform a large run of tests that are slightly different from one another. It could make use of the modeling layer to see how each test interacts with parts of the system.

4. Conclusions

This position paper argues that humans should be relieved of the task of maintaining system models. After a good-enough initial implementation, the models should evolve in the field, by incorporating new relevant workload-system attributes. A robust model design will require maintenance of fine-grained, pervasive system statistics, and may benefit from accelerated observation-based learning techniques. In the long run, the models themselves will thus embody the necessary knowledge and rules of thumb that currently humans attempt to learn.

5. Acknowledgements

We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Oracle, Panasas, Seagate, and Symantec). This research was sponsored in part by NSF grants #CNS-0326453, #CCF-0621508 and #CCF-0621499, by DoE award DE-FC02-06ER25767, and by ARO agreement DAAD19–02–1–0389.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: versatile cluster-based storage. In *Conference on File and Storage Technologies*, pages 59–72, 2005.

[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] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and regression trees*. Chapman and Hall/CRC, 1998.

[4] 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.

[5] T. M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[6] J. C. Mogul. Emergent (mis)behavior vs. complex software systems. In *EuroSys*, pages 293–304, 2006.

[7] F. Moore. *Storage New Horizons*. Horison Information Strategies, 2005.

[8] S. E. Perl and W. E. Weihl. Performance assertion checking. In *ACM Symposium on Operating System Principles*, pages 134–145, 5–8 December 1993.

[9] 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.

[10] G. Tesauro, R. Das, N. Jong, and M. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *International Conference on Autonomic Computing*, pages 65–73, 2006.

[11] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. In *International Conference on Autonomic Computing*, 2006.

[12] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 3–14, 2006.

[13] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *MASCOTS*, pages 588–595, 2004.