IRONModel: Robust Performance Models in the Wild

Eno Thereska
Microsoft Research

Gregory R. Ganger
Carnegie Mellon University

ABSTRACT

Traditional performance models are too brittle to be relied on for continuous capacity planning and performance debugging in many computer systems. Simply put, a brittle model is often inaccurate and incorrect. We find two types of reasons why a model's prediction might diverge from the reality: (1) the underlying system might be misconfigured or buggy or (2) the model's assumptions might be incorrect. The extra effort of manually finding and fixing the source of these discrepancies, continuously, in both the system and model, is one reason why many system designers and administrators avoid using mathematical models altogether. Instead, they opt for simple, but often inaccurate, "rules-of-thumb".

This paper describes IRONModel, a robust performance modeling architecture. Through studying performance anomalies encountered in an experimental cluster-based storage system, we analyze why and how models and actual system implementations get out-of-sync. Lessons learned from that study are incorporated into IRONModel. IRONModel leverages the redundancy of high-level system specifications described through models and low-level system implementation to localize many types of system-model inconsistencies. IRONModel can guide designers to the potential source of the discrepancy, and, if appropriate, can semi-automatically evolve the models to handle unanticipated inputs.

Categories and Subject Descriptors

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

General Terms

Design,Management,Performance

Keywords

Management, what-if, behavioral modeling, active probing

1. INTRODUCTION

System designers and administrators shy away from using mathematical models to tune their systems. A glance at several large code bases from operating systems (Windows and Linux), database systems (MySql, Microsoft's SQL Server, Postgres), and web servers (Apache and Microsoft's IIS) reveals a distressing lack of any built-in models for continuous capacity planning and tuning. These systems are used by millions of users and manage several important resources (e.g., hardware resources such as CPU, network, buffer caches and disks, and software resources such as locks). Theoretically, even simple queuing laws from 30 years ago [9] could provide first-order answers to important what-if questions (e.g., the Windows operating system should arguably answer the question "What would happen to user response time for the Quake game if I double the CPU speed?", since it should know best how the Quake game utilizes its resources). None of these systems can answer such questions today, however, and tuning them is still a black-art best left to experienced administrators.

Over the last 3 years, we have studied the sources behind the lack of enthusiasm for incorporating and using mathematical models in systems by tracking their usefulness in an experimental cluster-based storage system. This paper addresses one such source of concern: model brittleness. We found that models, especially those based on queuing analysis, can be brittle for three reasons:

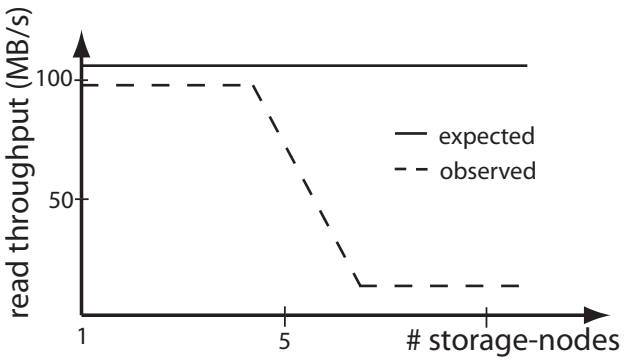
(1) Models are not first-class citizens: Models are usually built by model designers, not system designers. Models are built about a system, not within the system. Each time the system changes, the models might become obsolete. The model designers would then need to consult the system designers as to what might have changed. The added communication overhead, coupled with tight project timelines often means that fixing the model (or system) has low-priority.

(2) Systems can be misconfigured or buggy: System components might have been implemented wrong or might be later misconfigured in the field. Traditional idealistic models are not prepared to deal with this harsh, but common, reality.

(3) Models can be limited or buggy: Building models is not a flawless process either. We observe that models often have regions of system-workload interactions in which they work well and regions in which they do not. Reasons that such regions exist include non-linear behavior that is mathematically difficult to model and incomplete understanding by the model creator on how the system behaves under certain complex conditions (e.g., performance of a distributed storage system when small buffers in network switches overflow).

This paper's main contribution is IRONModel, an instance of a robust performance modeling framework. IRONModel is built on several principles. First, the system designer builds and incorporates the models within the system. We find that approaches in which the system's behavior is second-guessed by tools outside the system lead to inaccurate models and added model management

SIGMETRICS'08, June 2–6, 2008, Annapolis, Maryland, USA.



1: Observed versus expected average throughput when striping data.

overhead. Second, the models continuously validate the system's structural behavior (i.e., how requests flow through it) and performance behavior (i.e., how long each request takes to be processed by each component). The redundancy of high-level system specifications described through the models and low-level implementations can be used to identify the presence and help localize the source of a performance anomaly. Helping this localization process in a distributed system is crucial. Third, queuing-based mathematical models are coupled with statistical components that give fidelity estimates by keeping track of historical information about their predictions. System-workload regions of operation where the models frequently mispredict will have low fidelity. In many cases, these statistical models can continuously adjust queuing models' parameters for unexpected workload-system attributes.

Figure 1 shows a particularly tough anomaly observed in our storage system for which both the system and initial models exhibited incorrect behavior. This graph shows a single client's read throughput as a function of number of storage-nodes data is striped over. The observed behavior differs from the expected behavior given by queuing analysis, especially as data is striped over a large number of storage nodes.[1] When IRONModel was introduced into the system, it localized the problem to the network resource (by self-checking and eliminating resources that were working as the models expected). The problem was eventually localized to a network switch type with insufficient buffer space. That switch type services over 100 machines in the cluster. Fixing the problem would have required buying new switches with bigger buffers, which due to budget constraints was not possible. Hence, the network models, to be useful for later predictions, had to evolve to account for the inadequacies of the existing switches. This example is indicative of a series of "harsh reality checks" one is faced with when using mathematical models in a real system.

Out of 29 system-model discrepancies encountered during a 3 year period in our experimental storage system, IRONModel correctly localizes the source of the problem in 23 cases, identifying 18 system bugs and 5 model bugs. For all the 5 model bugs, IRONModel was able to retrain the models correctly with minimal human interaction.

2. BACKGROUND AND RELATED WORK

System models allow one to reason about the behavior of a system while abstracting away implementation details. Models can be useful for answering what-if questions. They take as input a vector of workload and system characteristics/attributes and output the expected behavior (in this paper, performance metrics such as throughput and response times) of the modeled component.

2.1 Expectation-based queuing models

We are interested in building robust performance models for systems whose internals we know (either we have the source code or we are building them from scratch). We will call such models *expectation-based models*.[2] This paper does not address building models for "black-box" systems, for which there exist no expectations and hence all observations are statistical. Traditional expectation-based models have had a hardwired definition of "normalcy" (e.g., see [12, 22, 24, 25, 27, 33]). Indeed, highly accurate models have been built for disk arrays, network installations, cache behavior, and CPU behavior. Our experience, however, has been that hardwiring normalcy leads to obsolete models. For example, the network model for the environment shown in Figure 1 was obsolete for the switches with insufficient buffer space.

Designers can model both structural and performance properties of a system and workload. Figure 2 illustrates one simple expectation in a hypothetical system consisting of a database and a storage system. A structural expectation is that, when 3-way replication is used, three storage-nodes should be contacted on a write, and acks should be subsequently received. A performance expectation is that three times the original block size should be seen on the client's network card. Information on CPU and network demands can be automatically discovered (e.g., CPU data encoding/encrypting should use 0.02 ms and it should take 0.5 ms to send the data to the storage-nodes). A myriad of methods are available for creating performance expectations. For example, a CPU model might be based on direct measurements (i.e., model emulates the real CPU processing of data and reports on the time that takes). A network model might be an analytical formula that relates the request size and network speed to the time it takes to transmit the request. Cache and disk models might be based on simulation and might replay previously collected traces with a hypothetical cache size and disk type. Each of these models shares the property that the algorithms of the underlying modeled resource are known (e.g., the cache model uses the same replacement algorithm as the real cache manager).

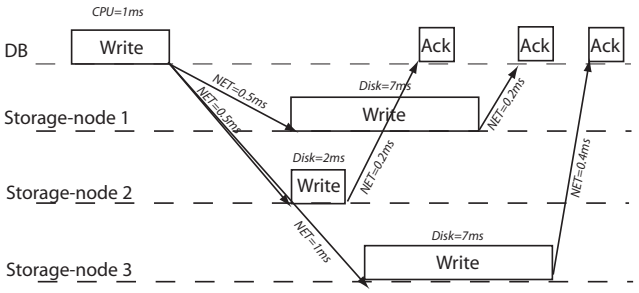
In a distributed system with multiple resource tiers, one needs a general framework to reason about the effects of a hypothetical change in any of the tiers on end-to-end performance. *Queuing analysis*, which models the system as a network of queues, is the building block of such a framework. Each queue model represents a resource. Customers' requests place demands on these resources. There is much related work on queuing analysis, dating back 30+ years (e.g., see [9, 13, 15]). This paper does not invent any new queuing laws. The contribution here is to enable them to be robust. Fundamentally, queuing theory assumes knowledge of the way requests flow through the system's service centers (i.e., knowledge of the *queuing network*) and knowledge of the performance of the individual resources (i.e., *service centers* along the queuing network). For a sufficiently complex system, our experience indicates that this knowledge needs to be constantly updated.

2.2 Related work

Several success stories in *initial* system capacity planning (e.g., see [4] for a storage system capacity planning tool, and [17] for an email server capacity planning tool) have proved that queuing models are useful. However, there is little work done on *continuous* capacity planning tuning, and few studies check the long-term

[1] This problem known as the TCP-incast problem [23], and we'll revisit it in Section 6.

[2] In this context, *expectation* means intended behavior, not statistical mean or average.



2: Structural and performance expectations.

behavior of models (and systems) in harsh, but common, system deployments. Evidence from position papers indicates that unexpected behavior is common [11, 19]. Theoretical work encourages combining expectations with observations (e.g., see Chapter 12 of [18], and [29]).

2.2.1 Similar approaches to similar problems

PSpec [22] and PIP [24] are most related to our work. PSpec is a language that allows system designers to write assertions about the performance behavior of their system. Once the assertions are written, they are continuously checked against the system. PIP augments PSpec by allowing designers to write assertions about the structural behavior of the system as well. IRONModel builds on these approaches and generalizes them by trusting neither the model nor the system implementation as correct. IRONModel uses a hybrid modeling scheme, where expectation-based models are augmented with statistical observation-based models to provide long-term fidelity in the model and system. IRONModel currently relies on system- and programmer-specific conventions for writing expectations and using a language like PIP's is left as future work.

2.2.2 Different approaches to similar problems

Much work has explored the use of purely statistical approaches to inferring system normalcy. Statistical models do not make *a priori* assumptions on system behavior. Instead, they infer normalcy by observing the workload-system interaction space. These models are often used when components of the system are "black-box", i.e., the model designer has no knowledge about their internals (e.g., see [2, 8, 16]). For example, Cohen et al. [8] describe a performance model that links sudden performance degradations to the values of performance counters collected throughout a 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 usage on that server.

Statistical models are the remaining option when expectations are not available. To test their pros and cons, we actually tried replacing expectation-based models with statistical ones for several resource types (e.g., CPU and network). We make the following observations: 1) Statistical models do not have a notion of correct system behavior. In many cases, (e.g., Figure 1's network throughput degradation case) a statistical model "learns" the bad behavior, rather than flagging it as incorrect. An expectation-based model can easily flag such anomalies. 2) Statistical models are not efficient in multi-workload environments. In particular, they require much training data to predict workload interference effects. 3) Fundamentally, given enough time and training data, statistical models do approach the power of expectation-based models. Hence, the decision on whether to use them depends primarily on the applications' tolerance to low-fidelity predictions until enough training data has been observed.

IRONModel side-steps the need to choose between the two modeling types and uses both to create a robust modeling framework.

2.2.3 Similar approaches to different problems

Detecting and understanding performance bugs is part of the larger subject of finding bugs in systems. We mention only two examples here. Engler et al.'s work treats correctness bugs as deviant behavior [10]. The abstraction level differs from ours: low-level programming code in Engler et al.'s work, and higher-level request flows in a distributed system in our work. Our work also has parallels in recent work on using programmer comments to check for implementation bugs in source code [28]. The authors use the redundancy between programmer comments and source code to detect inconsistencies between the two. Such inconsistencies might indicate either bad comments or buggy code. IRONModel uses the redundancy between performance models and implementations to detect inconsistencies.

3. IRONModel'S DESIGN

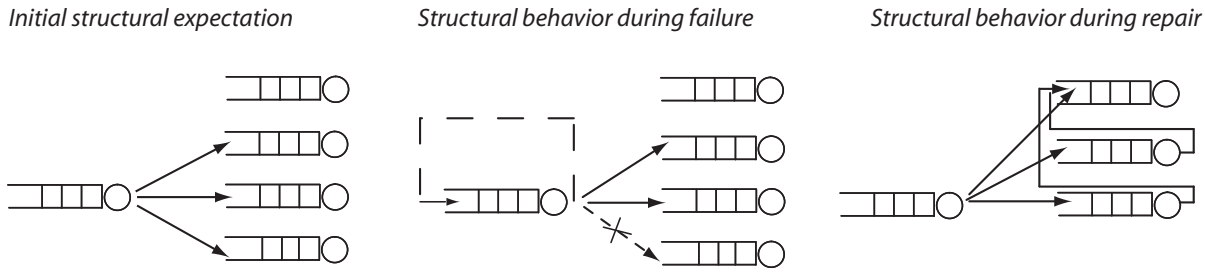
3.1 Overview of approach

IRONModel uses three approaches to make models robust. First, models are part of the system or can otherwise obtain information on all requests the system sees and their performance. Thus, IRONModel makes use of the redundancy between a model's high-level specifications and system implementation to compare and contrast system and model behavior. This enables the second and third approaches to work. Second, IRONModel uses *mismatch localization* to localize the source of a performance anomaly to a few resources in a distributed system. In a system with 100+ resources, this approach is quicker than having a human manually check each resource using trial-and-error approaches. Third, IRONModel uses *model refinement* techniques to make educated guesses as to what the source of the inconsistency might be. Furthermore, the models, over time, provide a notion of fidelity with predictions; predictions with low fidelity could be disfavored by upper decision-making layers (which might include human administrators).

In a queuing network, IRONModel must detect two kinds of mismatches. The first, *structural mismatches*, refers to requests taking unanticipated paths through the system. For example, for the same example shown in Figure 2, a transient network partitioning might cause storage-node 1 to be disconnected from the rest of the system. A request directed to that storage-node might then enter a retry loop before finally being redirected to another storage-node. IRONModel uses end-to-end tracing techniques to monitor service centers' entry and exit points and automatically creates a new structural profile for that case.

Performance mismatches, on the other hand, refer to requests taking shorter or longer than models predict (for the buffer cache resource, mismatches mean that a request hit in cache, when it should have missed, and vice-versa). For example, a misconfigured network switch between the database client and storage nodes in Figure 2 could lead to significant degradation of network throughput (a similar degradation is shown in Figure 1). In that case, each model in the system would self-check (e.g., the CPU, buffer cache and disk models might report that the expected and observed behavior match), and the network model might report a mismatch.

To refine its models, IRONModel uses *observation-based* statistical models. Such models have at their core simple machine learning algorithms. First, they keep track of historical data to create a notion of fidelity with each prediction. Second, they rank the possi-



3: Structural deviations during failure and repair for 3-way replication. The initial structural expectation for write requests indicates that a write should contact three storage-nodes. If one of the storage-node fails, one of the writes times out and reenters the sending service center. During repair, a spare storage-node takes over from the failed one and is populated by the two remaining storage-nodes. Three graphs can be created and contrasted for each of these scenarios (nodes in such graphs would be entry and exit points from the service centers above and edges would represent the service center's response times).

ble causes of a system-model inconsistency. These models collect and analyze, at each service center, several relevant *attributes* to find statistical correlations with observed inconsistencies. For example, for the network performance degradation in Figure 1, the attributes most correlated with bad performance are the number of storage-nodes and the switch type (e.g., switches from vendor A might have a different "badness" profile than from a vendor B). IRONModel's statistical models have a key advantage over off-the-shelf machine learning tools. In IRONModel, the statistical models derive their initial structure from the expectation-based models and hence require no training data to start making predictions.

3.2 Mismatch localization

Structural mismatches happen when designer expectations do not anticipate ways requests might flow through service centers.

We learned that, to detect structural mismatches, a system must have a good measurement infrastructure in place that keeps track of requests at service centers' entry and exit points. Methods for designing such *end-to-end tracking* of requests have been an active area of research recently [5, 7, 31]. These methods were primarily developed for accurately measuring latencies in a system. Our framework, called Stardust, was also initially developed for accurate performance measurements [31], and we have recently found it to be useful for anomaly localization as well. Here, we briefly sketch its general design relevant for localizing structural mismatches. Section 4 discusses implementation details of one particular instance of this framework in a distributed storage system.

Stardust tracks every client request along its execution path. It collects activity records, network RPC records, buffer cache reference records, disk I/O records, etc. The sequence of records allows tracking of a request as it moves in the system, from one computer, through the network, to another computer, and back. An activity record is a sequence of (attribute, value) pairs. Each activity record contains an automatically-generated header comprised of, among other fields, a timestamp and a request ID field. Each timestamp is a unique value generated by the computer clock that permits cycle-accurate timing measurements of requests. The request ID permits records associated with a given request to be correlated within and across computers. Activity records are posted on the critical path; however, as Section 5 shows, such posting causes minimal impact on foreground performance.

Stardust creates causal graphs, similar to the one shown in Figure 3, for every request. Those graphs can then be contrasted with the graph of structural expectation that the designer first inputs. Structural mismatches manifest themselves as a change in the fan-in/out of service center nodes. Performance mismatches manifest themselves as a change in the edge latency between service center nodes (a reasonable policy might define "change" as a one standard deviation from the expected average edge latency). To localize mismatches, a model must be built to continuously self-check its behavior and the behavior of the system.

3.3 Model refinement and fidelity

After a successful mismatch localization, a refinement component makes educated suggestions to the model designers on the reason behind the mismatch. Over time, models incorporate new correlations between performance and system-workload attributes.

3.3.1 A zero-training machine learning model

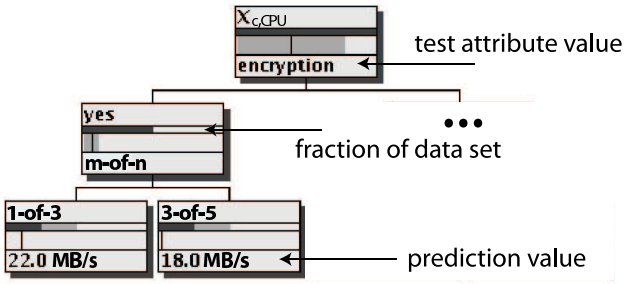
To make educated suggestions on the reason behind a mismatch, models need to correlate system-workload attributes with performance metrics. Attributes with a high correlation with performance are reported to the model designer as potential culprits. Formally, we want a correlator that approximates the function $\mathcal{F}(\mathcal{A}_1, ..., \mathcal{A}_n) = \mathcal{P}$, where $\mathcal{A}_i$ is an element of the workload-system attribute space and $\mathcal{P}$ is the performance metric of interest. Attributes can be any relevant observations about the operating environment. For example, for a disk resource, traditional attributes of interest include *disk type*, *request inter-arrival time*, *read:write ratio*, etc. $\mathcal{P}$ can be any metric of interest we want to validate, such as, throughput X. We wanted a correlator to satisfy several requirements:

Handling of mixed-type attributes and combinative associations: Workload-system attributes can take on categorical, discrete or continuous values. For example, average *inter-arrival time* is an attribute that takes a continuous value, but *switch type* has categorical attributes (Vendor *A*, ..., Vendor *Z*). Furthermore, a system's behavior might depend on combinations of attributes. For example, the expected throughput from a disk array might depend on both workload burstiness and request locality.

Reasonably fast and adaptive: The learner must be able to make predictions reasonably fast, on day 1, and ideally retrain itself quickly. In many scenarios, the training and predictions can be made offline, perhaps at night when the system utilization is low. In addition, making the prediction must be inexpensive in terms of computational and storage requirements. The learner must efficiently adapt to new workloads and learn incrementally.

Cost-sensitive and interpretable: The learner should be able to reduce the overall cost of mispredictions by taking application-specific cost functions into consideration during training. For example, a learner could be accurate 99% of the time, but the cost of the 1% misprediction could outweigh the benefit of being correct 99% of the time. Newly learned rules or correlations should ideally be human-readable. System designers and administrators we talk to place a lot of emphasis on needing to build their trust in the system and validating simple correlations with their intuition.

The base algorithm we chose, CART, is borrowed from the ma-



4: Initial Z-CART model. The initial model is built using expectation-based models (domain expertise) and requires no training data. To make a prediction (at the leaves), Z-CART recursively selects an attribute of the system-workload space to split on.

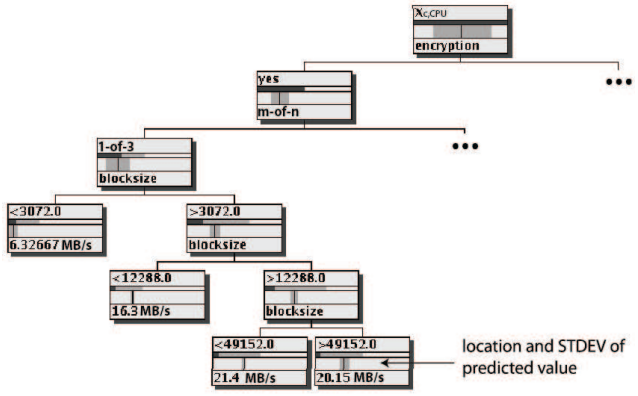
chine learning community [18] and it fulfills most of the above requirements. Training time is $O(nHeight(n))$, and predictions take $O(Height(n))$ time, where n is the number of observations and $Height(n)$ is the height of the decision tree (which, for most trees we have constructed, is between 10 and 100).

Traditional CART models, however, (much like other machine learning models, like Bayes networks or neural networks) suffer from requiring much training data before even simple classifications can happen. Because IRONModel already incorporates expectation-based models, training data is not required to make predictions. We call Z-CART (zero-training CART) the fusion of CART with expectation-based models. Z-CART derives its initial structure from the expectation-based models (we think of the latter as *domain knowledge*). Over time, Z-CART validates the expectation-based models are finds new, unforeseen correlations.

Figure 4 shows an example starting point for a Z-CART model. The goal of the model is to predict the maximum throughput for a client c, $X_{c,CPU}$, from the CPU resource. For this example, assume that the CPU resource is only used to encrypt blocks and then encode them using an m-of-n erasure coding scheme (e.g., 1-of-3 means 3-way replication, whereas 3-of-5 is an erasure coding scheme that writes data to 5 storage nodes and needs to read from 3 of them to reconstruct the original data; both schemes tolerate two storage-node crashes). The CPU model might use direct measurements to make this prediction. It might encrypt and encode a block of data and report on the time (or demand, $D_{c,CPU}$) it took. $X_{c,CPU}$ would then equal $1/D_{c,CPU}$.

Imagine a situation in which, after good predictions for weeks, a certain workload starts getting less than half of its predicted throughput (this happened in our system). After successful mismatch localization, the CPU model's initial prediction of $D_{c,CPU}$ is found to be different from the actual $D_{c,CPU}$. Stardust has meanwhile collected trace records containing workload attributes (e.g., block size, read:write ratio, file name, etc.) for all requests passing through that service center. Z-CART uses the original CART algorithm for selecting the attribute most correlated to the unexpected drop in performance. CART computes a metric called the *information gain* on each available attribute, and then greedily chooses the attribute with the largest information gain. Intuitively, the higher the information gain, the higher the correlation of an attribute and classification metric. Analytically, the information gain from choosing attribute A_i with value a from a set of classification instances S with (observed) probability distribution $P(s)$ is:

$$Gain(S,A_i) = Entropy(S) - \sum_{a \in A_i} \frac{|S_a|}{|S|} Entropy(S_a) \qquad (1)$$



5: Modified Z-CART model. After several new observations, the initial Z-CART model evolves to incorporate the new block size attribute. The predicted value (vertical line) together with one standard deviation (shaded area around vertical line) is shown by a GUI tool [6]. The location of the predicted value is relative to the minimum and maximum values observed (in this case 3.6 MB/s and 57 MB/s respectively).

$$Entropy(S) \equiv - \sum_{\text{all classes } s} P(s) log_2 P(s) \qquad (2)$$

Figure 5 shows the modified Z-CART model after the block size attribute has been incorporated into it (the reason *why* the system behaves differently as a function of block size is deferred to later sections that describe the system in more detail). The leaves of the Z-CART model contain fidelity information. Fidelity is defined as the number of *pure* samples seen in the field (i.e., how many of the samples does it classify correctly?). If $\mathcal{P}$ is categorical (e.g., a yes/no answer to "is $X_{c,CPU}$ less than 50 MB/s?"), the leaf maintains counters that keep track of observations for each category. If $\mathcal{P}$ is discrete or continuous (e.g., an answer to "what is $X_{c,CPU}$?"), the leaf maintains a histogram of observations.

3.3.2 Accelerating learning through active probing

In many cases, IRONModel might help accelerate the learning process of finding new correlations. *Active probing* is the term we use to describe the synthetic creation of workload-system interactions for the purpose of building fidelity for correlations. IRONModel expects system designers to construct template test cases (usually a few lines of code) to probe the system. IRONModel can guide the probes based on observed statistical correlations with performance. For example, if the attribute "read size" is found to have an unexpected (hence not modeled) strong correlation with performance, IRONModel reports the correlation to the system designer who writes a small template probe with "read size" as one of the attributes. IRONModel can then run the probe on the system and refine the Z-CART model. From our experience, we have identified three types of probes necessary to explore the workload-system interaction space.

Probing of primary attributes that have an immediate effect on performance: This probing involves changing primary workload and system attributes, thus effectively generating new synthetic requests and resources to test with. For example, in a storage system with read requests of the form (file ID, offset, size), synthetic requests can be created by assigning the attribute "size" values in the range 1-512 KB. As another example, to see the workload behavior when a small cache size is used, the system could temporarily assign a smaller cache partition to the workload.

Probing of workload interference: This probing involves study-

ing unanticipated effects of workload interference. We have found that modeling the effect of workload interference is more difficult than modeling single-workload scenarios, and the effects of workload interference are almost never gotten right upfront.

Probing of long-term system behavior: Systems behave differently during their lifetime, even when no new resources have been added. In storage systems, in particular, we found that performance usually degrades over time for one primary reason: performance degradation of key data structures (e.g., hash table performance as more and more files are stored in the system) and storage-nodes as the disks get fuller (due to several reasons, e.g., use of lower-density disk tracks towards the middle of the disk).

3.4 Summary and discussion

IRONModel uses the redundancy between models and system to contrast their behavior and localize mismatches. IRONModel constantly validates a given model and attempts to refine it by discovering new correlations between system-workload attributes and performance metrics. Ultimately, every what-if question is answered by the Z-CART models. The first time the question is posed, Z-CART consults the expectation-based model. The answer from the expectation-based model is used to construct the initial Z-CART model and is also returned as the answer to the what-if question. Each Z-CART model continously refines its predictions and fidelity in the field.

The question remains on how much have we reduced human intervention and what do humans still need to do? System designers need to provide support for end-to-end tracing and need to provide reasonable expectation-based models to start with. IRONModel automatically checks traces collected during run time for structural and performance discrepancies. For each discrepancy, IRONModel ranks the most likely attributes associated with it using the entropy test and presents the results to the system designer or administrator, together with suggestions for new test templates for active probing. The designer ultimately has to find the root cause of the problem after IRONModel has reduced the search space.

4. IRONModel IN A STORAGE SYSTEM

This section serves illustrates how expectation-based models can be efficiently accommodated as first-class citizens in a real system. It also provides context for the evaluation and case studies.

We have implemented a prototype of IRONModel in an experimental cluster-based storage system called Ursa Minor [1]. Ursa Minor is being developed for deployment in a real data center. On the software side, Ursa Minor consists of about 250,000 lines of code. On the mechanical front, clients of the storage system first find where their data resides by querying a metadata service (or MDS). The metadata service is also responsible for authorizing client access to storage-nodes through the use of capabilities. Then, clients proceed to access the data from storage-nodes in parallel through an erasure coding protocol family. The storage-nodes have CPUs, buffer cache and disks. Storage-nodes are heterogeneous, as they get upgraded or retired over time and sometimes are purchased from different vendors.

Models in Ursa Minor are needed to predict performance consequences of data migration and resource upgrades [30]. For example, a workload migration what-if question of the form "What is the expected throughput/response client c can get If its workload is moved to a set of storage-nodes S?" needs answers from several models. First, the buffer cache hit rate of the new workload and the existing workloads on those storage-nodes need to be evaluated by a buffer cache model. Second, the disk demand $D_{c,DISK}$[3] for each of the I/O workloads' requests that miss in buffer cache will need to be predicted by a disk model. Third, the network demand $D_{c,NET}$ on each of the storage-nodes that results from adding/subtracting workloads needs to be predicted by a network model.

Each individual model makes the above predictions. A second modeling tier uses standard bottleneck analysis (e.g., see Chapter 5 of Lazowska et al. [15]) to derive throughput and response time bounds. For example, to make throughput predictions for closed-loop workloads, let client c have N_c requests outstanding, and let its average think time be Z_c. Then client c's throughput X_c is:

$$X_c \leq min\left(\frac{1}{D_c^{max}}, \frac{N_c}{D_c + Z_c}\right) \quad (3)$$

where D_c^{max} is the largest demand client c places on any resource and D_c is the sum of all demands on all resources a request uses. There is a similar formula for response time that follows from Little's law. For open-loop workloads the models only predict peak throughput as $1/D_c^{max}$.

The implementation of the models has been incremental. Initially, Ursa Minor was designed with expectation-based models built-in [30]. We studied how well those initial models worked and why they failed. Then, we implemented IRONModel, an instance of a robust modeling framework.

4.1 Adding expectation-based models

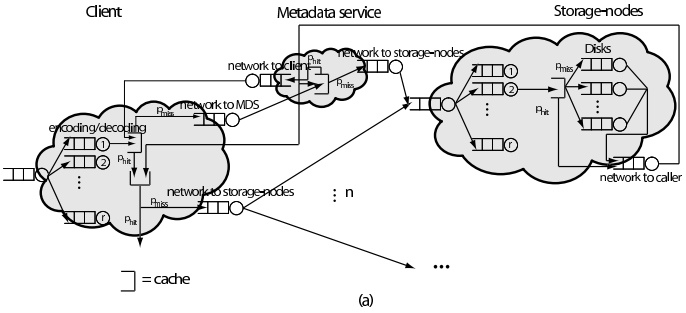
Figure 6(a) shows a simplified queuing network (only for write requests) inside Ursa Minor. We explicitly defined the structural behavior of the system during the system design phase. This was not an easy task. However, the energy spent on it is a fraction of the energy already spent verifying the correctness of the various algorithms and protocols in the system. Concretely, several months were spent by several people designing and reviewing how requests would flow in Ursa Minor. Translating that work into a queuing network took one person less than a month. Below, we briefly sketch how each individual model works to make clear what could go wrong.

4.1.1 Model for CPU-bound service centers

Models for CPU-bound service centers answer questions of the form "What is the CPU demand $D_{c,CPU}$ for requests from client c, If the data is encoded using scheme E?". The CPU model uses direct measurements of encode/decode and encrypt/decrypt costs to answer these questions. Direct measurements of the CPU cost are acceptable, since each encode/decode operation is short in duration. Direct measurements eliminate the need to construct analytical models for different CPU architectures. Each time the above question is asked, the CPU model encodes and decodes one block several times with the new hypothetical encoding and produces the average CPU demand for reads and writes.

As first hinted in Section 3.3.1 and further elaborated in Section 6, we eventually discovered that we were not modeling the CPU consumption correctly. A secondary CPU-bound service center, which we initially did not consider, models the CPU consumed by TCP network stack processing inside the kernel. The CPU consumed by the network stack is a function of the request size, and the model uses direct measurements in this case too. A tool called Iperf [26] is used to actively probe the CPU consumption from the network stack as a function of request size.

[3]All models can operate with quantiles of performance metrics, e.g., 99^{th} quantile of demand. However, for simplicity, this section means *average* demand when it refers to demand.



Resource	Some relevant system and workload attributes
CPU	**Request type**, **encoding (m, n, encryption style)**, **read:write ratio**, block size, kernel version, TCP offload(yes/no), CPU type
Network	**Encoding (m, n, encryption style)**, **block size**, **read:write ratio**, endpoints, NIC type, switch type
Buffer cache	**sequence of accesses**, adaptive workload(yes/no)
Disk	**sequential (yes/no)**, **request size**, disk type, arrival rate, spatial locality

(b)

6: Simplified queuing network in Ursa Minor. The queuing diagram(a) shows the path for writes, from a single client. The clouds represent major software components in Ursa Minor. In the attributes Table (b), in bold are attributes we *felt sure* had a direct relationship with performance. These are the attributes of the expectation-based models. IRONModel validates these relationships and discovers new ones over time.

4.1.2 Network model

The goal of the network model is to answer questions of the form "What is the network demand $D_{c,NET}$ for requests from client c, If the data is encoded using scheme E?". Inputs to the network model are the hypothetical encoding E and the measured read:write ratio of the workload. In Ursa Minor, a write updates n storage-nodes, and a read retrieves data from only m storage-nodes. The fragment's size (a fragment is a fraction of data sent to each storage-node) is the original request block size divided by m:

$$\text{Bytes sent} = p_{read} BlockSize + p_{write} BlockSize \frac{n}{m} \quad (4)$$

$$Time = \frac{\text{Bytes sent}}{Bandwidth} + \text{network setup time} \quad (5)$$

Modeling modern LAN network protocols can be much more complicated, in practice, since there are factors outside of Ursa Minor's control to be considered. For example, we do not have direct control over the switches and routers in the system. TCP can behave in complex ways depending on timeouts and drop rates within the network [12]. However, the above analytical model proved to be a good starting point.

4.1.3 Buffer pool models

The goal of a general buffer cache model is to answer questions of the form "What is the probability $p_{c,HIT}$ that requests from client c are absorbed[4] in cache, If the cache size is X MB?" The buffer cache model uses simulation to make a prediction. In Ursa Minor, predictions cannot be accurately done using analytical formulae due to the complexity of the caching algorithms. The model uses buffer cache records of each of the $W_1, W_2, ..., W_w$ workloads, collected through Stardust, and replays them using the buffer cache size and policies of the target storage-node. In Ursa Minor, the cache size is partitioned and dedicated to each client (the cache partitioning algorithms are beyond the scope of this paper, and they are described in [34]). Hence, the analysis above can be done independently for every workload W_c, without considering how other workloads might interfere with it.

4.1.4 Disk model

The goal of the disk model is to answer questions of the form "What is the average service time $D_{c,DISK}$ of a request from client c, If that request accesses a particular disk?". The average service time for a request is dependent on the access patterns of the workload and the policies of the underlying storage-node. Storage-nodes in Ursa Minor are optimized for writes, utilize battery-backed RAM, use a log-structured layout on disk, and prefetch aggressively. Request scheduling is round-robin across each workload, where the length of the scheduling quanta for each round is determined analytically as described by [34].

When a disk is installed, a model is built for it. The model is based on the disk's average random read RND_{read} and write RND_{write} service times and average sequential read SEQ_{read} and write SEQ_{write} service times. These four parameters are usually provided by disk vendors and are also easy to extract empirically. The disk model is a combination of simulation-based and analytical methods. It receives the sequence of I/Os from each of the workloads (from the buffer cache what-if model), scans the combined trace to find sequential and random streams within it, and assigns an expected service time to each request. Within a quanta, each request gets a service time as follows. Let RUN be an ordered set of requests that are consecutively sequential (there are read and write runs), and let $RUN[i]$ be the i^{th} request in the run (c^i simply states that the i^{th} request is from client c). Then, within a quanta Q_c:

$$D'_{c^i,DISK} = \begin{cases} SEQ_{read} & \text{if } c^i \in \text{read RUN} \wedge c^i \neq \text{RUN[1]} \\ SEQ_{write} & \text{if } c^i \in \text{write RUN} \wedge c^i \neq \text{RUN[1]} \\ RND_{read} & \text{if } c^i = \text{read RUN[1]} \vee c^i \notin \text{RUN} \\ RND_{write} & \text{if } c^i = \text{write RUN[1]} \vee c^i \notin \text{RUN} \end{cases} \quad (6)$$

Thus, when considering the quanta length $|Q_c|$ for each client c, assuming request from all clients arrive uniformly distributed during the quanta times, the expected request demand and additional think time for each client request is:

$$D_{c^i,DISK} = \left(\frac{|Q_c|}{\sum_j |Q_j|} \right) D'_{c^i,DISK} \quad (7)$$

$$Z_{c^i,DISK} = \left(\frac{\sum_{j \neq c} |Q_j|}{\sum_j |Q_j|} \right) \left(\frac{\sum_{j \neq c} |Q_j|}{2} \right) \quad (8)$$

The length of the quantas to achieve a desired client response time and throughput are estimated by inverting the above equation and solving for the quanta times. The description of the *policy* for choosing the quanta times in Ursa Minor is beyond the scope of this paper, and is explained by [34].

4.2 Adding observation-based models

The observation-based models in Ursa Minor are responsible for helping model designers detect new structural and performance behavior in the system. Stardust keeps track of the requests at entry and exit points for each service center in the system. Every

[4]For reads, "absorbed" means the request is found in cache; for writes "absorbed" means a dirty buffer is overwritten in cache.

	Workload runtime (s)	Performance overhead %	Read path (#queries)	Write path (#queries)	RAM for in-memory DB (MB)	Total activity records generated (MB/s)
Postmark	9792	0	8.5	19	220	0.34
OLTP	636	1.4	8.5	135	103	0.57
IOzone	329	5.3	11.4	109	323	3.37
Linux-build	1786	0	8.2	42.9	103	0.49
Sci	670	2.6	4	12.5	263	1.85

1: IRONModel's overhead and memory needed for efficient request flow analysis. Results are averages of 5 runs.

time a request passes through an instrumentation point, an activity record (containing fields such as timestamp, request ID, and relevant attributes to that service center — usually 64-128 bytes in total length) is stored in a relational database. Using the information contained in the database tables, the system can discover how requests are propagating through it, even when few or no expectation models are available. The system periodically constructs and maintains a graph of how requests are flowing through the system.

The prototype in Ursa Minor for localizing performance deviations currently only works offline. It calculates various statistics based on observed throughput and response time and compares those to what expectation-based models predict.

Model retraining is done through Z-CART, which currently uses the DTX package as the underlying classification and regression tree structure [6]. Each resource has one Z-CART model. Figure 6(b) shows the initial attributes Z-CART models used (in bold), and other attributes they subsequently correlated with performance. For the latter attributes, we knew their correlation with performance was non-negligible, but not the exact strength of correlation. We exposed these attributes to Z-CART, which ranked their correlation and incorporated them into its tree structure. It is worth stressing that the process of discovering new correlations is incremental. The attributes presented here represent what we know so far, from the various experiments we have run on Ursa Minor. Future workloads could expose the need for having more attributes to chose from.

4.3 Summary and discussion

This section describes how systems can be made to efficiently accommodate a modeling architecture like IRONModel. All models in Ursa Minor are built-in and written by the system designers. The apparent simplicity of the above expectation-based models reflects much thought we placed in the system design process. For example, the system was designed with the property that workloads can be analyzed separately from one another through algorithms for performance isolation [34]. Just that design consideration simplified the expectation-based models considerably.

This paper's testbed is Ursa Minor, however we have incorporated a similar instrumentation infrastructure and similar expectation-based models in a legacy database system (SQL Server) as well [20]. That gives us faith that the measurement and basic modeling infrastructure can be used in other systems. The algorithms that make the modeling robust, however, have only been incorporated in Ursa Minor and adding them to SQL Server is part of future work.

5. BASELINE EVALUATION

This section briefly evaluates key properties that demonstrate IRONModel's efficiency.

5.1 Experimental setup

For this baseline evaluation we only use a subset of the machines in the cluster. Clients are run on machines with Pentium 4 Xeon 3.0 GHz processors with 2 GB of RAM. Unless otherwise mentioned, all storage-nodes have Pentium 4 2.6 GHz processors with 1 GB of RAM; each has a single Intel 82546 gigabit Ethernet adapter. The disk configuration in each computer varies and disk capacities range from 8 to 250 GB. All computers run the Debian "testing" distribution and use Linux kernel version 2.4.22.

The workloads used for this baseline evaluation are drawn from a pool of representative storage system workloads. All workloads use a 1-of-1 encoding, i.e., the data is not replicated. Each workload touches a total of 3 machines for operation (client machine, metadata server and storage-node). **Postmark** is a single-threaded file system benchmark designed to emulate small file workloads, such as e-mail and netnews [14]. **OLTP** is a TPCC-like [32] benchmark that mimics an on-line database performing transaction processing. 10 clients access a 5 GB database concurrently. **IOzone** is a single-threaded file system benchmark that can be used to measure streaming data access performance [21]. For our experiments, IOzone measures the performance of 64 KB sequential writes and reads to a single 2 GB file. **"linux-build"** is a development workload measures the amount of time to clean and build the source tree of Linux kernel 2.6.13-4. **Sci** is a scientific workload designed to analyze seismic wave-fields produced during earthquakes [3].

5.2 IRONModel's overheads

5.2.1 Collecting request flow graphs

Stardust's efficiency as a general measurement infrastructure is first evaluated in [31]. Here we show new results pertaining to its usefulness as an infrastructure for model checking.

There are approximately 200 entry and exit points from service center types in Ursa Minor that post activity records each time a request flows through them. Stardust places demands on the CPU for encoding and decoding trace records, as well as network and storage demand for sending the records in relational databases. It also places a fixed demand of 20 MB of buffer cache at each computer for temporarily buffering records. The impact of the instrumentation on the above workloads is observed to be from 0-6%, as shown in Table 1. The amount of trace data collected can be significant (as is the case in the IOzone case). However, after problem localization, only the data related to anomalous behavior needs to be kept. In all the above experiments no anomalous behavior was observed and all the data collected can be eventually discarded.

5.2.2 Cost of parsing observations

An important property of Stardust is ease of querying. In particular, creating a request flow graph (or causal path) for a request is a common operation that needs to be efficient. Table 1 shows the average number of SQL queries required to create such a path for each of the workloads. All queries are simple SELECT queries that just select all columns of a row with a given request ID. These queries do not involve joins (the joins are done by the algorithm that creates the request flow graph in memory, and not by issuing JOIN statements to the database). The request ID column has an index on it. The time to create a path depends mainly on how deep

requests flow through the system. The deeper a request flows into the system (e.g., when it misses in the client cache and has to go to the storage-nodes), the longer it takes to recreate its path. Writes, for example, tend to have deeper paths than reads, since many reads hit in cache.

Table 1 also shows the amount of RAM memory needed to keep activity records relevant to anomaly localization fully in an in-memory database to construct the request path efficiently for all requests. In practice, we have seen individual path creation times ranging from a few microseconds, when the request IDs are still in the buffer cache of the Activity DBs, to a few milliseconds when the request IDs need to be retrieved from disk. For example, creating request flow graphs for all requests of the OLTP workload when the database resides fully in memory, takes on average 2714 seconds (147 seconds for the 90546 read requests and 2567 seconds for the 111858 write requests). Creating request flow graphs for all requests of the Postmark workload when the database fully resides in memory, takes on average 526 seconds (42 seconds for the 25673 read requests and 484 seconds for the 150086 write requests).

In practice, only those requests for which the client complains or which have a response time above a service-level objective threshold (e.g., one standard deviation away from the expectation) need to be checked.

5.2.3 Cost of creating expectations

Expectations are given by the expectation-based models described in Section 4.1. For a given workload, the expectation-based models report on the CPU and network demand expected to be consumed per request (the CPU and network models take less than 1 ms to make these predictions). The expectation-based models report on the expected cache behavior of a request and its expected disk service time. We have observed that for cache hits the cache simulator and real cache manager need similar times to process a request. The cache and disk simulators are on average three orders of magnitude faster than the real system when handling cache misses. The simulator spends at most 9,500 CPU cycles handling a miss, whereas, on a 3.0 Ghz processor, the real system spends the equivalent of about 22,500,000 CPU cycles. For a given client, the cache and disk models create expectations for common cache sizes once (e.g., 512 MB, 1 GB, 2 GB) and then memoize the results.

5.2.4 Cost of Z-CART operations

Asymptotic theoretical bounds for Z-CART operations were given in Section 3.3.1. The base CART algorithm itself has been extensively studied [18] and we do not intend to replicate those results. Instead, we report empirical observations relevant to this paper, for one of the models in the system, the CPU model. The Z-CART CPU model was build from actively probing 2.5 million possible CPU configurations that cover commonly-used data encoding options on 200 servers in the data center. Actively probing the system to generate the above configurations took approximately 30 minutes. Building the model (i.e., analyzing the CPU demand for each of the 2.5 million configurations) took 3 minutes. Z-CART needed 200 MB of RAM to do the analysis. The final decision tree was a 4 MB data structure, 21 levels deep and contained 34000 nodes. Querying Z-CART to make a new prediction took approximately 0.02 ms, which in practice means that 1 million new predictions take approximately 20 seconds to make. Hence, the overall cost is acceptable. Similar observations can be made for the other models.

6. CASE STUDIES AND EXPERIENCE

This section evaluates the efficacy of IRONModel to locate mismatches and revise expectations. The performance anomalies described here happened while we ran the above workloads nightly to stress different parts of the system.

Type of bug	Localization method	Who will fix?	#
System misconfig/bugs (24)	IRONModel	Administrator	1
		Programmer	17
	Other technique	Administrator	1
		Programmer	6
		Client	1
Model bug (5)	IRONModel	Re-training	5
	Other technique	Programmer	1

2: Key performance-related problem categories (2004 - 2007). Other techniques to find bugs included looking at *printf* statements, using *gdb*, and trial-and-error. Some problems fall into multiple categories.

6.1 Overall effectiveness

Table 2 shows performance problems that we encountered while building and experimenting with Ursa Minor. Out of 29 problems encountered, 23 of them were because of system misconfiguration or bugs in the code. For five of the problems, the models themselves had limited regions of operation. Many problems (23) first arose before IRONModel (and even Stardust) was incorporated in the system and we replicate them as best as possible with the current code base.

The column "Who will fix?" shows who we think will ultimately have to fix the system or model (for Ursa Minor, we were both administrators, programmers and sometimes clients; in practice these roles might be separate).

6.2 Localizing sources of mismatch

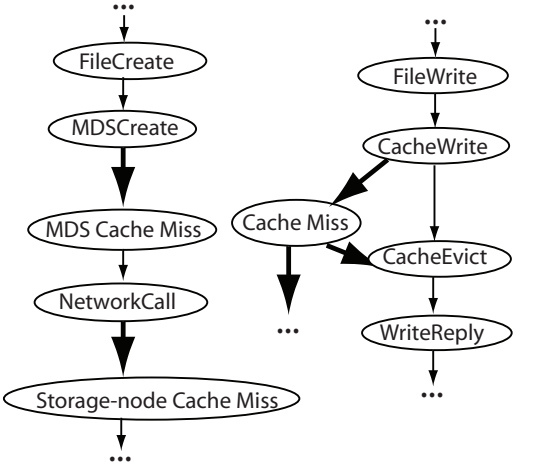
This subsection describes a few concrete examples when the localizer worked well and when it did not. For this discussion, we differentiate between cases when the system was found to be buggy or misconfigured, while the models correctly reflected the designers' intentions, and cases when the system was operating as expected, but the models had limitations and incorrectly flagged the system behavior as suspicious. This classification was done after the root cause of the problem was discovered and is based on whether the system or the models had to be eventually fixed. However, both the system component and its model are flagged as suspicious before the root cause is identified.

Buggy system implementations or misconfigurations: A representative problem in this category relates to poor *aging* of data structures that leads to degradation in performance. Two concrete instances of this problem that we experienced are degraded hash table performance over time and degraded disk performance as the disk gets fuller. For this discussion, we focus on degradation of hash table performance. An initial hashing algorithm had a bug that led to non-uniform hashing. Figure 7 shows how the latency graph changes as more files are stored in the system. The Postmark benchmark is run, and the measurements are taken after 10,000 and 20,000 files are created. The drastic degradation over time is not anticipated by the models (that predict the same disk service time in each case). When the problem was experimentally replicated, the storage-node model raised a flag to report the discrepancy in expected and measured performance. The localization directs the human's attention between two instrumentation points.

For most other system bugs or misconfigurations we have seen (e.g., unexpected locking or retry loops), the mismatch is shown as a change in edge latencies or number of edges.

(a) After 10,000 files created *(b) After 20,000 files created*

7: Performance degradation over time. A hashing algorithm bug leads to unexpected performance drops over time (bold edges). The nodes contain information such as the name of the component posting the record. The edges contain latency information. This is a simplified graph of a larger graph (15 nodes) that is not shown here for simplification.



(a) Performance deviations *(b) Structural deviations*

8: Multiple mismatches due to a software upgrade. These (simplified) request flow graphs contrast the behavior of requests under the 2.4 and 2.6 Linux kernel. File creates experience performance deviations (bold edges indicate latency differences). File writes experience structural deviations (bold edges indicate entrance to a new service center — the cache read handler). Surprisingly, in 2.6 the client's writes were unaligned, resulting in read-modify-writes.

Model bugs: A representative problem in this category relates to non-linear behavior of the individual models that we did not correctly model initially. A concrete instance involves the CPU model. As described first in Section 3.3.1 and then in Section 4, our initial CPU model under-predicted the CPU demand when the block size used was small (e.g., 512 B). In queuing terminology, we had not foreseen the effects of a service center inside the kernel, the network stack. 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 first. 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. The discrepancy showed up between two instrumentation points at the entrance and exit of the encode/decode module and the human's attention was directed there. Section 6.3 describes a bug with the network model as well, and shows how both CPU and network model can evolve over time semi-automatically.

Handling multiple symptoms: There were times when IRONModel detected multiple mismatches in the system. A concrete case of this happening is when we upgraded the Linux kernel from version 2.4 to version 2.6. Several performance benchmarks experienced performance changes after that. Through Stardust, latency graphs were obtained for each of the workloads under the 2.4 and 2.6 versions and compared. Furthermore, each of the expectation-based models self-checked to see if the expectations matched the observations. In practice, one might run different benchmarks and examine the paths from those that showed fewest discrepancies first. That is the approach we took. Figure 8 shows two request flow graphs for two different file system calls (file create and file write) for the Postmark benchmark. Bold edges indicate discrepancies between model and system.

Handling adaptive workloads: Adaptive workloads might change their behavior depending on the performance of the underlying system. Traditional models assume that, whether Ursa Minor is "slow" or "fast", any Ursa Minor client's sequence of operations will be the same. This assumption holds well, especially for static benchmarks. Many real-world applications are also not adaptive. However, there are some applications, such as web servers, that might experience a different workload depending on the speed of the underlying storage system. That happens, for example, when a web user might depart from the site (e.g., if it is too slow).

As a concrete instance, we modified the OLTP benchmark to keep changing its indexing behavior as a function of Ursa Minor's performance. Our initial models assumed that workloads were not adaptive, and were not helpful in diagnosing why the predicted and observed performance were different. For example, the buffer cache model would simulate the effect of doubling the buffer cache size and find it beneficial to double it. After the cache size was doubled, the sequence of accesses the database sent to it were different from what was originally simulated. IRONModel detects discrepancies at the buffer cache accesses before and after predictions are made, and chooses not to make predictions for adaptive workloads.

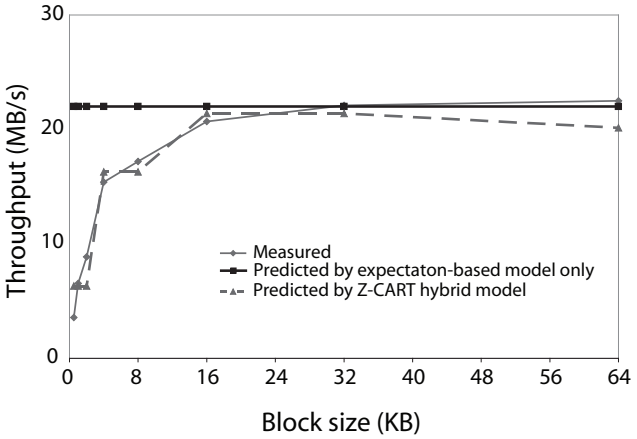
Example limitations: IRONModel has some limitations. One category of such problems involved anomalies external to the storage system. For example, the client running the scientific workload was having a problem with the network link just before entering Ursa Minor. From IRONModel's perspective, the requests were behaving as expected, once in the system. A second more subtle example involved implicit push-back from the system to the client. In one concrete instance, Ursa Minor had a thread exhaustion problem at a storage-node that implicitly put pressure on the client to send fewer requests. The symptoms of this problem were that fewer outstanding requests were observed in Ursa Minor. However, the number of outstanding requests is an input to the models, not a metric that they can predict.

A second category of limitations involved components we had no direct control over (and thus did not instrument). For example, this included performance problems inside the Linux kernel (all of Ursa Minor code runs in user-space). For such problems, IRONModel attributes the discrepancies to the black-box component, but cannot be more fine-grained.

6.3 Retraining and fidelity

This section illustrates, through several concrete examples, how observation-based models could help model designers discover new workload-system correlations. In addition, this section describes how fidelity can be reported with each prediction.

Unexpected CPU bottleneck: The localization of this problem instance was described in Section 6.2. Examining this situation with observation-based models after the fact, we found that the models could localize the problem between two Stardust instrumentation points. When all resource models (CPU, network,



9: Improved prediction from hybrid models. After learning the new correlation from observations on a particular server, the results from the Z-CART model generalize on any server with the same CPU type. The standard deviation for the measured performance is negligible, whereas for the predicted performance it is shown in Figure 5. Throughput is estimated as $1/D_{c,CPU}$ in this example.

cache, disks) self-checked, the CPU model was found to be the culprit (i.e., it under-predicted the CPU demand). Z-CART noticed that the attribute "block size" was significantly smaller than in the test cases and eventually incorporated that attribute in the decision tree (parts of the resulting tree and fidelity values were first shown in Figure 5). Of course, "block size" is an attribute that the programmer had to expose to Z-CART, for Z-CART to discover the correlation. Figure 9 shows the improvement in accuracy from the hybrid modeling technique.

When striping goes wrong: As described in Section 4, 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. In experiments, we observed that a particular configuration led to a workload having 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, see Figure 1 for symptoms).

The manual diagnosis of the problem, done when it was originally encountered, 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. Deeper manual investigation revealed that overflowing of switch buffers, leading to packet drops, was the root cause. That started TCP retransmissions on the storage nodes. The problem is known as the "incast" problem [23], and is rather unique to distributed storage systems that read data from multiple sources synchronously.

Using model self-checking after the fact, the diagnoses are better guided. For example, the cache model predicts that the workload would get a hit rate of 10% with 256 MB, and indeed that is what the workload is getting. However, the network model reveals that remote procedure calls (RPCs) are taking 20 ms, when they should only be taking 0.2 ms.

For erasure coding schemes, this incast problem is related to m, the number of nodes the client reads from. To make matters worse, the minimum m for which this problem arises is dependent on the switch type. The switches in Ursa Minor are off-the-shelf commodity components, and IRONModel considers them them to be black-box (i.e., IRONModel has no understanding of their internal algorithms). Furthermore, we cannot afford to replace them. To explore how Z-CART could help, we exposed to it *switch type* as one of the attributes to check. We then ran 825 experiment instances with erasure coding schemes chosen randomly among 5-of-6, 7-of-8 and 9-of-10. Given this experimental setup, the Z-CART network model automatically adjusted its expectations for the relationship between performance and m, as shown in Figure 10.

Example limitations: The main limitation of retraining stems from a common problem with most machine learning algorithms: correlations do not imply causality. Whenever a change to the system (for example, upgrading the Linux kernel) changes several system attributes at once, all of those attributes will have the same correlation ranking with a subsequent discrepancy. Solving this issue does require some human intuition in understanding which attributes caused the others to change in the first place.

7. CONCLUSIONS

This paper, through case studies with models for common resources found in distributed systems, analyzes why and how performance models get out-of-sync with the system over time. We find that traditional models are brittle because they assume idealized system-workload interactions. The reality is that both the system and the models are often buggy or misconfigured and unanticipated behavior is the norm. The main contribution of the paper, IRONModel, is a general robust performance modeling framework.

IRONModel incorporates models as first-class citizens inside a system. By using the redundancy between each model's high-level specification and actual system implementation, IRONModel can localize performance anomalies and can give hints to the system and model designer regarding the root-cause of the problem. A specific implementation of IRONModel in an experimental cluster-based storage system has proved successful in reducing the administrator's search space for most anomalies experienced.

8. ACKNOWLEDGEMENTS

We thank our shepherd Prashant Shenoy and many anonymous reviewers for their feedback and suggestions. We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, Network Appliance, Oracle, Seagate, and Symantec) for their interest, insights, feedback, and support. This work is supported in part by NSF grants CNS-0326453 and CCF-0621508, by Army Research Office grant DAAD19-02-1-0389, and by the Department of Energy award DE-FC02-06ER25767.

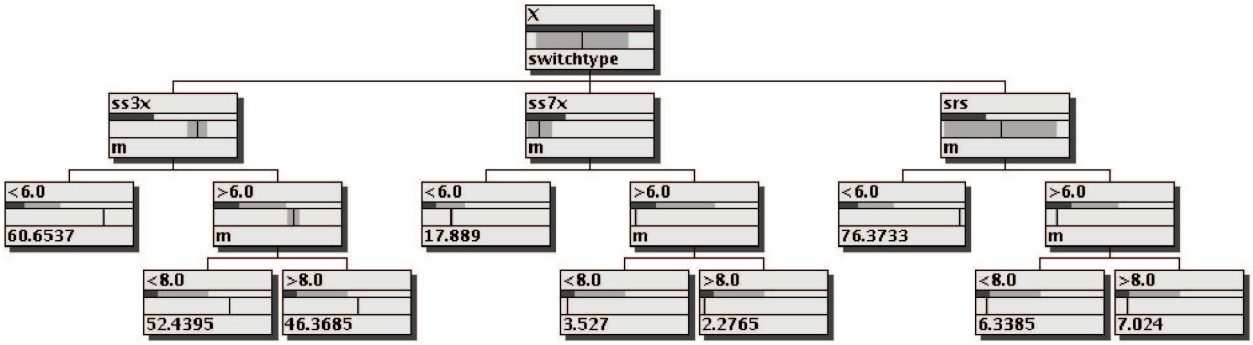
9. REFERENCES

[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. 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. ACM Symposium on Operating System Principles, pages 74–89, 2003.

[3] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High Resolution Forward and Inverse Earthquake Modeling on Terasacale Computers. ACM International Conference on Supercomputing, 2003.

[4] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, **23**(4):337–374, November 2005.



10: Z-CART adapts the predictions to the switch type. Ursa Minor has three types of donated switches (anonymized to *srs*, *ss3x*, *ss7x*), and each of them behaves differently as a function of *m* for erasure coding schemes. We treat them as black-box and Z-CART discovers their behavior over time. The location of the predicted value is relative to the minimum and maximum values observed (in this case 2.1 MB/s and 77 MB/s respectively). This is a simplification of the real Z-CART tree (it keeps several variables, such as block size, constant), which has over 30000 nodes and is more than 20 levels deep.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. Symposium on Operating Systems Design and Implementation, pages 259–272, 2004.

[6] C. Borgelt. DTreeGUI - Decision and Regression Tree GUI and Viewer, 2007. http://www.borgelt.net//dtgui.html.

[7] A. Chanda, A. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. EUROSYS, pages 17–30, 2007.

[8] 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. Symposium on Operating Systems Design and Implementation, pages 231–244, 2004.

[9] P. J. Denning and J. P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, **10**(3):225–261, September 1978.

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. ACM Symposium on Operating System Principles, 2001.

[11] S. D. Gribble. Robustness in Complex Systems. Hot Topics in Operating Systems, 2000.

[12] Q. He, C. Dovrolis, and M. Ammar. On the predictability of large transfer TCP throughput. ACM SIGCOMM Conference, pages 145–156, 2005.

[13] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 1991.

[14] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[15] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queuing network models*. Prentice Hall, 1984.

[16] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 37–48, 2007.

[17] Microsoft. System Center Capacity Planner, 2007. http://www.microsoft.com/systemcenter/sccp/.

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

[19] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. EuroSys, pages 293–304, 2006.

[20] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2005.

[21] W. Norcott and D. Capps. IOzone filesystem benchmark program, 2002. http://www.iozone.org.

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

[23] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. Conference on File and Storage Technologies, 2008.

[24] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. Symposium on Networked Systems Design and Implementation, pages 115–128, 2006.

[25] K. Shen, M. Zhong, and C. Li. I/O system performance debugging using model-driven anomaly characterization. Conference on File and Storage Technologies, pages 309–322, 2005.

[26] Sourceforge.net. Iperf, 2007. http://sourceforge.net/projects/iperf.

[27] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. Symposium on Networked Systems Design and Implementation, 2005.

[28] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments? */. ACM Symposium on Operating System Principles, 2007.

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

[30] 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. International conference on autonomic computing, pages 187–198, 2006.

[31] 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. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 3–14, 2006.

[32] Transaction Processing Performance Council. TPC Benchmark C, December 2002. http://www.tpc.org/tpcc/Revision5.1.0.

[33] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, pages 183–192, 2001.

[34] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. Conference on File and Storage Technologies, 2007.