Enabling what-if explorations in systems

ENO THERESKA

August 2007
CMU–PDL–07–103

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

To my family and friends.

Abstract

With a large percentage of total system cost going to system administration tasks, ease of system management remains a difficult and important goal. As a step towards that goal, this dissertation presents a success story on building systems that are *self-predicting*. Self-predicting systems continuously monitor themselves and provide quantitative answers to *What*...*if* questions about hypothetical workload or resource changes. Self-prediction has the potential to simplify administrators' decision making, such as acquisition planning and performance tuning, by reducing the detailed workload and internal system knowledge required.

Self-prediction has as the primary building block mathematical models, that, once built into the system, analyze past, and predict future behavior. Because of the traditional disconnect between systems researchers and theoretical researchers, however, there are fundamental difficulties in enabling existing mathematical models to make meaningful predictions in real systems. In part, this dissertation serves as a bridge between research in theory (e.g., queuing theory and statistical theory) and research in systems (e.g., database and storage systems). It identifies ways to build systems to support use of mathematical models and addresses fundamental show-stoppers that keep models from being useful in practice. For example, we explore many opportunities to deeply understand workload-system interactions by having models be first-class system components, rather than developing and deploying them separately from the system, as is traditionally done. As another example, lack of good measurement information in a distributed system can be a show-stopper for models based on queuing analysis. This dissertation introduces a measurement framework that replaces performance counters with end-to-end activity tracing. End-to-end tracing allows contextual information to be propagated with requests so that queuing models can attribute resource demands to the correct workloads. In addition, this dissertation presents a first step towards a robust, hybrid mathematical modeling framework, based on models that reflect domain expertise and models that guide model designers to discover new, unforeseen system behavior once the system is deployed. Such robust models could continuously evaluate their accuracy and adjust their predictions accordingly. Self-evaluation can enable confidence values to be provided with predictions, including identification of situations where no trustworthy predictions can be produced.

Through an analysis of positive and negative lessons learned, in a storage system that we designed from scratch as well as in a legacy commercial database system, this dissertation makes the case that systems can be built to accommodate mathematical models efficiently, but cautions that mathematical models are not a panacea. Models are as good as the system is; to make predictions more meaningful, systems should be built so that they are inherently more predictable to start with.

Acknowledgements

During my years as a graduate student at Carnegie Mellon University I have been very fortunate to learn from brilliant people. I'd like to thank my advisor, Prof. Greg Ganger, for his unbelievable belief in me, and his patience while I was growing up and learning from my mistakes. I stayed at CMU to work with him, and I am glad I did. I am looking forward to collaborating with Greg in the future. I'd like to thank the members of my thesis committee for their continous support. Prof. Mor Harchol-Balter and her inspiring class in queuing theory provided the necessary theoretical background for this dissertation. Prof. Raj Rajkumar has supported me ever since I started the education at CMU 9 years ago. Prof. Dave O'Hallaron and I have had interesting conversations throughout my years in graduate school. Dr. Dushyanth Narayanan was my mentor during my internship at Microsoft Research and we have continued to collaborate ever since.

Special thanks go to a large group of folks that have been part of my academic life during graduate school. In particular, John Linwood Griffin, David Petrou, Jiri Schindler, Steve Schlosser, Craig Soules, John Strunk and Jay J. Wylie are senior peers I have counted on for advice on career and life management. I have learned from them from my first day as a graduate student. Bill Courtright, the lab's executive director, has been a mentor whose advice I gladly take. Karen Lindenfelser and Joan Digney ensured I got everything I needed to be happy: extra points on homework, unlimited money for travel, good artwork, special pizza, and of course, abundant gossip about life at CMU!

Throughout graduate school I have collaborated with many brilliant students. I have especially enjoyed working with Michael Abd-El-Malek and I thank him for his heroic individual effort on the Ursa Minor project. I also thank him for providing good tea during our regular tea time in the lab. I thank all the contributors of the Ursa Minor project for providing me with a good testbed for my research. During my five years in graduate school, I have also enjoyed working with Michael Mesnier, Brandon Salmon, Raja Sambasivan and Alice Zheng on several other projects, and I thank them for their patience with me.

Pittsburgh has been a great place to live these last 9 years. I thank my roommate and colleague, Niraj Tolia, for making life bearable at home. Pittsburgh Filmmakers and the folks there were good to me during my secret studies in photography. The folks at the Animal Friends shelter provided furry animals for me to play with, which lowered my blood pressure on weekends. The many interesting folks I met in Pittsburgh (you know who you are) made my life more enjoyable.

Several people have been with me seemingly forever, and they know me best. I'd like to thank my family for providing everything for me. Without their encouragement, especially at tough times, I would not be who I am. Two generations in my family have been responsible for my upbringing, and devoted huge chunks of their life to me. I represent all of you now. My friends have always accepted me for who I am (perhaps after giving up hoping I'd change). I thank Ale, Alex, Cathy, Evis, Eszter and George-O, Micky and Timo for the past and for what is yet to come.

Contents

Figures

Tables

1 Introduction

This chapter motivates the need for building easy-to-manage and predictable systems and overviews the contributions of this dissertation.

1.1 System management and predictability

Much successful research and engineering has gone and still goes into building high-performance, scalable systems. As users continue to expect more from such systems, there is tension between system complexity and ease of management. Good, reliable systems, in the traditional sense, are met with skepticism and complaints from customers, because it is difficult to find skilled administrators to tune them [Lohman, 2007].

Ease of management has been recognized as a crucial next step in building robust systems, as can be seen by the call-for-arms from different research groups [Chaudhuri and Weikum, 2000; Ganger et al., 2003; International Business Machines Corp., 2001], creation of specialized workshops and conferences [ICAC, 2004; SAACS, 2005] and tracks on management in well-established systems conferences [OSDI, 2004; Usenix, 2006].

One major subset of management tasks involves performance tuning decisions, including decisions on trading off performance and other metrics such as availability, reliability, capacity and power consumption. Performance tuning decisions involve acquisitions, component configurations, assignment of datasets/workloads to components, and performance problem resolution. For example, in database systems, tuning problems include creation of indexes to speed up data lookup, layout of data on physical machines, selection of appropriate buffer cache parameters, etc. In storage systems, tuning problems include selecting the right data distribution and load balancing. In operating systems, tuning problems include determining when to upgrade resources (CPU/RAM/disks), how to best partition physical memory among virtual machines, determining which software models may be performance bottlenecks, etc.

For many of these management tasks, the most complex aspect is understanding the performance consequences of any given option. Non-performance issues, related to other metrics such as availability, capacity and power consumption, are also involved. But, these usually require much less understanding of the inner workings of system components and applications. Performance tuning consequences usually depend on workload specifics (e.g., the interleaved I/O patterns of the applications) and system internal algorithms.

Traditionally, administrators use two tools when making such decisions: their expertise and system over-provisioning [Weikum et al., 2002]. Most administrators work with a collection of rules-of-thumb learned and developed over their years of experience. Combined with whatever understanding of application and system specifics are available to them, they apply these rules-of-thumb to planning challenges. For example, one administrator might apply the rule "if average queueing delays are greater than 10 ms, then spread data/work over more disks" to resolve a perceived performance problem. Since human-utilized rules-of-thumb are rarely precise and have trouble scaling, over-provisioning is used to reduce the need for detailed decisions. For example, one common historical rule-of-thumb calls for ensuring that disk utilization stays below 30% (i.e., always have three times the necessary disk throughput available). Both tools are expensive, expertise because it requires specialization and over-provisioning because it wastes power, hardware and human[1] resources. Further, sufficient expertise becomes increasingly difficult to achieve as systems and applications grow in complexity.

This thesis investigates how systems can provide better assistance to administrators. I believe that systems should be *self-predicting*: able to provide quantitative answers to administrators' high-level questions involved with

[1]The additional hardware must be configured and maintained.

their planning. With appropriate built-in monitoring and modeling tools, this dissertation shows that systems can answer *What*...*if* questions about potential changes. For example, "*What* would be the performance of workload X *if* its data were moved from device A to device B?". With answers to such *What*...*if* questions, administrators could make informed decisions with much less expertise. Further, iterating over *What*...*if* questions (e.g., one for each possible option) enables a search-based approach to automating, or at least guiding, planning and tuning decisions. That approach has the potential to form a major building block for internal system optimizers.

Self-prediction has as its primary building block mathematical models that, once built into the system, analyze past behavior and predict future behavior. Because of the traditional disconnect between systems researchers and theoretical researchers, however, there are fundamental difficulties in enabling existing mathematical models to make meaningful predictions in real systems. In part, this dissertation serves as a bridge between research in theory (e.g., queuing theory and statistical theory) and research in systems (e.g., database and storage systems). It identifies ways to build systems to support use of mathematical models and addresses broad show-stoppers on the way to making them practical.

First, mathematical models need to be built inside the system they model, to better understand workload-system interactions that are hard to observe from outside the system. Second, mathematical models need good measurements from the system, with appropriate context, at the right abstraction levels. Aggregate performance counters, the measurement mechanism of most systems today, are fundamentally inadequate in shared, distributed systems. They are inadequate in shared systems because they do not expose per-workload information. In distributed settings, it is difficult to correlate or find causality links between them. For example, contextual information (e.g., "which originating SQL query led to high I/O load on the storage-nodes?") is impossible to obtain definitively. Without context, models are limited to making local optimizations. For example, a model that looks at statistics obtained at a single disk may determine that there is opportunity to reduce its requests' response times. However, a model that has more sophisticated statistics at all system levels may determine that only 1% of the requests go to that disk and focus should be placed on reducing network communication times instead. This thesis promotes end-to-end tracing as a fundamental measurement tool in any system. End-to-end tracing replaces existing counters with sequences of "activity records" that mark steps reached in the processing of any given request. Contextual information is carefully propagated with each request as it flows in the system.

Third, mathematical models need to be robust, or else they could become obsolete in the field. Models must be able to cope with component upgrades, human misconfigurations, and workload-system interactions that were not anticipated. This thesis presents first evidence that self-evolving models may be achievable. Such models continuously evaluate their accuracy and adjust their predictions accordingly. Self-evaluation also makes it possible to provide confidence values with predictions, including identification of situations where no trustworthy predictions can be produced.

Another show-stopper, separability of analysis, is indirectly addressed in this thesis. Clients using a system must be sufficiently insulated from one another so that their requests do not interfere in uncontrollable ways. Interference comes in terms of network, buffer cache and disk contention, and if not controlled properly it can lead to inherently ad hoc system behavior. In the face of such behavior, the performance of each workload can be highly dependent on the characteristics of the other workloads in the system (the models would have to check all possible ways the workloads could interfere, and/or make a worst-case interference prediction). Checking all dependencies leads to prohibitively expensive (and thus not scalable) predictions. Making worst-case predictions is not satisfactory in systems where worst-case and best-case predictions are orders of magnitude different (as is the case for storage systems). This dissertation shows that simple models can be a useful tool to ensure performance insulation. In turn, performance insulation ensures that models make predictions efficiently while still being simple enough for the system designers and programmers to construct.

The mathematical tools explored in this thesis come from queuing theory and statistical theory ("statistical theory" and "machine learning" are used interchangeably). This thesis shows how fundamental laws in queuing theory, known as *operational laws*, are a good starting point for modeling system components we are designing from scratch (thus, designers know the internal system "queues"). Statistical theory, on the other hand, has proven useful for further refining the queuing models in the field, and also to model off-the-shelf, legacy components whose internals are unknown.

These mathematical tools, once built into the system, allow the system components to expose *What*...*if* interfaces about their behavior. In a multi-tiered system *What*...*if* questions can be layered, with high-level *What*...*if* models combining the answers of multiple lower-level *What*...*if* models. For example, "*What* would be the performance of client A's workload *if* we add client B's workload onto the storage-nodes it is using?" needs answers to questions about how the cache hit rate, disk workload and network utilization would change. These latter resource-specific mathematical tools are based on a spectrum of approaches, from direct measurements to analytical formulae to simulation.

1.2 Thesis statement and contributions

The thesis of this work is:

Mathematical models can be integrated into systems and used to guide decisions related to a broad range of performance tuning problems.

My work and results support this thesis statement as follows:

(1) I show that a new measurement framework, based on end-to-end tracing, enables mathematical models to make predictions in shared, distributed systems. Such predictions were difficult, if not impossible to make before. I have incorporated this measurement framework (called Stardust) into a cluster-based storage system (called Ursa Minor).

(2) I show how combining traditional queuing models with machine learning models leads to a robust modeling infrastructure that was not achievable when these models were used in isolation. I have incorpo-

rated a prototype of this modeling infrastructure (called Observer) into Ursa Minor. The queuing models make good predictions in common regions of system operation. The machine learning models help identify workload-system interactions that were not anticipated and guides humans towards localizing sources of discrepancy. Observer provides confidence values with predictions, so that higher level policy layers can better judge if predictions are trustworthy.

(3) I demonstrate that a broad range of performance tuning decisions can be better guided using built-in mathematical models. Such decisions include data encoding, placement and system upgrades.

1.3 Example What...if explorations

This section provides illustrative interactions with Observer, the part of the system that is responsible for answering high-level *What...if* questions, posed by either external administrators or internal system tuning modules. The first and third examples are treated and evaluated in later chapters in this dissertation. The second example fits the broader claims of this thesis, but it is not evaluated in this dissertation. Chapter 2 will develop a more formal taxonomy of problems that this thesis does and does not address.

Should I migrate the data?: Whenever new storage-nodes are purchased, a decision needs to be made as to what data should be migrated to them. Ideally, the storage system itself would provide answers to questions of the form "*What* would be the performance of workload X *if* its data is migrated to the new storage-nodes?" The process of deciding what data should be migrated could then be reduced to searching for the best answer to a series of such *What...if* questions.

To answer such a question, Observer would first determine the *inherent demand* that workload X places on relevant system resources (network, buffer cache and disk in this case). This could be done automatically, using a combination of techniques such as direct measurements, analytical formulae and simulation. For example, using an analytical formula, Observer could determine that workload X needs 1 ms of network time to transmit a single block to the storage nodes. Through simulation it could determine a function between allocated buffer cache size and hit rate and a distribution of disk service times for each request that misses in cache. Stardust, the measurement framework, would collect the traces at the buffer cache and disk levels in order to enable such simulation.

Observer would then *translate* the demands of workload X from the current storage-nodes to the new storage-nodes. For example, if the new storage-nodes only have 512 MB of buffer cache, a buffer cache simulator could calculate the cache miss rate that workload X would experience when moving to the new storage-nodes. A disk simulator could calculate the service time of the requests that miss in the hypothetical 512 MB cache. The new storage-nodes might have other workloads sharing them, and Stardust would collect information on how loaded each resource is. The information on the existing load would then be used to scale down the predicted performance that workload X would see on the new storage-nodes.

Finally, Observer would determine the new *bottleneck resource* that would result from the data migration. The bottleneck resource is the one where requests from workload X would spend most of their time. With the new configuration above, for example, workload X's bottleneck might shift from being the network to being the disk. Using bottleneck analysis, Observer could then compute and report that workload X would get at most a 10 MB/s throughput and average response time of 4 ms.

Should I turn off machines to save power?: An administrator of a data center may consider turning off a subset of the servers over the weekend, to save power. Ideally, the system itself would provide answers to questions of the form "*What* would be the performance of each workload *if* servers 1-54 are turned off?" to assist with the decision.

Observer would first determine which workloads make use of those servers through the traces collected by Stardust. Some workloads might be replicated on other servers and could continue to run even if some replicas go offline. For those workloads, Observer needs to calculate the new performance that results from taking some resources offline. Other workloads might not be replicated, and hence turning the servers off would mean that their performance would be zero.

Observer would then determine how requests would flow in the system if some workload replicas were to be turned off. For example, if a workload's data was originally replicated 3 times, Observer would expect that on a write, 3 storage-nodes would be contacted. With one of the replicas going offline, only 2 storage nodes would be contacted on each write. That means less load on the network. But, less parallelism would be achieved during reads. Using operational laws from queuing analysis, Observer would estimate the new throughput and response times of the workloads and report them to the administrator. The administrator might check the service-level agreement it has with the cluster's clients (e.g., are they willing to tolerate a 20% performance drop to save 10% on their power bill?) and then decide whether turning off machines is safe.

An unexpected bottleneck: A bank may decide to encrypt customer data before sending it to storage-nodes. Ideally, the system itself would provide answers to questions of the form "*What* would be the performance of workload X *if* its data is encrypted?" Encryption places a large demand on the CPU resource of the client machine, and Observer needs to predict that demand. To do so, Observer poses the low-level *What*...*if* question "*What* would be the demand placed on the CPU *if* data was encrypted?" to the component responsible for encryption. That component encrypts and decrypts a block and reports that it takes 1 ms to encrypt/decrypt a single 16 KB block of data.

Imagine a situation in which a certain workload gets less than half of its predicted throughput in the field. A (traditional) manual inspection of the resources consumed might reveal that the model was significantly underpredicting the amount of CPU consumed and thus did not flag the CPU as a potential bottleneck. Through trial and error, it could be 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 first. The initial CPU model might be built using commonly-used block sizes of 8-16 KB for which the per-block cost is amortized by the per-byte cost. The model designers might not have foreseen the different behavior from small block sizes (it is unreasonable to expect designers to foresee all possible behavior).

A solution to such problems promoted in this dissertation is self-evolving, robust models. Using self-evolving models could help in two ways. First, the models themselves would localize the problem by self-checking during usage. All resource models (CPU, network, cache, disks) would self-check and the CPU one would be found to be the culprit (e.g., it may have predicted each block needed 1 ms of CPU time; in reality it was taking 2-3 ms). Second, a statistical model would correlate the attribute “block size” with the new behavior of the CPU and would eventually adjust the initial model to handle the unexpected low performance behavior. This dissertation examines the role of models in localizing problems and discovering new correlations, and role of humans in finding the actual root-cause of the problem and building systems to expose more measurements to the models.

1.4 Thesis map and bibliographical notes

This dissertation consists of 7 chapters. Chapter 2 presents broad background and related work. It presents a taxonomy of common management problems and focuses on performance self-prediction as a core technology towards self-managing systems. The state-of-art in addressing most of these management problems uses rules-of-thumb and over-provisioning. This chapter describes why that is the case and explains the key challenges humans face when they attempt to make predictions in distributed, shared systems.

Chapter 3 presents a common mathematical modeling infrastructure that, once built into a system, could be used to make performance predictions. A taxonomy of possible solutions is presented, based on how “black-box” the system under consideration is assumed to be (i.e., based on how much knowledge designers have of the system: are they building it from scratch or is it a legacy component?). Show-stoppers that prevent this modeling infrastructure from working in practice are discussed.

Chapter 4 describes the solution to a main show-stopper: measuring without context. That chapter is self-contained and includes relevant related work. The solution makes use of built-in, fine-grained, per-request, per-resource statistics causally linked in a distributed system. This monitoring layer exports querying interfaces to the models presented in Chapter 3 and enables them to produce informed predictions. This chapter evaluates the effort that goes into building such a monitoring infrastructure and its overhead on system resources and foreground performance. The main message of this chapter is that monitoring should be ON at all times and can be efficient.

Chapter 5 evaluates the efficacy of the Observer infrastructure in making predictions. Common case studies, drawn from the Ursa Minor storage system testbed [Abd-El-Malek et al., 2005], are used to illustrate how the infrastructure works and its expected accuracy. Ursa Minor is an excellent testbed since it has many performance tuning knobs, and it would be exceedingly difficult for an administrator to fully understand the consequences of using them. That chapter shows how the system itself can answer *What...if* questions relating to data encoding, data placement, and system upgrades. It also evaluates the need for separability of analysis and how models help achieve that.

Chapter 6 addresses another show-stopper: handling unforeseen behavior. The prototypes described use observation-based machine learning techniques to augment the expectation-based models described in Chapter 3. Observation-based models attempt to find new, unforeseen correlations between system-workload attributes and eventual performance. Strongly correlated attributes are eventually incorporated into the expectation-based models (after verification from the designer/programmer.)

Chapter 7 summarizes the contributions of this thesis. Appendix A discusses lessons learned when a legacy system is made self-predicting and describes the steps involved.

Parts of this dissertation have been previously published. A white-paper that made the case for building self-predicting systems appeared in [Thereska et al., 2005]. Chapter 4 appeared in [Thereska et al., 2006].

Most of the ideas presented in Chapter 3 and Chapter 5 were published in [Abd-El-Malek et al., 2006; Thereska et al., 2006]. A white-paper making the case for building robust self-evolving models (discussed in Chapter 6) appeared in [Thereska et al., 2007]. Appendix A contains ideas first published in [Narayanan et al., 2005, 2006]. Finally, the architecture of Ursa Minor, the testbed that is used for evaluation purposes throughout this case study, is described in [Abd-El-Malek et al., 2005].

2 Background

This chapter provides background and discusses broad related work on management tasks and mathematical modeling of computer systems.

2.1 Management tasks and current approaches

As systems become more complex, human administration becomes more expensive. Current methods to scale out systems often sacrifice predictability and ease of management for traditional "success" metrics such as performance and availability. However, with total system cost being a better metric [Moore, 2005], it becomes clear that administrative and management costs are non-negligible. For example, an organization purchasing a large-scale database or storage system today will spend more on administration and management than for actual software and hardware combined [Allen, 2001; Gray, 2003; Moore, 2005].

"Management" is a catch-all term for a large set of tasks, making it a vague target for improvement. To clarify the targets of this thesis, this section places the problems we target in the context of other system management tasks, shown in Table 2.1. Our compilation of this list is influenced by systems on which this dissertation has concrete evidence from, namely storage and database systems. I believe these tasks are also representative of operating systems and networks systems (storage and database systems sometimes make use of these latter two). There are other system types, such as sensor networks and real time embedded systems for which this list may be different, however.

Task	Sub-tasks	Description
Protection	*Data corruption*	User mistakes or software problems
	Small-scale failures	E.g., disk head failure
	Large-scale failures	E.g., building collapse
	Archiving	For short- and long-term
Tuning	**User complaints**	Root-cause analysis of user complaints
	Index creation	Speedup by indexing
	Memory allocation	How to partition RAM
	CPU allocation	How to partition work among CPUs
	Data encoding	Should one use replication or RAID
	Data placement	Where should I place the data?
	Device configuration	Configuring purchased devices
Planning	**Initial capacity planning**	Determining initial purchases
	Continuous capacity planning	Planning for system growth
	Software updates	Updating the software of the system
Monitoring	**System inventory**	Keeping track of resources
	Statistics management	Collecting, organizing and accessing various system statistics
Repair	*Restoring data*	E.g., after failure restore from backup
	Replacing failed components	E.g., failed disks must be replaced eventually
Security	*Investigating intrusions*	Going over access logs
	Preventing breaches	Applying patches

Table 2.1. **The focus of this thesis in the context of system management tasks.** The focus of our work has been on the entries in bold.

Although all of the management tasks listed are time consuming, several of them, particularly in the categories of performance tuning, capacity planning and system monitoring, are especially hard and are the focus of this thesis. These tasks require a deep level of understanding of the interactions between a workload and the system (software and hardware). Administrators often find it difficult to keep track of the capabilities of hundreds of system resources, their internal algorithms, and the characteristics of the many workloads supported. Most modern distributed systems do little more than export hundreds of performance counters to the administrator; she still needs to filter the irrelevant ones, analyze the relevant ones, and understand the consequences of a tuning decision.

In practice, administrators use rules-of-thumb and system over-provisioning ("KIWI" or kill-it-with-iron) to address tuning and planning tasks. Rules-of-thumb may work well in some cases, usually in small-scale systems [Weikum et al., 2002], but fall short in others, especially in heterogeneous distributed systems, where resources may be shared among many workloads. This is because rules-of-thumb are general (hence, rarely precise for the system and workloads at hand); furthermore, it is sometimes hard to resolve conflicts among rules-of-thumb [Salmon et al., 2003]. Because of these shortcomings, the KIWI method is used, as well, to reduce the need for precision.

Both rules-of-thumb and over-provisioning require the administrator to have specialized knowledge of the system and workload characteristics to know which rules-of-thumb to apply and what parts of the system to over-provision and how. Blind over-provisioning can be prohibitively expensive, adding not only to the acquisition costs but also to the administrative and energy costs (power and cooling of the system). The goal of this thesis is to have the system itself provide quantitative answers to *What...if* questions that explore tradeoffs among (over-)provisioning, performance and predictability, all with minimal administrator involvement.

2.1.1 Why are the problems challenging?

It is challenging for human administrators to predict the performance consequences of potential changes. Doing so requires them to understand system internals (e.g., buffer cache replacement policies) and keep track of the workloads each resource is seeing (e.g., buffer cache records on each storage-node). The system itself is in a better position to keep track of this information. However, there are challenges that need to be overcome for that to happen:

– Lack of formality in system design. Without a formal notion of expected behavior built into the system, the system cannot answer *What...if* questions on hypothetical changes to that behavior. A formal notion of behavior would include a notion of *structural behavior* (i.e., how are requests expected to flow through the system?) and *performance behavior* (e.g., how long should it take for a request to be serviced from a generic disk?). It is arguably more difficult today to attempt to put a structure on system behavior due to added system complexity. Because of tight deadlines, system designers do not codify any notion of expected behavior into the systems they build, leaving it to administrators to guess the internals of the systems they deploy.

– Inadequate measurement infrastructures. Surprisingly, until recently, there have not been any good performance measurement tools for shared, distributed infrastructures. Good tools are needed to understand how each workload interacts with a resource. The main difficulty has been managing statistics for requests that may propagate through different service centers in a distributed system. By the time a request reaches service center N, all the context (e.g., where did that request originate and how does it relate to the rest of the workload?) is lost.

– Fragile tools that lose touch with reality. Many system administrators and designers we have talked to know well that systems do not always work as expected; hence, there is skepticism about using *What...if* tools to automate or guide administrative decisions. Any serious solution has to deal with rather harsh reality checks. First, models will have limited regions of operation (the designer cannot be realistically expected to model all possible workloads and interactions with the service center). These models will need to identify when they do and don't work and they also need to evolve over time, ideally with minimal human involvement. Second, the system may be misconfigured to start with (often by humans). Models are of more use if they help in localizing misconfiguration problems, but they must indicate their lack of fidelity in such cases.

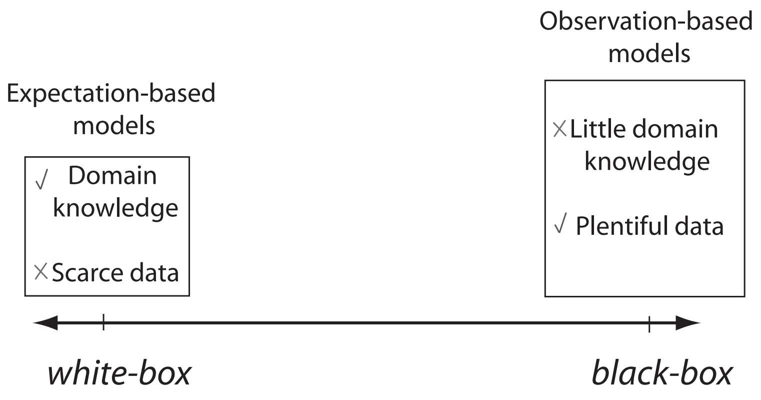


Figure 2.1. **Spectrum of system models.** System models range from expectation-based to observation-based, depending on how black-box the system is considered to be.

2.2 Toolbox of mathematical models and how they might help

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. Models could make predictions to answer *What...if* questions when it is prohibitively expensive to change the system to make real assessments regarding these same questions. The spectrum of modeling approaches, as shown in Figure 2.1, is bounded by purely *expectation*-based and *observation*-based models. Neither alone is adequate for robust modeling, but this dissertation shows that each has a role to play.

Expectation-based models: Expectation-based models use designer and programmer knowledge about how their systems behave; thus, the system is viewed as “white-box”. The models have a built-in, hardwired definition of “normalcy” (e.g., see [He et al., 2005; Perl and Weihl, 1993; Reynolds et al., 2006; Shen et al., 2005; Stewart and Shen, 2005; Uysal et al., 2001]). Indeed, highly accurate models have been built for disk arrays, network installations, cache behavior, and CPU behavior.

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

A myriad of methods are available for making performance predictions. For example, a CPU model may be based on direct measurements (i.e., it decrypts a block and reports on the time it takes). A network model may be an analytical formula that relates the request size and network speed to the time it takes to transmit the request. The cache and disk models could be based on simulation and could 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 well known (i.e., the model for the cache manager knows the replacement algorithm for the cache). This is the white-box method of modeling components.

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 [Aguilera et al., 2003; Barham et al., 2004; Chen et al., 2004; Cohen et al., 2004; Dinda, 2006; He et al., 2005; Mesnier et al., 2007; Wang et al., 2004]). These models are often used when components of the system are "black-box" (i.e., no knowledge is assumed about their internals).

For example, Mesnier et al. [2007] and Wang et al. [2004] built storage-node models based on observing historical behavior and correlating workload characteristics/attributes, such as inter-arrival time, request locality, etc., with storage system performance. The measurements of these attributes was done at the entrance and exit points of the black-box storage-node.

Observation-based models are an option when pre-existing models are not available. However, they require a large set of training data (i.e., previous observations), an issue that 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* workload mixes to make a reasonable performance prediction.

Many machine learning models are available to implement observation-based models, including neural networks, classification and decision trees, multi-dimensional regression tools, etc. Each of these tools shares the property that algorithms of the underlying modeled resource are not known; however, observations (also known as training data) are usually available, and the tools make an educated guess at how the black-box component operates.

A hybrid approach to modeling: Expectation-based models are the right starting point for future system designers, and we start by describing their design in Chapter 3. However, the issue remains on how to evolve these models over time with minimal burden to humans. Models become obsolete if they do not evolve as the system evolves. Indeed, we observed this to be the case with performance models in systems that we originally considered to be white-box, since we designed and built them from scratch. In addition, we found black-box behavior that resulted from either 1) unforeseen workload characteristics or system configuration characteristics, 2) unforeseen interaction among white-box components or 3) administrator misconfiguration of the system.

We have come to believe that a robust solution will need to augment expectation-based approaches with observation-based approaches. 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. This thesis explores this hybrid modeling technique.

There is little previous work done on evaluating how hybrid modeling approaches can keep models from becoming obsolete. Standard machine learning books (e.g., see chapter 12 of [Mitchell, 1997]) recognize that there could be positive consequences to augmenting learning methods with domain knowledge. A recent paper in network modeling evaluates the pros and cons of white-box and black-box techniques and proposes an investigation into hybrid models as future work [He et al., 2005]. Tesauro et al. [2006] investigates how machine learning tools can be used when there are transient changes in the system (e.g., machines added or removed) that cannot be easily modeled using expectation-based models that use queuing analysis. This dissertation provides a first view of the challenges and opportunities of hybrid models through case studies that touch many resource types (e.g., CPU, network, buffer cache and disks).

2.2.1 Queuing analysis as backbone

Irrespective of whether individual models are expectation-based or observation-based (or a hybrid), one needs to have a framework to reason about the effects on performance of a hypothetical change when combining multiple models. In general, *queuing analysis* is the building block of such a framework. In queuing analysis, the system is represented as a network of queues. Each queue represents a resource, and customers or workloads consume these resources. Some key results in queuing analysis that are especially relevant to this thesis are known as *operational laws*.

Figure 2.2 illustrates a network with two *service centers*, one representing some CPU processing, and the other some disk processing. The *queue* part of each service center indicates that a request may wait for other requests to complete before entering the *server* part of the service center. Let's illustrate a common operational law through this diagram. This law is commonly known as Amdahl's law. Intuitively, it states that even if a component of a system is sped up by several fold, that does not mean that the overall system speed is increased by the same magnitude. For example, if the CPU in Figure 2.2 is doubled in speed, the overall response time decreases by at most 10% (intuitively, this best case happens when there is little or no queuing in the service centers). Chapter 3 will discuss in more detail how to model a system using a network of queues. Note that the properties of the individual resources could be derived from expectation models (e.g., a simulation of the disk) or observation models (e.g., observation on how similar disks have been behaving on the past).

There is much related work on queuing analysis, and especially operational laws, dating back 30+ years (e.g., see [Denning and Buzen, 1978; Lazowska et al., 1984; Menasce and Almeida, 1998]). This thesis does not invent any new queuing laws. The contribution here is to enable them to work in distributed, shared systems by removing show-stoppers that prevent them from working in practice. More context than given here is needed to fully understand these show-stoppers, and that is provided in Chapter 3. However, some of these show-stoppers are so fundamental that they can be understood with the context so far.

For example, queuing analysis assumes that detailed per-workload, per-resource information is available, but this information is not trivial to obtain. Lazowska et al., for example, state in Chapter 7 of their book that "...most current measurement tools do not provide sufficient information to determine the input parameters appropriate to each customer class with the same accuracy as can be done for single class models. This not only complicates the process of parameterization, but also means that the potentially greater accuracy of a multiple class model can be offset by inaccurate inputs." [Lazowska et al., 1984]. My work on end-to-end tracing described in Chapter 4 solves this problem and the problem of obtaining these statistics in a distributed system.

Fundamentally, queuing theory assumes knowledge of the way requests flow through the system's service centers. Our experience, discussed in Chapter 5, shows that, even when building a system from scratch, it is very difficult for system designers and programmers to enumerate all of a system's flows and their characteristics. Hence, this thesis describes an approach to making use of queuing analysis in systems more robust. That approach involves guiding model designers to discover new service centers in the system that have not been previously modeled and continuously calibrating param-

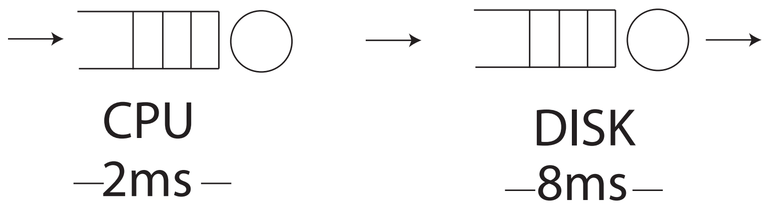


Figure 2.2. **A queuing network with two service centers.** Making the CPU twice as fast would only decrease response time by at most 10% (Amdahl's law).

eters of existing queues in the system. Chapter 6 introduces my approach for doing so.

2.2.2 Success stories using mathematical models

These have been several success stories in using mathematical models for performance analysis and prediction in real systems.

Success stories in capacity planning: Mathematical models have traditionally been used in initial capacity planning approaches. Initial capacity planning makes rough estimates about workloads that a system is expected to service and rough estimates on the capability of system components. For example, initial planning for setting up a system to host a World Cup web site may result in the following estimates: approximately 100,000 people a day are expected to view the web site (data from previous World Cups); 80% of the clients are interested in 20% of the web pages from the site, so enough memory needs to be purchased to cache all those pages; availability requirements may be high, but not as high as a bank or an online retailer. Hence, it is determined that all data should be replicated once, and for that, approximately 10 TB of storage space will be needed.

As a concrete example, the Disk Array Designer [Anderson et al., 2005] computes a good initial configuration of a storage system by iterating over the space of possible storage device attributes. Indy [Hardwick et al., 2001] attempts to predict the bottleneck shifts resulting from resource upgrades (i.e., changing system attributes). Many other case studies found in [Lazowska et al., 1984; Menasce and Almeida, 1998] show how the same approach can be used to design system components, from micro-processor farms to large-scale distributed systems with hundreds of diverse resources.

Most of this previous work assumes humans know the parameters to input into the mathematical models (workload characteristics and system characteristics). That assumption, unfortunately, does not hold well in the systems we looked at. A high-level difference between previous work and this work is that we make no such assumptions. The models we built are part of the system. As such, they continuously measure workload and system characteristics, for each workload and resource in a multi-tier system. A major focus of this dissertation is also designing systems to better assist models. As the remainder of this dissertation will show, unless care is taken in the system design phase to make the system algorithms inherently more predictable, models by themselves will not be sufficient for meaningful predictions.

Success stories in database systems: There have been a few examples of systems being able to answer *What...if* questions, especially in the area of database systems (see [Chaudhuri and Weikum, 2000; Weikum et al., 2002; Chaudhuri and Weikum, 2006] for calls to the database community to work on self-managing solutions). The AutoAdmin tool, for example, can answer *What...if* performance questions regarding creation of indices [Chaudhuri and Narasayya, 1998]. The DB2 advisor provides similar functionality [Valentin et al., 2000]. The prototype Resource Advisor we built into SQL Server answers *What...if* questions as a function of the cache size [Narayanan et al., 2005]. This thesis presents a framework that can answer a broad range of such *What...if* questions.

2.3 Alternative approaches to modeling

Modeling is not the only way to answer *What...if* questions. This section surveys other ways that reduce the need for having accurate models.

Direct measurements: Answers to *What...if* questions can come from direct measurements. For example, imagine that an administrator would like to know the consequences of migrating data from storage-node A to storage-node B. The administrator could migrate a small fraction of the data and observe the performance of that fraction over time. Indeed, if the data set is small, migrating the data would be a fine way to answer the migration *What...if* question. In general, however, there are many situations when the penalty of direct measurements would be high. For the same example above, it would be too expensive to perform direct measurements if the dataset was large and had to be moved as a unit.

Predictable systems: Hand-in-hand with having models to make predictions, systems could be built so that they are inherently more predictable. This does not entirely eliminate the need to model the system, but has the potential to reduce the complexity of models. High predictability is especially desirable for shared systems, where uncontrolled workload interference frequently leads to behavior that is difficult to model. A way to build predictable systems is to ensure the system has the *separability* property: the behavior of each workload can be analyzed separately form the behavior of other workloads. This requirement is very hard to meet, because, in practice, resource sharing means resource contention between workloads. However, the separability property can be relaxed to demanding that that any contention be bounded and/or predictable. Work done in this area revolves around insuring performance insulation of the various resources (e.g., CPU and network [Banga et al., 1998; Reumann et al., 2000] or buffer cache and disks [Chambliss et al., 2003; Karlsson et al., 2004; Wachs et al., 2007]. A concrete example that will be directly relevant when we consider the case studies in Chapter 5 is that of storage performance insulation [Wachs et al., 2007]. Whenever a prediction is made that workload W_n will get X MB/s of throughput, that prediction should not be annulled when another workload W_{n+1} enters the system. The work presented by Wachs et al. [2007] combines algorithms in smart cache partitioning, request scheduling, and deep prefetching/write-back to guarantee performance insulation. In turn, that keeps the models presented in this dissertation simple.

Another proposal for building more predictable systems has been to promote smaller, simpler components or appliances [Chaudhuri and Weikum, 2000; Magoutis et al., 2000; Sapuntzakis and Lam, 2003]. These components should do one thing and do it well. For example, Chaudhuri and Weikum [2000] describe how databases are getting ever more complex because they need to support advanced features of SQL that few people use. They call for "RISC-like" database components that behave predictably when answering simple, common SQL queries. Although system complexity definitely needs to be reduced, the "RISC-like" approach still needs models that answer *What...if* questions, albeit the models may be simpler.

3 Modeling for What...if analysis and the reality check

This chapter describes the challenges to achieving the goals of a modeling infrastructure: once built into a system, it should be able to answer a broad range of *What...if* questions. A main challenge for such an infrastructure involves ease of accommodation in the system. Another challenge is the ability to evolve over time.

3.1 Modeling goals, non-goals and assumptions

This section describes the main goals of the modeling infrastructure as well as the (few) assumptions it makes when operating in a general system.

Performance predictions as main focus: This dissertation's focus is on performance prediction and answering *What...if* questions related to performance. Performance, however, is only one of many dimensions that an administrator may care about. *What...if* questions may also be asked on power consumption, availability and reliability metrics, security and confidentiality metrics, etc. Chapter 5 will illustrate how these other dimensions manifest themselves in a cluster-based storage system, but the main evaluation will focus on the performance dimension. Answering performance *What...if* questions is a hard problem. It requires much understanding of the inner workings of system components and workloads.

Performance metrics and success metrics: The two main performance metrics of interest are throughput and response time. We differentiate between a *client's* performance and a *dataset's* performance. For example, if a database is accessed by two clients A and B, the clients' performance is the throughput and response time each of A and B receive. The dataset's performance is the maximum throughput that the database can serve to any number of clients. Although modeling should allow for both kinds of predictions to be made, the focus here is on predicting performance from the perspective of a client.

When making a prediction, as systems designers, we are not always concerned about traditional "success" metrics like accuracy (e.g., expressed in metrics such as root-mean-square-error). When evaluating a prediction algorithm, we are often concerned with maximizing particular system-specific "goodness" metrics. For example, a prediction algorithm 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.

Adaptive models: A main message of this thesis is that each system service should be designed with a *What...if* model as part of it. For example, the designer of the storage-node service should incorporate a *What...if* model of the storage-node that answers hypothetical questions on workload or resource changes. Whereas it is reasonable to expect system designers to construct good initial behavioral models for how the system should behave (this belief stems from our experiences described in Chapter 5), one cannot expect that the system designer is a modeling expert. The models should ideally refine themselves over time, or at least guide humans to understand the limitations of the model's operations.

Few new mathematical tools: This thesis does not directly create new mathematical tools. There are plenty of tools to choose from, and our focus is on how to build systems to make use of them.

3.1.1 The impact of workload characteristics

Workload characteristics/attributes have always been a decisive factor in determining how well models work. A general modeling infrastructure should make few, if any, assumptions about them.

In the general case, the main workload characteristic Observer (our modeling infrastructure) needs is the *demand* a request from the workload places on each of the system's resources. Queuing analysis is able to give bounds on mean throughput and response time based on only that information, as discussed in Section 3.3. What is gained by making no other assumptions is that reasonable performance bounds are always available. What is lost is that second-order metrics, such as variance of throughput and response time, cannot be predicted in the general case. To make such predictions, further assumptions about workload characteristics are needed (e.g., exponential inter-arrival times). Because the monitoring infrastructure collects detailed statistics about a workload (as described in Chapter 4), if such assumptions are found to hold, then further predictions (e.g., on variance) are possible. Thus, the infrastructure is general, but allows for more specialized models.

We assume that workloads will run for a while before any predictions are made. This assumption allows the monitoring infrastructure to understand the inherent demand that workloads place on a system's resources. In practice, this is a reasonable assumption. Even when buying a new system (e.g., a bank buying a new storage array), a workload has usually been running on a previous system (e.g., bank transactions have been running on the older array), which allows demands to be extracted. Administrators may decide to manually input information about future workloads into the modeling infrastructure, but, confidence in predictions is only gained over time.

There are several worthwhile notes on how to interpret the predictions based on several key workload characteristics:

Adaptive workloads: An implicit hope of any modeling tool, including Observer, is that the application will not radically change its behavior based on the performance of the underlying system. For example, Observer assumes that, whether a storage system is "slow" or "fast", the storage system's clients' sequence of operations will be the same. This assumption holds well in most cases, because most applications are not adaptive. However, there are some applications, such as web server workloads, that may experience a different workload depending on the speed of the underlying storage system. That happens, for example, when a web user may depart from the site (for example, if it is too slow) and go to another. In general, the adaptive workload can be considered as coming from a self-tuning system that adapts to our underlying system. Initial predictions for such workloads may be inaccurate, but the least Observer could do is “remember” not to predict again for such workloads.

Open-loop workloads: Workloads can be classified as open-, closed-loop (or hybrid), depending on their request arrival times. Briefly, the arrival time of requests in closed-loop systems is directly correlated with their departure time (e.g., a new request is sent only after a previous one completes). This correlation does not exist in open-loop systems (e.g., a client may send one request a second, irrespectively of how long each request takes to complete). Schroeder et al. [2006] provides a refreshing description of properties of such workloads. In general, prediction bounds for closed-loop workloads are tight for both throughput and response time. Intuitively, this is because the closed-loop nature of the workloads “self-regulates” their behavior. However, response time for open-loop workloads, may, at least theoretically, be unbounded. Intuitively, that is because the workload can, in theory, be infinitely bursty. However, in practice, our monitoring infrastructure observes the smallest, largest and average bursts in the system, and can usually bound the response time of the workload, based on past observations. Also, due to an inherent limit on the number of outstanding threads a system supports, high-rate open-loop workloads often still appear as closed-loop from the perspective of the system.

In general, there are mathematical formulae that specialize in open-loop and closed-loop workloads. The difficulties in making them apply in realistic systems are the same in both cases, and alleviating those difficulties is what this dissertation targets.

Phased-workloads: Phased workloads have repeatable phases of operation. Phases can be differentiated based on access patterns (e.g., sequential accesses followed by random accesses) or workload intensity (e.g., Monday’s vs. Sunday’s workloads). Figure 3.1, for example, illustrates the Cello and DEAS workloads as having phases of high and low loads. Any of the models

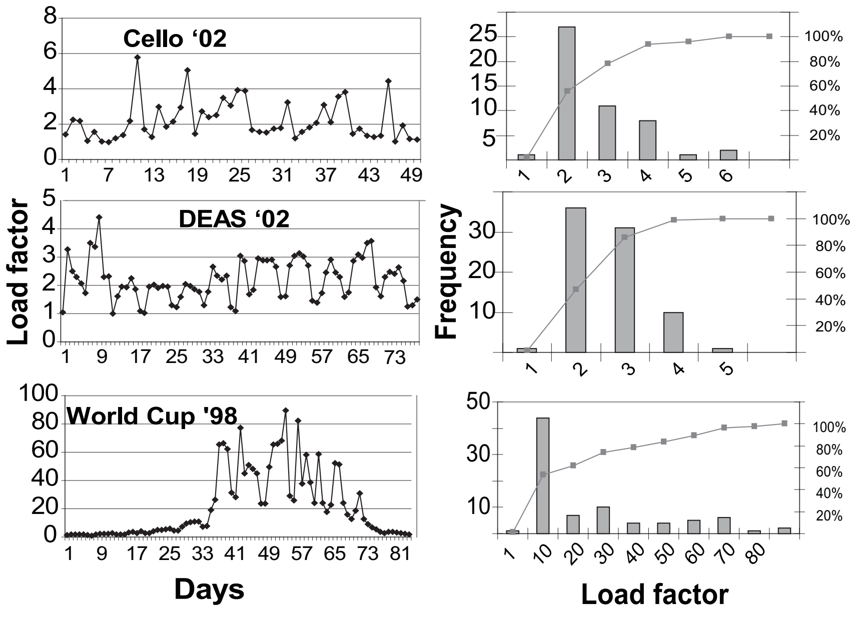


Figure 3.1. **Three workloads with different load distributions.** The Cello workload is taken from a storage array at HP Labs [Hewlett-Packard, 2007]. The DEAS workload is taken from a file server servicing departmental needs at the EECS department of Harvard University [Ellard et al., 2003]. The World Cup workload is taken from a web site hosting information about the Soccer World Cup in 1998 [Arlitt and Jin, 2000]. A load factor of 1 denotes the day with the lowest request arrival rates.

presented in this dissertation can make predictions based on the "tail" of the load distribution. For example, for the above two workloads, provisioning for a 90^{th} percentile load would mean handling cases in which the load is up to 4 times higher than the lightest load the system sees. The exact percentile desired can be quantified in the SLA. Predicting for the tail is a standard over-provisioning approach [Borowsky et al., 1998; Urgaonkar et al., 2002].

Drastically changing workloads: Figure 3.1 includes data for the workload that a soccer World Cup 1998 web site saw over the course of two months [Arlitt and Jin, 2000]. The load factor changes by 100 times when the actual cup starts. It is clear that past load history does not help the system with making predictions in this case. If the administrator has prior knowledge of future changes (e.g., World Cup starting, and a rough estimate on how that will affect the load on system resources) she can still use *What...if* models to guide her purchasing and provisioning decisions. If the administrator has no such knowledge, the system will only be able to make predictions after the change has been observed (in this example, after the first day).

3.2 From black-box to white-box: spectrum of solutions

The general architecture for the mathematical models in a multi-tier system is shown in Figure 3.2. As first discussed in Section 2.2, we identify two approaches in designing system models. Expectation-based models use designer and programmer knowledge of how systems behave; these systems are considered to be white-box. The models have a built-in, hardwired definition of "normalcy". Observation-based models do not make *a priori* assumptions about 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 that extrapolate future behavior from past observations.

Observer takes a hybrid approach and uses both. The benefit of using expectation-based models is that no training data (in the form of previous observations) is needed to make predictions. The disadvantage of expectation-based models is that they place some burden with the system designer to develop them in addition to the system. The benefit of using observation-based models is that they minimize that burden on the designer. Once "reasonable-enough" expectation-based models have been created, the observation-based models continuously refine them over time.

Expectation-based models, in general and as discussed in Section 3.4, may make use of analytical and simulation-based techniques. For example, model n's expectation-based part in Figure 3.2 may be a simulator that collects buffer cache traces, knows the cache manager algorithms, and replays the traces with a hypothetical cache size. Observation-based models, in general, and as discussed in Chapter 6, may make use of machine learn-

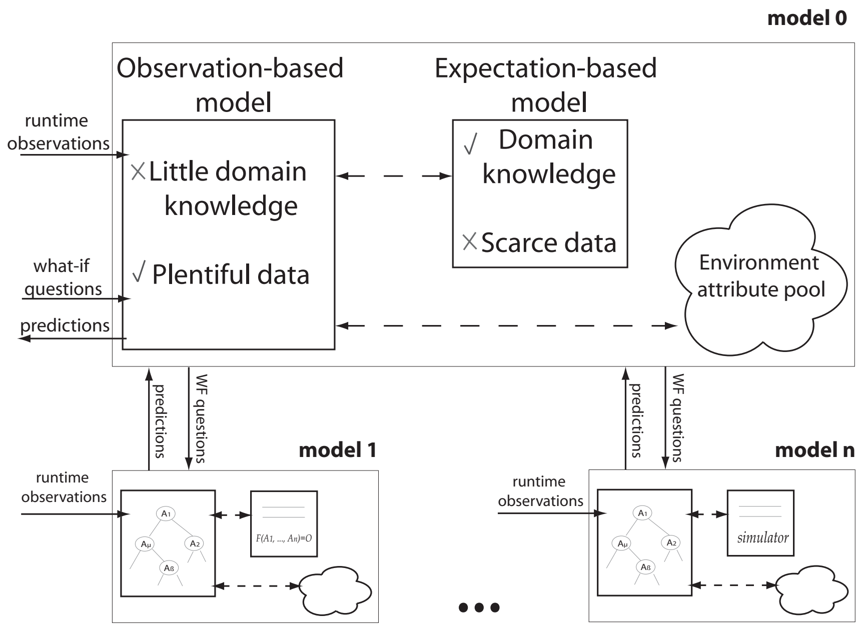


Figure 3.2. **Modeling architecture in a multi-tier system.** Each model self-checks and builds confidence for its predictions. New correlations discovered by observation-based models, using attributes exposed by the system designer, are eventually incorporated into expectation-based models by the model designer. Expectation-based models are discussed in this chapter and also Chapter 5. Observation-based models are discussed in Chapter 6.

ing techniques such as classification and regression models. These machine learning techniques continuously adjust the parameters of the analytical and simulation-based techniques that expectation-based models use. For example, model n's observation-based part may keep track of the accuracy of the simulator over time. Irrespective of whether individual models are expectation-based or observation-based (or a hybrid), one needs to have a framework to reason about the effects on performance of a hypothetical change in a system with multiple resources. In general, *queuing analysis* is the building block of such a framework, and the next section discusses it in greater detail.

3.3 Queuing analysis as backbone

This section reviews one of the two main mathematical tools used to answer *What...if* questions: queuing analysis. The second tool, statistical analysis, augments queuing analysis and is discussed further in Chapter 6. This section first reviews well-known queuing principles, known as *operational laws* and then describes challenges that make it difficult for them to be used in shared, distributed systems.

3.3.1 Overview of queuing analysis

The goal of queuing analysis is to compute performance metrics of interest (e.g., throughput, response time, utilization, etc.) for a system under a given workload. The "queuing" term refers to the particular way this analysis works. The system is represented as a network of queues through which requests flow. Figure 2.2 in Chapter 2 illustrated a simple network of two *service centers*, a CPU and a disk. Each service center has a *queue* (analogous to the queue of patients waiting to see a doctor) and a *server* (analogous to the doctor doing the actual examinations).

There are two fundamental actions required to define a queuing network. First, one must define its structure. *Structural behavior* is a term we use to describe how the service centers are linked to one another. Section 3.4 describes how system designers can specify structural behavior expectations. The second action required to define a queuing network is specifying its *performance behavior*. This usually involves specifying the characteristics (e.g., arrival rate) of requests that enter a service center and the characteristics of the server (e.g., how fast it can process a request of a given size — this is also known as service demand). Section 3.4 describes how most performance behavior can be automatically extracted using techniques such as direct measurement, simulation or analytical techniques.

Once the queuing network has been defined, it can be used to compute performance metrics of interest. In general, both asymptotic bounds and exact values can be calculated. Asymptotic bounds are quick to calculate and are usually sufficient for many *What...if* questions.

Notations and basic operational laws

This subsection overviews common operational laws and provides the basic notation used throughout this dissertation. The notation used is most consistent with the notation from Chapter 3 of Lazowska et al. [1984].

Assume we observe a service center for a time T and during that time, A_c requests arrive, C_c requests leave, and N_c is the average number of requests in the center, all from client c. Assume each client has a think time Z_c. Then we have the following definitions (Table 3.1 summarizes the notation used):

Definition 3.3.1 (*arrival rate*)**.** The arrival rate λ_c is defined as $\frac{A_c}{T}$.

Definition 3.3.2 (*throughput*)**.** The client throughput X_c is defined as $\frac{C_c}{T}$.

Definition 3.3.3 (*utilization*)**.** For a system made of a single service center k (e.g., a disk), if B_k is the measured length of time the service centers is busy, then the utilization U_k of that center is defined as $\frac{B_k}{T}$.

Definition 3.3.4 (*service time*)**.** The average service time for client c on service center k is $S_{c,k}$ and is defined as $\frac{B_{c,k}}{C_c}$.

Definition 3.3.5 (*response time*)**.** For any service center k in the system, let $W_{c,k}$ be the time requests from client c spend in the service center's queue. Then, the average response time $R_{c,k}$ for a request is defined as $W_{c,k} + S_{c,k}$.

Definition 3.3.6 (*visit count*)**.** The visit count from requests from client c on a service center k is $V_{c,k}$, the ratio of the number of request departures from a service center (e.g., disk) to the number of request departures from the whole system (e.g., a database that uses a disk). For example, if one database transaction requires five disk accesses to complete, then $V_{c,DISK}$ is 5.

Definition 3.3.7 (*demand*)**.** The service demand $D_{c,k}$ at service center k is defined as $V_{c,k}S_{c,k}$.

From these basic definitions we have the following theorems. The interested reader is advised to consult Chapter 3 of Lazowska et al. [1984] for proofs. We present sketches of derivations here:

$D_{c,k}$	Demand of a request from client c on resource k.
$D_{c,max}$	Largest demand client c places on any resource.
D_c	Sum of demands on all resources client c's request uses.
$S_{c,k}$	Service time for client c on resource k.
$R_{c,k}$	Response time for client c on resource k.
R_c	Response time for client c.
X_c	Throughput for client c.
N_c	Number of requests a client has outstanding.
N_c^*	Threshold for determining if client c's load is low or high.
Z_c	Client think time.

Table 3.1. **Relevant notation in queuing analysis.**

Theorem 3.3.8. *The utilization theorem states that $U_{c,k} = X_{c,k}S_{c,k}$. This follows from the definition utilization 3.3.3, throughput 3.3.2 and service time 3.3.4.*

Theorem 3.3.9. *Little's law states that $N_c = X_c(R_c + Z_c)$. The derivation of this law is subtle and we'll not attempt to derive it here. Notice that the utilization theorem 3.3.8 is a special case of Little's law, where the system under consideration is a single server, with no queue. The utilization $U_{c,k}$ (a number between 0 and 1) can be alternatively viewed as the average number of customers in the server. It follows from Little's law that $R_c = \frac{N_c}{X_c} - Z_c$.*

Theorem 3.3.10. *The forced flow theorem states that the throughput $X_{c,k}$ of a service center k equals $V_{c,k}X_c$. Intuitively, the throughput of a service center (e.g., disk) equals the throughput of the whole system (e.g., database) multiplied by the number of times each database transaction visits the disk.*

Observation 3.3.11. For open-loop workloads, the flow balance observation states that $A_c = C_c$, therefore $\lambda_c = X_c$ in practice.

Asymptotic bounds and exact solutions

The most computation-efficient way to answer several $\underline{What}$...$\underline{if}$ questions is to provide asymptotic bounds on throughput and response time. Asymptotic, in this context, means that either the workload "intensity" is low or is high. A low intensity workload is one where, on average, no requests spend time queueing. A high intensity workload is one where a service center becomes saturated. At that point, the queue size of that service center grows continuously. Often, these bounds are good enough; if more exact calculations are needed, however, other techniques can be used. Below we discuss how the basic theorems described in Section 3.3.1 lead to reasonable performance bounds.

Open-loop workloads: Bounds are computed differently depending on the workload type (open or closed-loop, as first discussed in Section 3.1. For open-loop workloads, the throughput and response time bounds are a function of the request arrival rate λ_c. The function is straightforward for throughput ($X_c = \lambda_c$). The maximum obtainable throughput happens when one of the service centers becomes saturated. The center that has the largest service demand is, by definition, the *bottleneck* center because it saturates first. If we denote the demand of the bottleneck center by $D_{c,max}$, then we have the following relationships:

$$\begin{aligned} U_{c,max}(\lambda_c) &= \lambda_c D_{c,max} \leq 1 \\ &\Rightarrow \lambda_{c,max} \leq \frac{1}{D_{c,max}} \end{aligned} \tag{3.1}$$

Response time R_c, in the best case, equals the sum of the service demands on all service centers k. That sum we denote by D_c. Of course, for requests that are processed in parallel by two or more service centers, only the demand from one of the service centers is part of this sum. The other service centers are ignored for the purpose of computing performance bounds. It is theoretically impossible to provide a worst-case bound on response time, however. Intuitively, that is because the workload can, in theory, be infinitely bursty. However, in practice our monitoring infrastructure, described in Chapter 4, observes the smallest, largest and average bursts in the system. Based on those past observations, it can bound the response time of the workload (this assumes that history is a good indicator of future behavior).

Closed-loop workloads: For closed-loop workloads, the bounds are a function of the number of customers in the system N_c. This is a subtle, but important difference from the open-loop case. In general, the maximum throughput is derived similarly to the case for the open-loop derivation:

$$\begin{aligned} U_{c,max}(N_c) &= X_c D_{c,max} \leq 1 \\ &\Rightarrow X_c \leq \frac{1}{D_{c,max}} \end{aligned} \tag{3.2}$$

The maximum throughput intuitively happens when the request *pipeline* in the system is full, because N_c is large. When N_c is small, indicating a lighter load, each request spends at least D_c units of time in the system, hence the throughput in that case is:

$$\frac{N_c}{N_c D_c + Z_c} \leq X_c \leq \frac{N_c}{D_c + Z_c} \tag{3.3}$$

The above equation has a *pessimistic* and an *optimistic* part to it. In the pessimistic case, each of the N_c requests in the system queues up behind all the others and spends $(N_c - 1)D_c + D_c$ time units in the system, plus Z_c time thinking. Hence, the throughput of each request is $\frac{1}{N_c D_c + Z_c}$ and the client's throughput is $\frac{N_c}{N_c D_c + Z_c}$. In the optimistic case, each of the N_c requests does not spend any time in queues at all. In this case, each of the N_c requests in the system spends D_c time units in the system, plus Z_c time thinking. Hence, the throughput of each request is $\frac{1}{D_c + Z_c}$ and the client's throughput is $\frac{N_c}{D_c + Z_c}$.

The threshold N_c^* for determining if the load is light (so that Equation 3.3 applies) or high (so that Equation 3.2 applies) is $N_c^* = (D_c + Z_c)/D_{c,max}$, where N_c^* can be thought of as the minimum number of requests required to keep the request pipeline full. For example, N_c^* would be $\frac{8+2}{8} = 1.25$ for the queuing network shown in Figure 2.2, if client think time was 0.

Using Little's law 3.3.9 the bounds for response time become

$$max(D_c, N_c D_{c,max} - Z_c) \leq R_c(N) \leq N_c D_c \tag{3.4}$$

Figure 3.3 illustrates these bounds graphically. From these graphs, it

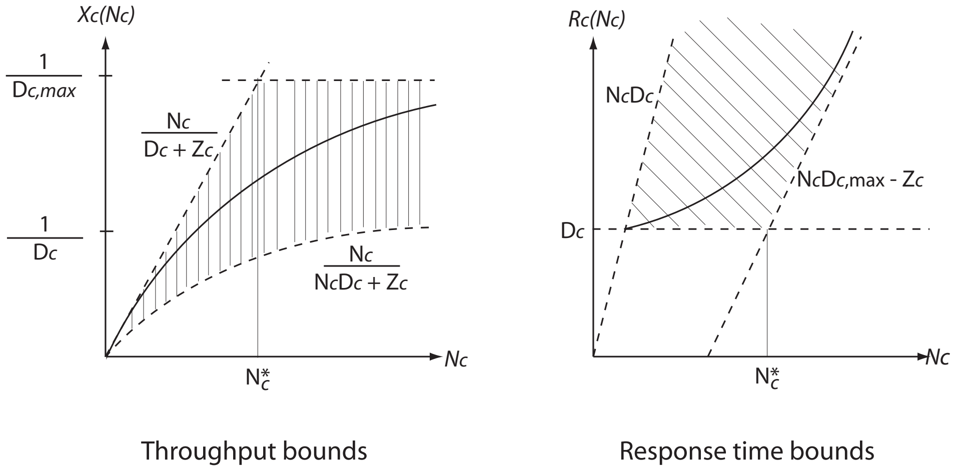


Figure 3.3. **Asymptotic bounds on performance.** This way of visualizing the formulae has been borrowed from Chapter 5 of Lazowska et al. [1984].

becomes clear that upgrading any other resource other than the bottleneck resource provides only modest performance improvements. Upgrading the bottleneck resource changes $D_{c,max}$ and thus leads to larger improvements in performance.

Whenever bounds are not sufficient, exact solutions are also possible to compute. A technique known as mean value analysis, or MVA, is often used (e.g., see Chapters 6 and 7 of Lazowska et al. [1984]). MVA uses an iterative approach to simultaneously solve a set of three equations which follow from first principles:

$$X_c(N_c) = \frac{N_c}{Z_c + \sum_{k=1}^{K} R_{c,k}(N_c)} \tag{3.5}$$

$$Q_{c,k}(N_c) = X_c(N_c) R_{c,k}(N_c) \tag{3.6}$$

where $Q_{c,k}$ is the queue length at center k from client c.

$$R_{c,k}(N_c) = D_{c,k}(1 + A_{c,k}(N_c)) \tag{3.7}$$

Little's Law	$N_c = X_c(R_c + Z_c)$
Forced flow theorem	$X_{c,k} = V_{c,k}X_c$
X_c bound (open-loop)	$\lambda_c = X_c \leq \frac{1}{D_{c,max}}$
X_c bound (closed-loop)	$\frac{N_c}{N_c D_c + Z_c} \leq X_c \leq min(\frac{N_c}{D_c+Z_c}, \frac{1}{D_{c,max}})$
R_c bound (open-loop)	$D_c \leq R_c(N_c)$
R_c bound (closed-loop)	$max(D_c, N_c D_{c,max} - Z_c \leq R_c(N_c) \leq N_c D_c$

Table 3.2. **Key operational laws.**

where $A_{c,k}$ is the queue length at center k seen by an arriving request from client c.

The key to solving this set of equations is the relationship between $A_{c,k}$ and $Q_{c,k}$, where $A_{c,k}(N_c) = Q_{c,k}(N_c - 1)$. Lazowska et al. [1984] describes the method by which this set of equations is solved and also provide a more efficient approximation algorithm for doing so. An off-the-shelf tool (e.g., the one provided by Gunther [2005]) can be used to solve this set of equations.

Table 3.2 summarizes the basic relationships mentioned in this section.

3.3.2 Wrap up on operational laws

In general, operational laws are simple and make few assumptions about workload characteristics. Each of the metrics, such as N_c, D_c, etc., can represent an average value, or a quantile value, e.g., the 90^{th} quantile of resource demand. Hence, predictions can be made for the "tail" of the distribution of each of these metrics.

The simplicity of the operational laws is also their main drawback. The bounds on performance can sometimes be loose, depending on service center implementations (e.g., scheduling policies). We do not have enough experience to show that more advanced, specialized models are more practical, however. For example, the use of queuing analysis that builds upon exponential distribution assumptions (e.g., $M/M/m$) is too specialized to make assumptions that may hold for some (e.g., web server frontend), but not all, parts of a system (e.g., backend DB and storage system). Hence, for the two systems our case studies are based on (a storage system and a database system), operational laws were the most straightforward laws to use.

This dissertation needs the operational laws and concrete models to build on, but the issues it addresses are relevant irrespective of the underlying mathematical framework. Hence, for operational laws, or more sophisticated queuing models (as well as for statistical models that we'll discuss in Chapter 6), several challenges make their use impractical in real systems.

3.3.3 Challenges and practical considerations

There are several challenges when using any of the mathematical models described in Section 3.3. Most of these challenges have in common the lack of system support for the models. Because, traditionally, models are built separately from the system (i.e., they are not built-in), system designers usually do not spend much time thinking about how to design systems to help the models work better.

Aggregate measurements without context

As first overviewed in Section 2.2.1, mathematical models need good parameter values to be effective. In the case of distributed systems with multiple clients/workloads, measuring the various performance behavior metrics (such as visit counts $V_{c,k}$, demands $D_{c,k}$, etc.) is very difficult. For example, performance counters measure an aggregate D_{CPU} at the database, but cannot provide fine-grained measurements per-client (i.e., cannot measure $D_{x,CPU}$). Figure 3.4 illustrates another problem with performance counters in a distributed system. It is impossible for the storage-node in the figure to determine which storage requests relate to which original DB transactions. The causality and context is not preserved through the system queues. So, in practice, even with fine-grained per-client measurements, one could measure $D_{x,CPU}$ at the database and $D_{y,DISK}$ at the storage-node, but one cannot know for sure if $x = y$. Chapter 4 expands on this fundamental issue and provides a solution.

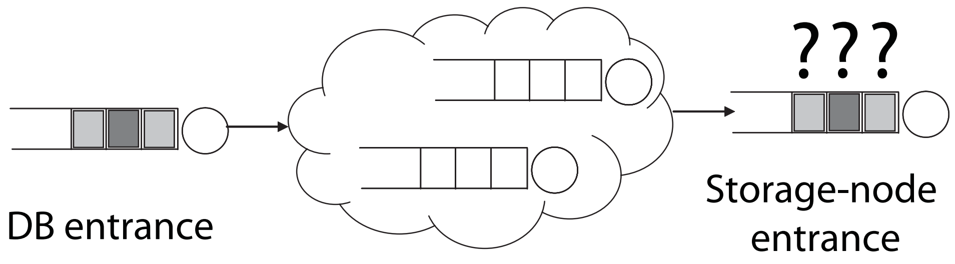


Figure 3.4. **What is a request?.** A database transaction propagates through various system queues and can create multiple sub-requests. By the time these sub-requests reach the storage subsystem the contextual information is lost.

Unforeseen behavior

As hinted in Section 2.2.1, models become obsolete if they do not evolve. Indeed, we observed this to be the case with performance models in systems that we considered to be white-box, since we designed and built them from scratch. Unforeseen behavior can result from 1) unforeseen workload characteristics or system configuration characteristics, 2) unforeseen interaction among service centers, and 3) administrator misconfiguration of the system.

In practice, (1) means that all predictions of performance metrics, such as throughput and response time, will have to be associated with a confidence value that indicates how believable those predictions are in practice. (2) means that the structure of the queuing network will have to be continuously validated. While it is reasonable for system designers to specify a good-enough starting structure, in practice they cannot know all service centers a request flows through. Hence, new service centers will need to be discovered in the field. (3) means that, even in face of good system and model design, the human can be the weakest link and misconfigure the system (and then later attempt to backtrack and locate the source of misconfiguration). Thus, cases when there is a mismatch between model predictions and the system will need to be identified. Chapter 6 introduces an approach for handling this fundamental issue.

Lack of separability of analysis

In shared systems (more than one workload sharing the resources), operational laws assume a basic level of performance insulation in the system for meaningful predictions. Performance insulation is a mechanism that, among other benefits, ensures that workloads do not misbehave by utilizing resources allocated to other workloads. For the purpose of this thesis, the main reason why the systems that have performance insulation help the models is a scalability issue. When adding workload W_{n+1} to the system, it is desirable that the queuing network does not have to be reevaluated for the existing W_n workloads. A secondary reason is system performance. Figure 3.5 illustrates a case where two streaming workloads propagate through the queues of a system, and, by the time they reach the storage system, they may be interleaved in a number of ways. The resulting performance varies drastically depending on the way requests interleave. Hence, although complex models can be built to predict the variance, it is questionable if the predictions are meaningful at all. After all, what is a prediction of the form $30\,\mathrm{MB/s}\pm 30$ useful for?

Although performance insulation for the CPU and network resources is usually straightforward to do (utilizing well-known scheduling techniques), it is much harder to enforce for storage systems. Storage systems have the property that the cost of a "context switch" between workloads can be very high (several milliseconds), and the performance of workloads can vary by two or more orders of magnitude (due to cached, disk-bound streaming and disk-bound random-access workloads). Our work in this area has found that it is necessary to combine smart cache partitioning, request scheduling, deep prefetching and smart write-back algorithms to guarantee performance insulation for disks [Wachs et al., 2007]. Any of these in isolation have proven insufficient in addressing the problem. Chapter 5 illustrates cases where, without separability of analysis, the predictions would be hard to obtain and the models would be forced to make worst-case predictions. Chapter 5 also illustrates how the models themselves can be used to guide the mechanisms needed to ensure performance insulation in a storage system.

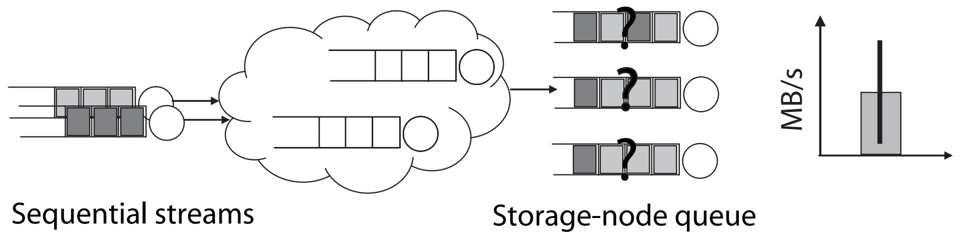


Figure 3.5. **Inherent system unpredictability.** If requests are allowed to inter-mix in uncontrolled ways, the resulting performance variance can be very high (for a disk that is the case independently of the scheduling algorithm used). This figure shows how two sequential workloads may appear as non-sequential by the time they reach the storage-node queue.

3.4 White-box modeling: expectation-based models

This section describes a general design of expectation-based models in systems. Chapter 5 describes concrete implementations through case studies. Expectation-based models are a crucial part of the modeling architecture shown in Figure 3.2. They are considered to be "white-box" since the system designer is expected to create them. The assumption is that the system designer can bootstrap the modeling process by providing reasonable expectations for the system under consideration.

Creating expectation-based models involves several steps. First, the designer must define *structural expectations*. Structural expectations enumerate the ways requests are expected to flow in a system. Using the queuing theory language developed in Section 3.3, this means that structural expectations define the location of the service centers and how they are connected to one another. Structural behavior can be expressed with a directional graph (can be cyclic), in which nodes represent individual service centers. An edge from service center SVC_1 to service center SVC_2 means that a request departing from service center SVC_1 will enter service center SVC_2 next. Edges can contain a rich set of attributes, such as the probability that a request will go to service center SVC_2, the average time it takes to go from one service center to another, etc.

Second, the designer must define *performance expectations*. Performance

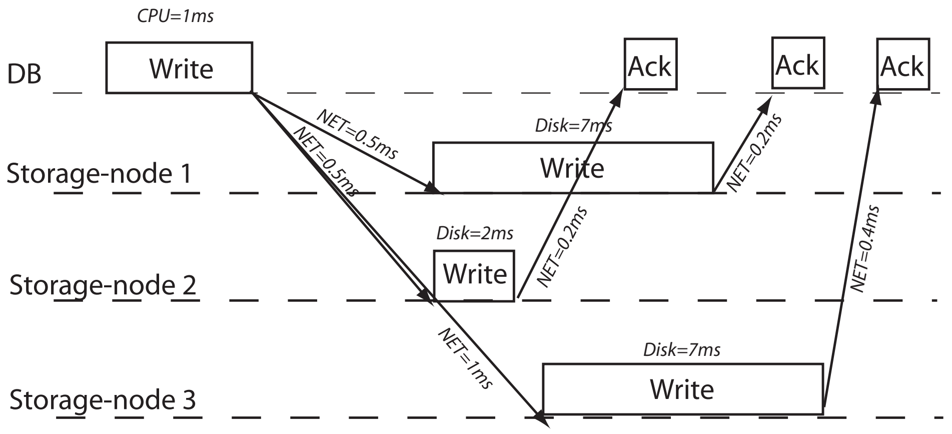


Figure 3.6. **Example structural and performance expectations.** A simple structural expectation is that, when using 3-way replication, 3 storage-nodes should be contacted on a write. Performance expectations can usually be automatically extracted once the structure of the queueing network is known.

expectations are functions that relate workload characteristics (such as arrival rate, request size, read:write ratio, etc.) and service center characteristics (such as queue size, service time, etc.) to eventual performance metrics, such as throughput and response time. The designer must create, for each service center, low-level *What...if* models that take any hypothetical workload and service center characteristics and predict any performance metric of interest. The *What...if* models are usually templates that describe the relationship between system-workload characteristics and performance. Specific values for the system under consideration can be filled in automatically in the templates.

Figure 3.6 illustrates one simple system expectation in a hypothetical system consisting of a database and a storage system. The 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 current CPU and network characteristics can be automatically discovered, and performance expectations can then be automatically created about CPU and network consumption for the write request (e.g., encoding should use 0.02 ms and it should take 0.5 ms to send the data to the storage-nodes). A structural expectation for the DB is that a transaction should spend CPU time in the query evaluator and optimizer, then access the buffer cache, and then access the underlying storage system. The exact CPU time and cache miss rate for a transaction type can be automatically discovered and can be part of the transaction profile.

Evaluating how difficult it is to create expectation-based models can only be done through case studies, and we present ours in Chapter 5. This thesis makes the case that part of designing any new system component should be defining the *What...if* interfaces to that component. Thus, we are asking that the system designer put more effort into creating good models up-front. Out experience supports that doing so helps with designing a more predictable system. Our case studies show that it is reasonable to expect the designer to augment key service centers with a *What...if* model. For example, the designer of the buffer cache manager inside a database, operating system, or storage system should model the behavior of the cache as a function of the cache size and workload. The model should be part of the cache manager and can export interfaces like "*What* would be the new cache hit rate for workload c, *if* we double the amount of cache size?" or "*What* would be the new cache hit rate for workload c, *if* the eviction algorithm changes from LRU to MRU?". This model could be based on simulation or an analytical formula (depending on the complexity of the cache manager, sometimes it is not possible to have a closed form analytical solution). Part of the contribution of Chapter 5 is to evaluate the usefulness of a range of techniques (from direct measurements to analytical formulae to simulation) that designers of these white-box models have available in their toolbox.

4 Measuring with context in distributed, shared systems

Tuning a distributed system requires good measurement infrastructures and tools. Current systems provide little assistance. Most provide insights only in the form of hundreds of performance counters that the system's administrator can try to interpret, analyze, and filter to diagnose performance problems. For example, most modern databases and operating systems come loaded with an array of performance counters [IBM Corporation, 2004; Microsoft, 2005; Oracle Corporation, 2004]. Performance counters, however plentiful, are inadequate for two primary reasons. First, in shared environments, aggregate performance counters do not differentiate between different workloads in the system and give only combined workload measurements. If the administrator is attempting to diagnose an issue with one of several workloads, aggregate counters are not helpful. Second, in a distributed system, performance counters cannot be easily correlated to high-level user observations about throughput and latency. The lack of causality and request flow information makes combining information across components difficult.

Stardust is the infrastructure that I have created for collecting and querying *end-to-end traces* in a distributed system. Trace records are logged for each step of a request, from when a request enters the system to when it is complete, including communication across distributed components. The trace records are stored in databases, and queries can be used to extract per-request flow graphs that show how requests flow through service centers, latencies, and resource demands.

This chapter is self-contained. It describes related work on measurement infrastructures as well as the design, implementation and evaluation of Stardust. It discusses the challenges faced in building this infrastructure and the opportunities that arise from having it in the system. The challenges included reducing the overhead the infrastructure places on foreground workloads, reducing the amount of spare resources needed to collect and process the traces generated, and reducing the difficulty of trace analysis. The opportunities include concrete tuning problems we are able to solve using Stardust and other tuning problems we have not yet addressed, but we believe are solvable using this infrastructure. We also discuss the limitations of end-to-end tracing, as a performance monitoring tool, and the kinds of problems it will not solve.

4.1 Background and related work

Performance monitoring is useful throughout a system's lifetime. In the initial system implementation stages, it can help developers understand inefficiencies in the design or implementation that result in poor performance. In the system deployment stages, it can help administrators identify bottlenecks in the system, predict future bottlenecks, determine useful upgrades, and plan for growth. When users complain, administrators can use the observations from periods of poor performance to help understand the source of the problem. Observations from real systems even drive research into building better system models and capacity planning techniques (much related work is described in [Lazowska et al., 1984; Menasce and Almeida, 1998]).

4.1.1 Three "simple" administrator questions

We use three concrete scenarios to illustrate how current measurement infrastructures are inadequate, even for simple administration tasks.

More RAM or faster disks?: When money is available to upgrade hardware, administrators must decide how to spend their limited budget. "Should I buy more RAM (for caching) or faster disks?" is a simple example choice, and even it is not straightforward to answer. The value of increased cache space is access pattern dependent and, worse, workload mixing can muddy the picture of what limits any given application's performance. When using performance counters, in particular, it is unclear which counters should be consulted to answer this question. In the systems that we have observed, none of the performance counters are adequate. For example, consider the counter that keeps track of buffer cache hits and misses. Even if that counter indicates that client A's workload never hits in the cache, it does not mean that adding more RAM for the cache would not help—a workload, for example, that scans a 500 MB object repeatedly, but has been allocated only 499 MB of RAM space (and thus never hits in buffer cache with an LRU replacement policy), would benefit greatly from a 1 MB increase in RAM space. The workload would see a 100% hit rate. Similarly, consider a counter that keeps track of the average disk queue size. A large value does not necessarily mean that faster disks would be better than more RAM.

Decisions are even more difficult for a shared infrastructure supporting multiple clients. For example, one client may benefit most from a RAM upgrade while another would benefit more from faster disks. Aggregate counters show overall averages, rather than per-workload information, so this information is hidden. If one client is more important than the other, going with an average-case choice is not appropriate. Interactions among workloads can also create situations where changing the amount of buffer cache for one causes a ripple effect on the performance of the others (e.g., the sequence of disk accesses changes). For example, we have seen cases where improving the cache hit rate for one client also provides a bigger than expected efficiency boost for another by reducing disk-level interference.

Where does time go?: When a particular set of requests are slower than expected, an administrator often needs to know why (and then implement a fix). "Where are requests from client A spending most of their time in the system?" is an example administrator question. This question is representative of situations in distributed environments with multiple service centers (e.g., a request passing through a web server that queries a database that retrieves data from a storage system). The administrator may want to know which service center accounts for what fraction of the average request's latency. This question can be answered by creating a request's flow graph, which represents the flow and timing of the request as it moves from service center to service center in the distributed system. Current performance counters do not help with this. One needs to know how requests moved through the system (counters only measure metrics locally) and how long they spent at each component (aggregate counters give an average measurement for all requests).

Why is the client complaining?: When users complain about their performance, administrators must figure out what happened, differentiate between transient problems (which can often be ignored) and recurring ones (which should be fixed), and decide what to do. "Why was the application's performance unacceptable at 2pm?" is an example starting point. At a minimum, the system will need to retain performance observations for a period of time so that administrators can go back and check. But, looking at performance counters, like CPU load, disk I/O rate and buffer cache hits/miss counts is rarely sufficient for root-cause analysis, for the reasons explained above. As well, the administrator (and/or an internal tuning module that makes use of mathematical models) may need to know the specific sequence of requests that led to the poor performance the client experienced and how those requests moved through the distributed system.

4.1.2 Traditional performance measurements

Traditional performance monitoring consists of pre-defined counters, such as "number of requests completed" and "average queue length". Such performance instrumentation is common in single-node systems [Cantrill et al., 2004; IBM Corporation, 2004; Microsoft, 2005; Oracle Corporation, 2004], or in each node of a distributed system. There are some monitoring infrastructures designed for distributed systems [Anderson and Patterson, 1997; Massie et al., 2004], but they focus on aggregate resource consumption statistics rather than per-client or per-request information. Such aggregate performance monitors provide little assistance with problems like those discussed above.

A second problem with performance counters is that their values cannot be easily correlated to high-level user observations about throughput and latency in a distributed system. In a busy web server with a backend database, is the high CPU utilization at the web server, is the high I/O rate at the database server, both, or are neither of them responsible for the high latency clients are noticing?

Detailed process accounting systems can address some of the shortcomings of performance counters. For example, Bouhana [1996] describes an accounting system that keeps per-user, per-resource demands in order to later bill the user appropriately. Existing accounting systems, however, are limited to centralized systems and simple batch processing systems. In most distributed systems, where request processing may involve applications running on several servers, existing approaches do not extend well. This is because such accounting systems do not provide support for causally linking work across servers. Among other things, Stardust can be used as the measurement component of an accounting system for distributed systems.

System logs are often used to capture basic workload or utilization information. The Windows operating system, for example, offers the ability to log performance counters and request/reply calls for later analysis [Microsoft, 2005]. The most common use of such logging is to retain a history of HTTP calls or SQL database queries. Such logging is useful for workload analysis and even trace-based simulation but, as commonly configured, provides little help with performance analysis. Currently, there is little support for integrated tools that allow the administrator to correlate log entries with system performance information. Stardust builds on fine-grained logging/tracing of activity across a distributed system, enabling both traditional information extraction and detailed performance analysis.

4.1.3 Towards end-to-end monitoring

Researchers have started exploring the use of *end-to-end tracing* of requests in a distributed system to better inform diagnosis and tuning questions. End-to-end tracing refers to collection, storage, and correlation of activity

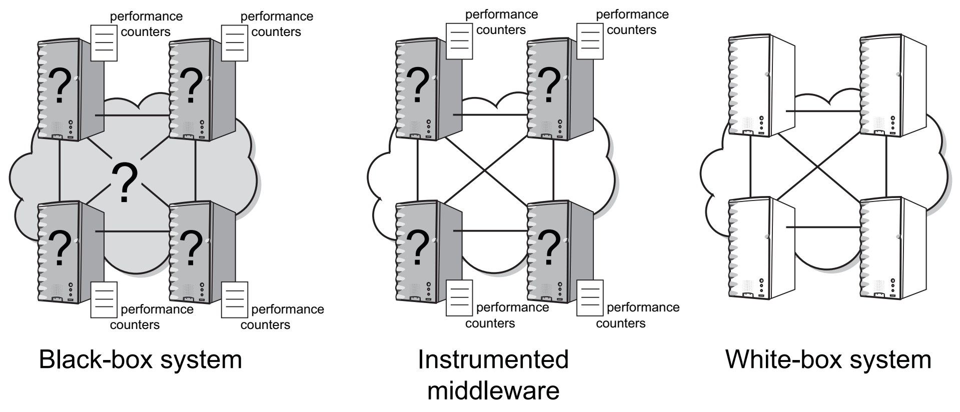


Figure 4.1. **The instrumentation framework depends on the system under consideration.** "Black-box" systems are systems whose components work together through well-defined interfaces but are closed-source. "Instrumented middleware systems" consist of black box components interconnected by a well-known middleware that provides resource multiplexing, communication management and accounting. "White-box" systems are systems whose internals are completely known to those building the instrumentation framework, either because the system is being built by those same people or because its source code is available. Such systems offer the opportunity to have the necessary instrumentation built-in from the start.

records that are generated by a single request from the moment it enters the first node in the distributed system until it leaves the system. Research in this area has focused on three system types (illustrated in Figure 4.1): black-box, middleware-based, and white-box systems.

"Black-box" systems are constructed of components that work together through well-defined interfaces but are closed-source. Although only high-level information can be determined in such an environment, researchers are developing approaches to determine causal paths of requests and the relationship between individual component performance and overall performance. For example, Aguilera et al. [2003] showed that coarse-grained end-to-end traces can be extracted via passive network monitoring without requiring any legacy application changes. Further, they showed that such tracing is sufficient to identify black-box components that are bottlenecks and guide an administrator's focus to them. Cohen et al. [2004] explored the efficacy of statistically correlating the values of per-black-box performance counters with high-level user observations, such as throughput or latency, to identify relationships that can guide diagnosis. These approaches improve on traditional approaches and can work when otherwise uninstrumented third-party applications are utilized.

"Instrumented middleware systems" are often deployed as a set of black box components running on top of middleware, such as J2EE or .NET, that provides resource multiplexing and management [Microsoft, 2005; Sun Microsystems, 2005]. Systems such as Pinpoint tag requests as they flow through the J2EE middleware and correlate middleware measurements with application-level throughput and latency [Chen et al., 2002]. Xaffire Inc. [2005] and Quest Software [2005] are commercial products that use similar tagging techniques. Instrumented middleware system provide deeper insight than pure black box and traditional approaches, but they still leave intra-component resource usage and delay sources unclear.

We use the term "white-box" system to describe a system whose internals can be modified and understood, either because the system is being built from the start or because its source code is available. Such systems offer the opportunity to have fine-grain instrumentation built in. For example, Magpie is a research system that collects traces at different points in a system and reconstructs causal paths from those traces [Barham et al., 2004; Isaacs et al., 2004]. Magpie relies on programmers to place instrumentation points in the appropriate system modules. In return, it offers critical path analysis and per-workload, per-resource monitoring. ETE is a similar system that is used to measure end-to-end response times [Hellerstein et al., 1999]. Hrischuk et al. [1995] define a specialized language to describe end-to-end traces and measure per-workload and per-resource demand as well as request response times. Stardust and this dissertation build on these ideas by developing an efficient querying framework for traces, reporting experiences from use in a cluster-based storage system, and performing feasibility studies under various system loads.

4.2 Stardust's design

Stardust's design was motivated by several goals:

Resource usage accounting: The instrumentation framework should provide accurate aggregate and per-client resource accounting. Aggregate accounting is sufficient when the administrator is concerned with the load on a resource. But, per-client accounting is needed to understand how individual clients contribute to that load. In a distributed system, a request may propagate through several service-centers, requiring per-client accounting of all of them. Resources of interest include the CPU, buffer cache, network and disks. There are other non-hardware resources too, for example a lock service can be thought as a software resource. For simplicity, we will focus on the hardware resources. Using queuing analysis notation, Stardust should give us the metrics represented in Table 3.1.

Request latency accounting: The instrumentation framework should provide per-client request latency information that records where each request spends its time as it flows through the system. Different clients may have different latency profiles. A client whose requests hit in the buffer cache, for example, will have a different profile than a client whose requests miss. A request may span servers in a distributed system, so care must be taken to causally link together sub-requests in each server that belong to the same original request.

Long-term statistics maintenance: The instrumentation framework should be able to maintain statistics for a long time, to allow for various correlations to be made. For example, as Chapter 6 will describe, many machine learning tools require system-level statistics to be exposed and maintained over time.

Instrumentation framework efficiency: The instrumentation framework should interfere minimally with the workloads running in the system. We envision the framework to be monitoring the system at all times; hence, overheads should be minimal. In addition, the programming burden for implementing the framework inside a system should be low.

Querying efficiency: The instrumentation framework should provide a flexible query interface. In a distributed storage system with hundreds of nodes and clients, it is important to have a versatile way to query the wealth of information generated.

4.2.1 Activity tracking and querying

Stardust tracks every client request along its execution path. Stardust retains activity records, such as buffer cache reference records, I/O records, and network transmit/receive records. The sequence of records allows tracking of a request as it moves in the system from one service center, through the network, to another service center, and back. Of course, the definition of "request" is specific to each service center. For example, from a client's perspective, a request may be a single SQL query. From a storage-node's perspective, a request may be a read or a write request (a SQL query may be translated to multiple storage-node read/write sub-requests).

An activity record is a structure with a timestamp, a type, and type-specific values. Figure 4.2 shows an example activity record. Each activity record contains an automatically-generated header comprised of a timestamp, a breadcrumb, a kernel-level process ID, and a user-level thread ID. The timestamp is a unique value generated by the CPU cycle counter that permits accurate timing measurements of requests. The breadcrumb is a request identifier (e.g., a 64-bit number) that permits records associated with a given request to be causally linked across service centers. Activity records are posted (i.e., trace records are generated) at strategic locations in the code so that the demand on a resource is captured. These locations are often the point of arrival to and departure from a service center. For example, the disk activity record is posted both when an I/O request is sent to disk and when the I/O request completion is reported. Both postings contain the same breadcrumb, because they belong to the same client request, and so can be associated in post-processing. Records are posted on the critical path; but, as this chapter's evaluation shows, such posting causes minimal impact on foreground performance.

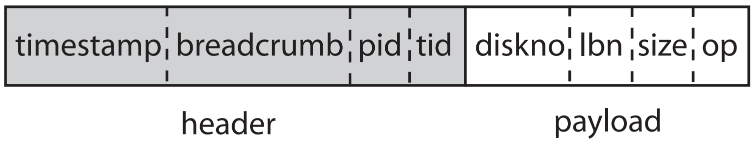


Figure 4.2. **Example activity record.** Each activity record has a common header and a payload of type-specific values. The payload for the disk request activity record shown includes the disk ID, logical block number (LBN), size of the I/O in bytes, and operation type.

Each server in an instrumented system runs a single Stardust client library. A Stardust client library is responsible for presenting any process running on that server with APIs for posting and querying activity records. For querying flexibility, Stardust records are stored in relational databases (Activity DBs). Activity records posted to a Stardust instance are buffered and the buffers are periodically sent to an Activity DB. Activity DBs are designed to run on the same infrastructure servers as the rest of the system. The DBs store the records in relational tables and answer queries on those records. Storing activity records in a database allows a flexible querying interface.

Activity DBs are part of the querying infrastructure, and they can be queried using SQL. For example, to get a disk I/O trace for a certain storage-node, one might query the Activity DB that keeps records for that storage-node's disk activity records. Collections of activity records are effectively super-sets of performance counters. Many performance counter values of interest can be computed by querying the Activity DBs.

4.2.2 Resource usage accounting

This section describes how Stardust enables the extraction of the per-workload demand placed on four common storage system resources: CPU, buffer cache, network and disk. When client requests enter an instrumented system, they are tagged with an initial breadcrumb value by the first service center. This breadcrumb value is passed between service centers as the request is serviced.

CPU demand: To allow measurement of per-client CPU demand, $D_{c,CPU}$, a component must post activity records related to thread context switching[1]. Context switching happens both in preemptive systems (when the scheduler decides to run another thread) and non-preemptive systems (when a thread yields). Thus, CPU processing of a request may be suspended and resumed several times (this is known as a *processor sharing* scheduling approach in queuing analysis). By monitoring the context switches and the requests being processed during a thread's run time, Stardust charges a request the exact amount of CPU time it used. That time is the sum of the time any thread spent processing that request and any of its sub-requests (that go to other service centers).

Buffer cache usage: Buffer cache usage for an individual client is captured by posting activity records each time the buffer cache is accessed. Buffer cache usage is defined as the sequence of buffer cache accesses during period T, that include read hits and misses, writes, readaheads and evictions.

Network demand: Network demand for an individual client, $D_{c,NET}$, is captured by posting a *NetworkTransmit* activity record each time data is transmitted from one server to another. These records contain, among other attributes, the number of bytes sent from the source to the destination service center. For any period T, the demand is then the total number of bytes transmitted during that period.

Disk demand: Disk demand for an individual client, $D_{c,DISK}$, is captured by posting a *DiskOp* activity record each time a client request initiates a disk request. A *DiskOp* record denotes the beginning or completion of a disk read or write. The disk demand during any time T is the sum of disk service times for the user's requests.

Measuring delayed demands

There are two important and tricky cases that complicate resource demand accounting. First, whereas read requests are usually synchronous (the user

[1]We have not experimented with event-driven systems, but, at least conceptually, calculating the CPU demand would be similar. An activity record could be posted each time processing of a request resumes.

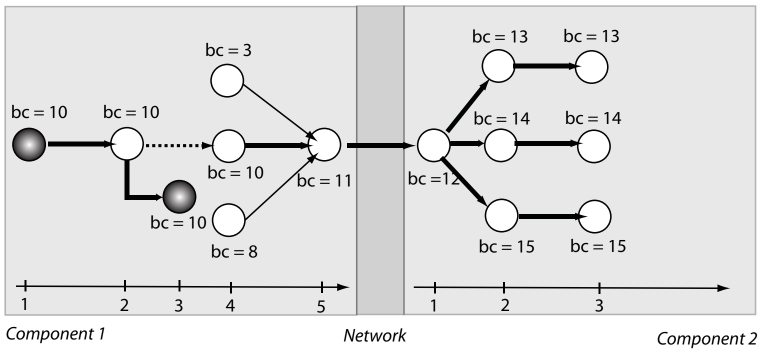


Figure 4.3. **Example causal path created by a generic Write call.** This diagram shows the creation of a causal path in a distributed environment with two components. Nodes in this illustration show instrumentation points in the system, and links show a request propagating through two such points. This diagram also illustrates the case when a Write request is declared complete (component 1, time 3) before it is propagated to the second component and when multiple writes are coalesced (component 1, time 5).

has to wait until the read completes before proceeding), there are asynchronous requests (e.g., writes). A write will often be inserted into a cache and have control returned to the user. The write propagates to other service centers (e.g., disk) at a later time. This is often done to hide the latency of writes and results in significant performance gains. Resource accounting for those requests must occur, however, even after control is returned to the user. In general, this problem arises anytime there is delayed processing of a request in the system.

Second, some requests from different users may be coalesced into a single, larger request to improve performance (e.g., coalescing disk requests). It is important to bill the resource usage of this larger request to the proper original requests.

Figure 4.3 illustrates both problems by illustrating the typical path of a write request. The request arrives in the system (denoted by the first black node) and departs (the second black node) after it has been stored in cache. At a later time, denoted by the dotted arrow (depicting the first problem), the request is coalesced with other requests and sent to the storage-node (depicting the second problem). The storage-node, in turn may split it into sub-requests. For accounting purposes, it is important to capture these cases, especially because writes are frequent in most systems.

Stardust solves the first issue by storing the breadcrumb together with the data in the cache. When the request is later processed, any sub-requests it generates use that breadcrumb, allowing the original request to be properly billed. If that sub-request is coalesced with other sub-requests, the many-to-one relationship is noted (through an explicit "stitch" record), and any resources the larger sub-request subsequently uses are billed to each original request (currently equally proportional).

4.2.3 Request latency accounting

This section describes how Stardust provides per-client request latency information that shows where a request spends its time as it is processed in the system. Each instrumentation point can be considered as a node in a *latency graph*, with links between nodes denoting causal relationships. These links also capture latency information between the instrumentation points. As a simple example, if one instrumentation point was before a disk request and another after the request completed, the link between them would denote the disk response time.

Identifying causal paths in a distributed environment: The processing of requests often generates multiple activity records at different servers in a distributed system. The servers in a distributed system are not expected to have synchronized clocks. It is important to causally link, or *stitch*, activity records together so that the path of the original request can be reconstructed in such environments. On the same server, two records R_1 and R_2 are totally ordered by their timestamp. If R_1 and R_2 have the same breadcrumb, but R_1 happens before R_2 then R_1 is a parent of R_2. On different servers, such ordering is possible only if the records R_1 and R_2 (which are created on their respective servers) are explicitly related through a "stitch" record that contains the breadcrumb of R_1 (or any of its children's sub-requests) and the breadcrumb of R_2 (or any of its parents' requests).

The example path in Figure 4.3 can be used to explain the above rules. Two physical servers are connected through a network. In a distributed storage system, for example, the first server could be a Metadata Server (which keeps information on where data is located) and the second a Storage-Node (which actually stores the data). Nodes in this illustration show activity records. Links between records show nodes that are related because they are associated with the same breadcrumb (i.e., breadcrumb 10). On each server, all records are totally ordered by a local timestamp, denoted by the timeline at the bottom. To show that all of the records on the first server happened before any of the records on the second server, a “stitch” record of the form $(bc = 11, bc = 12)$ is posted just before the request leaves the first server. The stitch record links the last record on the first server (child record of the originating request) to the first record on the other server (parent record for all subsequent records on that server). The reason a new breadcrumb is assigned every time a request moves to a new server is implementation-dependent and is described in Section 4.3.

Request flow graphs can be constructed for a batch of requests, as well as for each individual request. When creating a graph for a batch of requests, the graphs of individual requests are aggregated into a set that is the *union* of the individual request graphs. A main question that the set of graphs data structure can answer is “what is the average demand and/or response time between two instrumentation points A and B?” To answer that question, the latencies between nodes A and B are averaged for each graph in the set that contains those nodes.

4.2.4 Collection efficiency

There is overhead associated with collecting the traces. However, the overhead can be made negligible in most cases. CPU overhead at servers posting events is kept to a minimum by reducing the Stardust client work. The client only has to encode the trace records to network byte order and place them in pre-allocated buffers. When the buffers fill up, records are sent to the

Activity DBs. Thus, we are accepting the possibility of partial trace loss due to server failure in order to reduce tracing overhead.

Network and storage overheads are closely related; both depend on the quantity of trace data generated. Ultimately, the quantity of trace data generated depends on intended usage. Chapter 5 describes several concrete experiences that we have had in solving real problems using such traces. In general, performance problems that can be answered by looking at only resource loads require only per-client, per-resource performance counters. For such problems, it is possible to drastically reduce the amount of traces kept. This is done by transforming the raw traces into per-workload, per-resource demand information, every period T. Section 4.4 quantifies the space reduction from such pruning.

There are, however, several performance tuning problems that require certain types of records to be kept in their raw form (e.g., online performance anomaly detection). Section 4.4 analyzes the feasibility of keeping full traces and the efficiency of pruning them.

4.2.5 Querying efficiency

Activity records are stored in Activity DBs, which use relational tables. Any internal system entity (or external administrator) can use SQL to analyze traces. Each Activity DB contains all the trace records associated with a set of clients and servers. Thus, no query will involve accessing more than one Activity DB. We considered using distributed databases, but opted for simplicity.

Each Activity DB stores activity records in a number of tables, one for each record type. The system is designed such that the Activity DBs do not need to be restarted or recompiled if a component posts new record types. New tables can be created on-the-fly based on an XML description of the new record type.

One can generate any performance counter from the end-to-end activity traces. We call the performance counters generated from the traces *virtual* performance counters, because they are not hard-coded into the code, but are instead generated on the fly using SQL queries on the trace tables. As a simple example, consider the traditional counter that keeps track of the number of hits in a component's buffer cache (e.g., on a Storage-Node component). In Stardust, that counter can be generated from the following SQL query on the table that holds the buffer cache records for that Storage-Node:

> **SELECT** $count(*)$
> **FROM** $STORAGE\_NODE\_BUFFER\_CACHE\_TABLE$
> **WHERE** $optype = BUFFER\_READ\_HIT$

4.3 Stardust's implementation

4.3.1 Brief introduction to Ursa Minor

We have designed and implemented a cluster-based storage system, Ursa Minor, to target research problems in system management. Ursa Minor was designed from a clean slate; hence, we had the opportunity to include the instrumentation in the design of the system from the beginning without the need to retrofit an existing system. This section briefly describes it so that this chapter remains self-contained. Chapter 5 further describes features of Ursa Minor that will be relevant then.

The goals of Ursa Minor are described by Ganger et al. [Ganger et al., 2003], and the architecture and implementation are described by Abd-El-Malek et al. [Abd-El-Malek et al., 2005]. At the core of the architecture is the separation of mechanical functions (servicing client requests) from managerial functions (automating administrative activities). The managerial tier consists of agents and algorithms for automating internal decisions and helping administrators understand the consequences of external ones. The mechanical tier is designed to self-monitor, through Stardust, so as to provide information to the managerial tier. Below we define the main structural components of the system.

Clients: Clients of the system access data. Different data may have different availability, confidentiality and performance goals. Clients make use of the PASIS *protocol family* to read and write data [Goodson et al., 2004; Wylie, 2005]. Currently, clients in our setting use the NFS data access protocol [Sun Microsystems, 1989] to interact with intermediary NFS servers, which in turn use the PASIS protocols to read and write data to the storage-nodes on behalf of the clients.

Storage-nodes: The storage-nodes have CPUs, buffer cache and disks. Storage-nodes are expected to be heterogeneous, as they get upgraded or retired over time and sometimes are purchased from different vendors.

Metadata service: The metadata service (or MDS) is responsible for keeping track of files in the system. It is also responsible for authorizing client access to storage-nodes through the use of capabilities. An access to a file usually is preceded by an access to the MDS to get the metadata for accessing that file. Once the metadata is obtained, the client interacts directly with the storage-nodes to access the data.

Stardust client: A Stardust client (not to be confused with Ursa Minor clients described above) is any service in Ursa Minor that links to the Stardust client library. Currently, all the above services are Stardust clients as well.

Figure 4.4 shows a typical request flow through Ursa Minor. There are several resources used by a request as it flows through the system. First, significant CPU computation is required at both the NFS server and at the storage nodes. The client requires CPU computation to encode (decode) the data into (from) *fragments* that are stored onto N storage-nodes (e.g., when data is striped across N storage-nodes). Part of encoding may be compressing or encrypting the data. The storage nodes require CPU to check the integrity of data blocks, through checksums. Second, buffer cache space is required at the client, metadata service, and storage-nodes to hold frequently accessed data or metadata. Third, network bandwidth is required to transmit requests and responses from the various components in the system. Fourth, disk bandwidth is required at the storage-nodes to process read requests that miss in the cache and write requests.

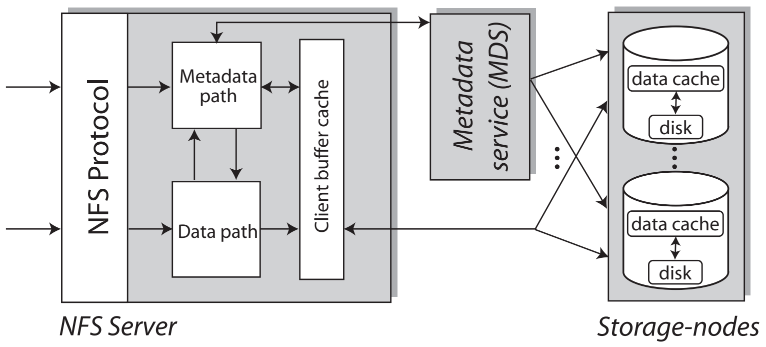


Figure 4.4. **Typical request path through Ursa Minor.** A request from a client enters the storage system at an NFS server. The server consults the metadata service to get the relevant metadata for the client request. Once the metadata is received, any data access for the request is served through the data path. The request may hit in the NFS server cache or miss and have to be serviced from the storage-nodes. The request may be sent to more than one storage-node, for example, when the data is replicated or striped among many nodes. A queuing network representation of this path is deferred to Chapter 5.

4.3.2 Instrumentation points

Table 4.1 shows the records used to measure resource demands. Table 4.2 shows the records used to measure latencies. Some records are used for both. There are approximately 200 instrumentation points in Ursa Minor, which currently has over 250,000 lines of code. Almost all instrumentation points are posted from user-level processes, because most request processing in Ursa Minor is done in user-level threads. The only exceptions are the kernel-level context switch records ($KernelProcessSwitch$), which are posted by the Linux kernel. This was the only modification necessary to the operating system. User-level context switches ($UserThreadSwitch$) are posted from the State Threads library [SourceForge.net, 2006]. Figure 4.5 shows a snippet of code and two posting calls to the Stardust client library, when a request enters and leaves the disk service center.

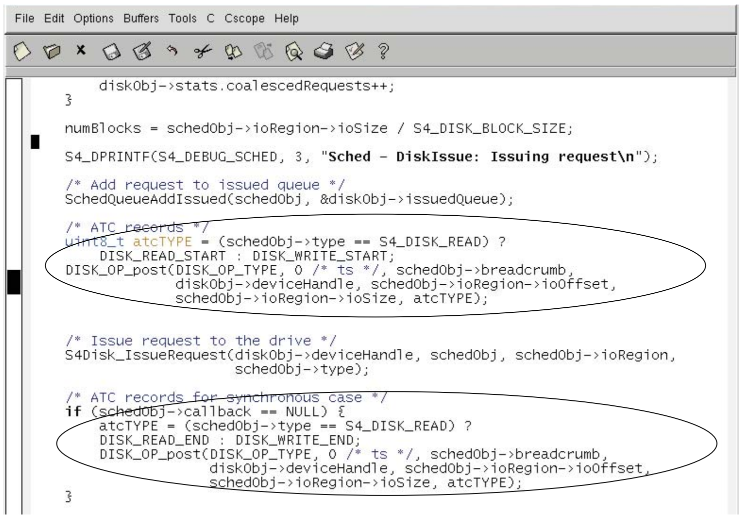
In Ursa Minor, monitoring is performed at all times. Additional record types can be added by programmers through new releases of the system.

	Record Type	Arguments	Description
CPU demand	*UserThreadSwitch*	*oldthread, newthread*	A user-level context switch
	KernelProcessSwitch	*CPU ID, old PID, new PID*	A kernel-level context switch
Buffer cache demand	*BufferReadHit*	*file, offset, size*	Denotes a buffer cache hit
	BufferReadMiss	*file, offset, size*	Denotes a buffer cache miss
	BufferWrite	*file, offset, size*	Marks buffer dirty
	BufferReadAhead	*file, offset, numpages, psize*	Prefetch pages (non-blocking)
	BufferFlush	*file, offset, size*	Flush a dirty page to disk
	BufferEvict	*file, offset, size*	Evict a page from the cache
Network demand	*NetworkTransmit*	*sender, receiver, numbytes*	Monitors network flow
Disk demand	*DiskOp*	*disk ID, LBN, size, operation*	Monitors disk activity

Table 4.1. **Activity records used to measure resource consumption.** *KernelProcessSwitch* records are provided by the Linux kernel (other operating systems, such as Windows, already expose kernel-level context switches [Microsoft, 2005]). The remainder are posted from instrumentation points in user-level processes. Note that there are multiple buffer caches in the system (e.g., at client, metadata service and storage-nodes), hence the buffer cache records are posted at all those levels.

	Record Type	Arguments	Description
NFS service	*NFSCall_type*	*user ID, call args*	Request arrives at the NFS service
	Buffer_type	*buffer args. See Table 4.1*	Request accesses the NFS buffer cache
	NFSReply_type	*reply args*	Request exits from the NFS service
MDS service	*MDSCall_type*	*call args*	Request arrives at the MDS service
	Buffer_type	*buffer args. See Table 4.1*	Request accesses the MDS buffer cache
	MDSReply_type	*reply args*	Request exits from the MDS service
Storage-node	*S-NCall_type*	*call args*	Request arrives at the storage-node
	Buffer_type	*buffer args. See Table 4.1*	Request accesses the storage-node buffer cache
	S-NReply_type	*reply args*	Request exits from the storage-node
	DiskOpCall	*call args*	Request accesses the storage-node's disk
	DiskOpReply	*call args*	Request exits from the storage-node's disk

Table 4.2. **Activity records used to measure request latency.** The above records capture entrance and exit points for key services in the system. NFS calls monitored include most calls specified in the NFS protocol [Sun Microsystems, 1989], of which the most common are : NFS_GETATTR, NFS_SETATTR, NFS_LOOKUP, NFS_READ, NFS_WRITE, NFS_CREATE, NFS_MKDIR, NFS_REMOVE, NFS_RMDIR, NFS_RENAME and NFS_COMMIT. MDS calls monitored include: MDS_LOOKUP, MDS_CREATE_OBJECT, MDS_RELEASE_OBJECT, MDS_APPROVE_WRITE, MDS_FINISH_WRITE. Storage-node calls monitored include: READ, WRITE, CREATE, DELETE. Disk calls monitored include: READ, WRITE.



```
    diskObj->stats.coalescedRequests++;
}

numBlocks = schedObj->ioRegion->ioSize / S4_DISK_BLOCK_SIZE;

S4_DPRINTF(S4_DEBUG_SCHED, 3, "Sched - DiskIssue: Issuing request\n");

/* Add request to issued queue */
SchedQueueAddIssued(schedObj, &diskObj->issuedQueue);

/* ATC records */
uint8_t atcTYPE = (schedObj->type == S4_DISK_READ) ?
    DISK_READ_START : DISK_WRITE_START;
DISK_OP_post(DISK_OP_TYPE, 0 /* ts */, schedObj->breadcrumb,
             diskObj->deviceHandle, schedObj->ioRegion->ioOffset,
             schedObj->ioRegion->ioSize, atcTYPE);

/* Issue request to the drive */
S4Disk_IssueRequest(diskObj->deviceHandle, schedObj, schedObj->ioRegion,
                    schedObj->type);

/* ATC records for synchronous case */
if (schedObj->callback == NULL) {
    atcTYPE = (schedObj->type == S4_DISK_READ) ?
    DISK_READ_END : DISK_WRITE_END;
    DISK_OP_post(DISK_OP_TYPE, 0 /* ts */, schedObj->breadcrumb,
                 diskObj->deviceHandle, schedObj->ioRegion->ioOffset,
                 schedObj->ioRegion->ioSize, atcTYPE);
}
```

Figure 4.5. **Example instrumentation of a Disk request.**

Such record types may be enabled or disabled at run time. We believe that this will encourage programmers to insert as many record types as necessary to diagnose a problem; they can always turn them off by default and re-enable them when associated components need to be analyzed. We currently use a small embedded database, SQLite [SQLite, 2005], for the ActivityDBs.

4.3.3 Implementation details and APIs

The design document for the Ursa Minor project has the latest implementation details for the Stardust client. This subsection extracts the salient points from it. The interested reader is advised to consult the design document for more details.

This subsection provides details on the seven main building blocks of Stardust:

(1) The APIs for posting activity records.

(2) The storage of the activity records.

(3) The stitching of activity records with the originating requests.

(4) The querying of the activity records across multiple service centers.

(5) The generation of statistics and virtual counters.

(6) Turning activity collection on and off.

(7) Balancing the load of ActivityDBs.

Posting activity records

Request activity is captured by inserting activity records inside the Ursa Minor code. As a simplified example, in order to allow one to determine how much CPU a storage-node has spent on behalf of requests from a certain client, the storage-node posts (StartWork: breadcrumb=5, time=0) and (EndWork: breadcrumb=5, time=0.0001) records whenever it starts and ends work on a particular request with breadcrumb 5. The CPU time for this request is then simply the difference in timestamps between these two records (assuming no context switches happened in between; in practice context switches do happen, and we handle them as well). One design goal is to provide a flexible, unified way of posting activity records.

An activity record is composed of a common activity record header and one or more tuples of the form (attribute, value). The activity record header is described in detail below.

Each record type has a unique API function used to post activity. The API is shown below:

recordtype_post

– Input arguments:

Header Common header

Flags Various flags

Value 1 Value that corresponds to Attribute 1

...

Value N Value that corresponds to Attribute N

– Output arguments:

void

– Description:

Any module posting this record includes the common activity record header and n values, one for each of the n attributes the activity record has. Currently, the only flag is called NODROP, which tells the Stardust client that it should not lose this record even if that means that the performance of the system could degrade.

The code used to implement the posting functions is automatically generated, much like with RPC stub generators. The provider specifies in a special file, called *records.txt*, the argument types the function takes and their respective names: RecordType_Post(Name1, type1, Name2, type2, ..., ...). *RecordType_Post* is the name of the function the code will call to post the record that contains N types, *type1, ..., typeN*. The types are the usual BYTE, SHORT, INTEGER, LONG, STRING, etc.

Once the module implementor defines the record types in the file mentioned above, a script called *register_records.pl* is run manually. This script generates automatically a *.h* file with the function definition and also creates a table in the database with name *FUNCTION_NAME_TABLE*. Existing tables are not affected by the script.

In addition to the types each user may specify, there is a common header prepended to each activity record. The format of the header is illustrated in Figure 4.6. The *CPU Cycle Timestamp* is a unique timestamp for this record on the given physical server. The *Breadcrumb* field is a unique request ID assigned to a request that created this activity record. In the current implementation, the breadcrumb is a 64 bit CPU cycle counter value, whose uppermost 8 bits contain the client's ID[2]. The *Kernel PID* is the process ID of the process that is posting the record. The *User-thread PID* is the thread ID of the user-lever thread that posted the record. It could be the ID of a *state thread* or *pthread*, depending on the particular implementation.

[2]For clients connecting through the NFS server the client ID is assigned by the administrator.

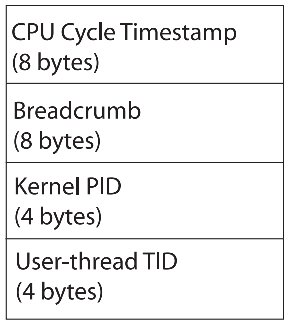


Figure 4.6. **The common activity record header.** This figure depicts the common activity record header prepended to every record. It's total size is 24 bytes.

These fields are usually filled in automatically by the Stardust client during a record posting.

Storage of activity records

Posted records are eventually stored in a relational database. There were several design considerations that led to deciding to store the activity in a relational database. Below, we briefly discuss each approach that was considered.

The first approach stores the trace records in flat files, with the file name being the time of the first trace recorded. This is similar to the way most trace-collectors have stored and distributed traces. For example, the HP Lab Cello traces [Hewlett-Packard, 2007] and the Harvard NFS traces [Ellard et al., 2003] were collected this way. An advantage of flat files is that compression is straightforward (e.g., tightly padding the records so that not many bytes are wasted, or gzip). Also, trace record entries do not have to be the full length of the type; diffs can be kept as described in [Samples, 1989; Verbowski et al., 2006]. The main drawback is in querying of the trace. Unless all queries want to query traces by time (the only implicit index), querying can be extremely inefficient. For example, listing all requests of from client c would require scanning through all the traces. In fact, most queries that relate to other information in addition to time, require scanning through all the traces. We are interested in flexible querying of activity records, hence this approach did not work well for us.

The second and third approaches use databases as the way to store online traces. They differ on the way the table layouts are built. The second approach uses just one big table. The table has numbered columns (1 through N). One of the first columns is the activity record type. To determine the meaning of the N columns for this type one has to consult a second small table that, for each type, keeps a mapping between column numbers and string describing what those columns are. The advantage of this approach is that each query only has to consider one table and the number of joins is reduced. The main disadvantage is that indexing any of the N columns is difficult (if not impossible), because they will be of different types. Also, we would have to declare the type of each of the columns as generic (in database jargon, *BLOB*). That could mean less efficient querying. Another disadvantage is the storage space wasted. Each row must have N columns, where N is the number of columns the largest row has. An improvement to this approach would be to have a certain number of columns dedicated to keeping integers, strings, etc., and have each record type use the appropriate columns. Although this approach would allow indexing of columns, it still suffers from the having storage space wasted.

In the third approach, each activity type has a separate table as shown in Figure 4.7. The main disadvantage of this approach is that join operations are required to collect demands and latency graphs. For example, if two tables are used, one for storing activity records that indicate a START event and the second for storing END events, calculating the time difference between END and START requires a join on the breadcrumb column. Hence, we have a classic tradeoff between storage space and querying speed. Our particular implementation chooses the third approach, but these tradeoffs need to be considered for other environments.

Pruning traces: The activity traces can take considerable space. However, we do not expect to maintain all raw traces forever. From our experience with using these traces, we have found that pruning the CPU, network, and disk traces to generate per-client, per-resource performance counters and

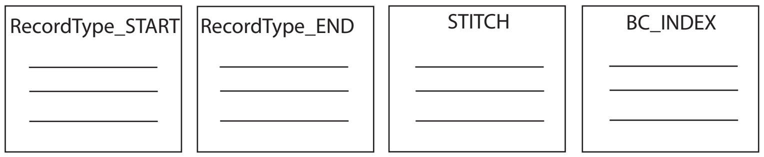


Figure 4.7. **The four main table types in Activity DBs.** Each START and END activity record goes to a separate table. There is a pair of such tables for records collected from each service center. There are STITCH tables that contain relationships between breadcrumbs. There is one such table for each Stardust client. There is only *one* breadcrumb index table per Activity DB that, for each breadcrumb in the system, contains a pointer to tables in which a record with that breadcrumb is stored.

request flow graphs every T seconds (e.g., 1 second) is acceptable for performance predictions. For *performance anomaly detection*, traces may have to be kept for longer, e.g., 1 to 2 weeks. Although some basic performance anomaly detection to verify model correctness is in the scope of this thesis (and is presented in Chapter 6), general anomaly detection, for example for security or intrusion detection purposes, is not. The storage space needed for several workload types, together with savings to be expected from pruning, are evaluated in Section 4.4.

Activity DB APIs: The insertion of records into the database is done in two steps. First, the Stardust client periodically sends to the Activity DBs large buffers containing activity records. Second, the Activity DB accepts these buffers and inserts the records they contain into the database.

The second step is done in a database-specific manner, using SQL. The first step is done using a data transport layer. The format of the message sent from the Stardust client to the DBs is:

records_send:<Buffer>. A buffer containing multiple records is sent to the Activity DBs through this message. Care is taken that network priority scheduling is used so that large buffers of records do not add to the latency of foreground requests. To do that, we changed the RPC layer in Ursa Minor to have scheduling support for Stardust traces. Without scheduling support, large buffers of traces are periodically sent to the Activity DBs. Any foreground requests that are blocked behind the buffers incur a large latency.

Scheduling support mitigates this effect. With scheduling support in the current implementation, the activity trace buffers are split into smaller sizes, and each small chunk is given a lower scheduling priority than foreground requests. This greatly reduces the additional latency seen by foreground requests, as Section 4.4 quantifies.

Activity DBs processing: During online collection, records are initially stored in a staging area, which is a flat file. This allows for very high throughput (e.g., 500,000 records/sec) at the Activity DB. A thread periodically takes records from the staging area and stores them in the relational database, thus enabling querying. It is expected that a database has lower insertion throughput (e.g., 50,000 records/sec, i.e., usually an order of magnitude higher than flat files). Records are written in bulk, i.e., one SQL query is made for inserting thousands of records at a time.

The layout of traces in tables is as follows: There is a unique per-endpoint (endpoint means service ID, e.g., the NFS, MDS services all have an endpoint associated with them), per-record type table. For example, if there are 100 endpoints in the system and each can post 1000 different record types, there will be 100x1000 = 100,000 tables in the database. The timestamp is the only primary key for each of these tables. Figure 4.7 shows the START and END table types (these two tables are there for most record types).

Handling changes in activity record schemas: It is not inconceivable that activity record types for a certain entity may change. For example, a provider could decide to add or remove an attribute. Another change could be to change the type of one of the attributes, say from an INTEGER to a STRING. The issue is: what happens to the database tables that store those entries and what happens to modules that query those records by using the old version of the query?

Currently we do not provide an implementation of a solution. The design, however, allows for the following workaround. In the case when new columns are added, no special care must be taken. The previous values obtained when the column was missing can be all NULL. In the case when columns are deleted, the whole column can be deleted and queries that needed that column will have to be modified. In the case when column types are changed, old values can be substituted by NULL.

Preserving context through stitching

Most client requests generate multiple activity records, as they move through the system. Stardust needs to associate activity records with client requests.

There are certain requirements that drive our context preservation design and implementation. First, adding breadcrumbs and propagating them should require minimal programmer effort and minimal code changes (e.g., APIs should ideally not change). Second, the activity tracking should be done incrementally, and meaningful activity records should still be posted even though not all service centers through which a request is passing are posting activity. Third, a request could spawn multiple sub-requests. There needs to be activity records matching every parent request with children requests. Fourth, multiple sub-requests could be coalesced into one sub-request. There need to be activity records matching every request with the coalesced request. Fifth, if a request crosses server boundaries, the breadcrumb must be propagated. This means that the messaging layer has to be agnostic to the breadcrumbs.

Without the stitching of activity records to preserve context, Stardust is still an interesting unified framework for collecting traces. However, it is the stitching of activity records that allows end-to-end tracing of requests through the system.

A challenge in implementing stitching is the tradeoff among 1) doing a lot of computation on the critical path for stitching activity records and 2) doing a lot of computation off the critical path during a query. Our current bias is for 2), i.e. very little computation should be done on the critical path (while still allowing for specialized tools to make use of such traces). However, given that some queries could happen online, Stardust cannot afford to be too slow in querying either.

Another approach that was considered, but not implemented, is described by Isaacs et al. [2004]. It consists of temporal joins on attributes that make up an activity record. That approach was subsequently used in the Magpie project [Barham et al., 2004]. Such an approach does not require passing breadcrumbs around, making it more elegant in that respect. A parser stitches requests together in a chain and it looks at any arbitrary attribute when performing the join. There are tradeoffs between Magpie's approach and Stardust's. Magpie's approach is more generic and flexible. Stardust's approach trades off flexibility in describing types of requests with efficiency of querying. It is well-known in the database literature that joins are expensive operations; Magpie's approach requires joins to track a request even within a single server, and each join may be on multiple attributes, whereas Stardust uses fewer joins (for example, no joins are needed when a request maintains the same breadcrumb) and they are only done on a single attribute (the breadcrumb). However, Stardust needs more information to be recorded (the breadcrumb) in each record.

The current implementation of Stardust propagates breadcrumbs through private state in user-level threads, rather than through API modifications. A user-level thread may call many functions. Instead of adding another parameter to these functions, the breadcrumb is made part of the thread structure. Of course, care is taken to propagate breadcrumbs from thread to thread as request processing moves through the system. The APIs used to get a unique breadcrumb is:

get_breadcrumb

- Input arguments:

  void

- Output arguments:

  breadcrumb An opaque data structure representing the breadcrumb.

- Description:

  A unique breadcrumb is received this way. The breadcrumb is unique within each server.

Relationships between breadcrumbs are posted as special activity records. For example, if request with breadcrumb 5 generates two new sub-requests, A and B, two new records of type "stitch" will be posted, with record payloads of the form (breadcrumb of 5, breadcrumb of A) and (breadcrumb of 5, breadcrumb of B). The API that does this is :

stitch_breadcrumbs

– Input arguments:

Source breadcrumb The breadcrumb initiating the link.

Destination breadcrumb The breadcrumb linked to the source breadcrumb.

– Output arguments:

void

– Description:

Two breadcrumb values are linked in a causal chain by posting a "stitch" record.

Whenever processing of a request moves from one thread to another, so does the breadcrumb (we as programmers had to find those places and explicitly pass the breadcrumbs from one thread to the other). Getting and setting the thread's breadcrumb is done through the APIs below:

get_current_breadcrumb

– Input arguments:

void

– Output arguments:

Breadcrumb Current breadcrumb of the processing thread.

– Description:

This function returns the breadcrumb associated with the currently executing user-level thread.

set_current_breadcrumb

– Input arguments:

Breadcrumb Breadcrumb of the request this thread is processing.

– Output arguments:

void

– Description:

This function sets the breadcrumb of the current thread.

If a request spawns many sub-requests (one-to-many), each of the new requests will have a new breadcrumb, and a record that stitches the old and a new breadcrumb will be posted. If multiple requests spawn one common request (many-to-one), as is the case when multiple write requests are coalesced into one large request, for example, then one new breadcrumb will be generated for the new request and records will be posted that stitch each of the old breadcrumbs to the new breadcrumb.

If a request moves from one Stardust client to another, for any reason, a new breadcrumb is assigned to that request *before* it leaves for the new Stardust client, and a record is posted that stitches the old breadcrumb that request had with the new one. The reason a new breadcrumb is assigned is to create the "happened-before" relationship. There is no synchrony assumption between Stardust clients (and the servers they reside on), hence this stitching serves as an ordering mechanism.

When stitching together a causal request chain for post-analysis, the program needs to know where to find the next request to stitch with. For example, a request at the NFS server may hit in cache (so the next record is in the CACHE_HIT table). It may also miss in cache (so the next record is in the CACHE_MISS table). Should the stitcher program check both tables? Once one considers the general problem that a request may potentially go anywhere, we are faced with the problem of having to potentially check each table in the Activity DB to find the next record. That is extremely expensive.

There are two ways to handle this situation. The first is by making assumptions about the structural behavior of the system. For example, a cache model can be used to predict if a request hits or misses in cache.

These expectations can be encoded in the form of a request path graph as described in Section 3.4 and Section 5.2. The second option is to make no structural behavioral assumptions. Instead of checking every Activity DB table, however, an index table (currently called BC_INDEX, as shown in Figure 4.7) is created which contains, for each breadcrumb seen in the system, the name of the instrumentation point it was posted from (and thus, implicitly the name of the Activity DB table to find the complete record). Any insertion in any Activity DB table incurs an insertion into the BC_INDEX table .

Currently, a hybrid version is implemented. After expectations are checked and verified, the index entries can be deleted. If the BC_INDEX table records unexpected relationships, the administrator may have to adjust the expectations as described in Section 6.1 and Section 6.3.

Querying activity records

Activity records are stored in relational Activity DBs to make querying easier. The database schema is such that resource demands can be extracted using simple queries that require no joins (or at most one join, if START and END activity records are in separate tables). Request latency graphs do require joins, however (on the breadcrumb column of the database tables).

Queries can be made directly with the database, by initiating a connection to it. This is the default and easiest way to query. A second supported way is to query through a Stardust interface below. First, the Stardust client for a desired entity is asked to provide the endpoint of the Activity DB on which it is storing its traces. This is done through the following API:

find_db

– Input arguments:

  Endpoint Endpoint identifying the Stardust client of an endpoint

– Output arguments:

  DB_Endpoint Endpoint of the Activity DB containing the records

– Description:

  The Activity DB endpoint for use by this Stardust client is returned.

Once the DB's endpoint is found, the APIs for querying are straightforward. A query is initialized through :

init_query

– Input arguments:

  DB_Endpoint Endpoint identifying the DB containing the records

  SQLQueryString An SQL statement

  CallbackFunction Callback function called when records returned from this query are available.

– Output arguments:

  QueryID Cookie that identifies this query.

– Description:

  A query is initiated through this call. A callback is specified. Whenever the query records are ready, the callback function is called. Note that depending on the implementation, multiple records may be returned through one callback call. Currently there can only be one callback attached to a single query.

Note that there is an equivalent method of querying: the pull model, where a user initiates a query and then periodically pulls results from it. Yet another way of obtaining query results is to save the output of a query to a file and return the file ID to the interested party. That party then reads the file. These approaches suffer from the problem of garbage collection. Handling old files/handlers and knowing when to delete them is not trivial.

Statistics and virtual counters

Many systems have hard-coded counters that calculate metrics of interest. For example, the performance counters found in Windows calculate the average number of disk I/Os, buffer cache misses, current CPU utilization etc.

(over a period T). These statistics are important to understand the overall health of the system. In addition to a set of well-known counters, Stardust supports "virtual counters", which are counters obtained through database queries. Unlike hard-coded counters, virtual counters can be generated when needed using the collected traces.

Turning activity collection on and off

By default, activity tracking is on all the time. If found necessary, each activity record type can be grouped into well-known groups. Tracing for any of these groups can be controlled through the following APIs:

set_filter

– Input arguments:

  Endpoints List of endpoints.

  My_Endpoint Endpoint of the entity setting the mask.

  Filter Filter that specifies what group will be turned on and off. A filter may contain regular expressions or wild characters.

– Output arguments:

  Void

– Description:

  A filter can specify which activity records should be collected (and/or which should not).

In addition to stopping record generation at the source, records can be filtered at the Activity DBs through a similar interface as the above. That interface interacts directly with the Activity DBs. Whether users will opt to stop records at the source or at the Activity DBs is left up to their preference. In general, stopping records at the source minimizes any impact on foreground workloads and is currently the only way we are using the filters.

Balancing the load of Activity DBs

The system has a pool of Activity DBs that holds the records from multiple Stardust clients. Each Activity DB consists of a database process and a frontend that receives queries and replies with answers to clients. A design decision to reduce complexity dictates that there should be no interactions between Activity DBs. A client's traces should fully reside on only one Activity DB. Each Stardust client chooses an Activity DB at random, the first time it executes. Thereafter, the Stardust client uses that Activity DB forever (the mapping between client and Activity DB is logged). Although the design is flexible, all our experiments have used one Activity DB.

We currently do not implement a solution to the case when an Activity DB is permanently removed. Also, if an Activity DB is temporarily down, the data to be sent to it is temporarily buffered at the Stardust client and then lost if the Activity DB does not come up on time. However, in theory, another Activity DB can be used as temporary storage for the Activity DB that is down. However, we have not experimented yet with graceful handling of Activity DB failures.

4.4 Evaluation

This section is self-contained and limits the evaluation to the efficiency of collection and querying of the activity records. Chapter 5 further evaluates the usefulness of this infrastructure.

4.4.1 Experimental setup

In the following experiments, we used Dell PowerEdge 650 servers equipped with a single 2.66 GHz Pentium 4 processor, 1 GB of RAM, and two Seagate ST33607LW (36 GB, 10k rpm) SCSI disks. The network configuration consisted of a single Intel 82546 gigabit Ethernet adapter in each server, connected via a Dell PowerConnect 5224 switch. The servers ran the Debian "testing" distribution and used Linux kernel version 2.4.22. The same server type was used for all clients and storage-nodes. The storage-nodes used one of the two local disks for data and the other for the operating system. Each NFS server and storage-node was configured to use 256 MB and 512 MB for its buffer cache, respectively. The experiments used several workloads with varying characteristics to assess the efficiency and efficacy of Stardust. These are all workloads a storage system is expected to handle.

OLTP workload: The OLTP workload is a TPCC-like [Transaction Processing Performance Council, 2002] workload that mimics an on-line database performing transaction processing. The workload is run on the Shore database storage manager [Carey et al., 1994] and configured to use 8 KB pages, 10 warehouses and 10 clients, giving it a 5 GB footprint. This workload is closed-loop. The performance of this workload is reported in transactions per minute (tpm).

Postmark workload: Postmark [Katcher, 1997] is a file system benchmark designed to emulate small file workloads, such as e-mail and netnews. It measures the number of transactions per second that the system is capable of supporting. A transaction is either a file create or file delete, paired with either a read or an append. The configuration parameters used were 100000 files, 50000 transactions, and 224 subdirectories. All other parameters were left at their default values. Postmark is a closed-loop workload. Postmark's performance is reported in transactions per second (tps).

IOzone workload: IOzone is a general file system benchmark that can be used to measure streaming data access performance [Norcott and Capps, 2002]. For our experiments, IOzone measures the performance of 64 KB sequential writes and reads to a single 2 GB file. IOzone is a closed-loop workload. IOzone's performance is reported in megabytes per second read.

"Linux build" development workload: The "Linux build" workload measures the amount of time to clean and build the source tree of Linux kernel 2.6.13-4. The benchmark copies the source tree onto a target system, then cleans and builds it. The results provide an indication of storage system performance for a programming and development workload. The source tree consumes approximately 200 MB before the build. This workload is closed-loop. The performance of this workload is measured in seconds to complete the build process.

SPEC SFS Suite: SPEC SFS is a benchmark suite designed to measure the response time of NFS requests for varying throughput levels [Standard Performance Evaluation Corporation, 1997]. The latency of the NFS server is measured as a function of throughput. This workload is open-loop.

"S1-V0" scientific workload: This workload corresponds to storing the output from sample queries performed by a system designed to analyze multi-dimensional *wavefield* datasets. Query outputs are written to storage for subsequent processing. S1 and V0 correspond to sample queries on multi-gigabyte seismic wave-fields produced by numerical simulations of ground motion wave propagation during strong earthquakes in Southern California [Akcelik et al., 2003]. S1 corresponds to the output of temporal queries on a surface, and V0 corresponds to the output of temporal queries on a volume. The performance of this workload is measured as the overall run time for each query. This workload is closed-loop.

All experiments are run five times and the average is reported (together with confidence intervals), unless otherwise mentioned.

4.4.2 Instrumentation framework efficiency

Baseline experiments: Table 4.3 shows the overheads of the end-to-end tracing when each of these workloads is run in isolation. The application data was stored on a single storage-node (i.e., there is no data replication). There is a single Activity DB for the traces. The baseline performance (with tracing disabled) and performance with tracing enabled is shown. As seen from the table, Stardust's demands on the CPU, network, and storage are relatively small. IOzone generates the largest amount of traces, because it processes many more requests per second than the other workloads. The instrumentation added at most a 6% performance penalty. The performance standard deviations for the performance metrics were as follows: for Postmark, the baseline and new standard deviations (STDEV) were both 0. For OLTP, the baseline STDEV was 3.4, whereas the new STDEV was 8.5. For IOzone, the baseline and new standard deviations were both 0. For Linux-build, the baseline STDEV was 3.3, whereas the new STDEV was 5.1. For

	CPU demand	**Network and storage demand (MB/s)**	**Perf. without tracing**	**Perf. with tracing**
Postmark	0.9%	0.34	11 tps	11 tps
OLTP	0.7%	0.57	910 tpm	898 tpm
IOzone	0.1%	3.37	38 MB/s	36 MB/s
Linux-build	0.1%	0.49	1094 secs	1101 secs
S1-V0	0.8%	1.85	669 secs	686 secs

Table 4.3. **Macro-benchmark performance.** This table illustrates the overheads of Stardust. Stardust places demands on the CPU for encoding and decoding trace records and on network and storage for sending the traces to the Activity DBs and storing them. It also places a fixed demand of 20 MB (by default) of buffer cache at each client machine. The impact of the instrumentation on the workload's performance is less than 6% in these experiments.

S1-V0, the baseline STDEV was 1.9, whereas the new STDEV was 1.8. The standard deviations for the CPU, network and storage demands were negligible.

Figure 4.8 shows the output from the SPEC SFS benchmark. Throughout the curve (from low to high NFS server load) the impact of the instrumentation on average throughput and latency is low.

Figure 4.9 shows the contents of the Activity DB for different workloads, after the workloads have finished running. For all workloads, most of the content represents CPU context switches. Each time a context switch (at the kernel or user level) occurs, it is logged ("CPU Kernel" and "CPU User" categories). Many of the workloads fit fully in buffer cache, and only a few of them generate disk I/O records. A considerable fraction of the database size is used for keeping causal path information in the form of STITCH tables.

Network scheduling support for traces: Figure 4.10 shows the impact of adding network scheduling support for the activity traces. We modified the RPC layer in Ursa Minor to give priority to other activities over delivery of traces. Without such scheduling support, large buffers of traces are periodically sent to the Activity DBs. Any foreground requests that

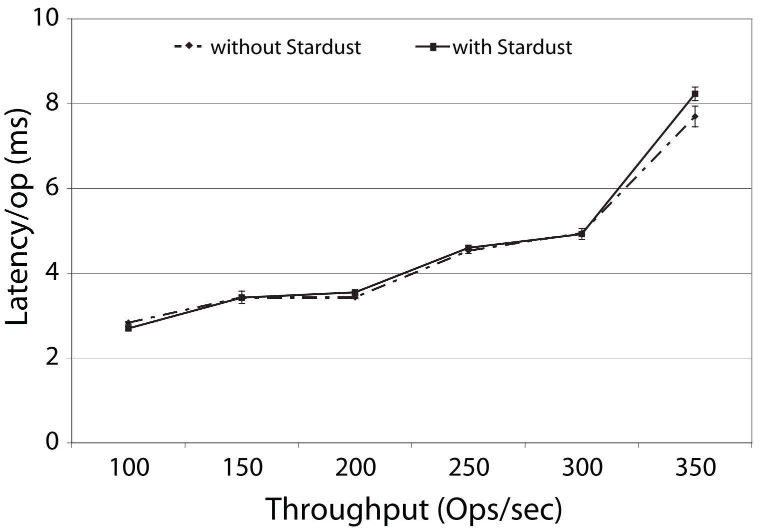


Figure 4.8. **Impact of instrumentation on latency of SFS workload.** For all throughput and latency levels, the overhead of instrumentation is low (6.8% in the worst case). Standard error bars are shown.

are blocked behind the buffers suffer a large latency. The scheduling support mitigates this effect. With it, the activity trace buffers are split into smaller sizes, and each small chunk is given a lower priority than foreground requests. This reduces the additional latency seen by foreground requests.

Efficiency with data redundancy: Because Ursa Minor is built on commodity components, data is often replicated across storage-nodes to ensure an acceptable level of reliability and crash tolerance (Chapter 5 describes an alternative to replication, called *erasure coding*, but this discussion focuses on the former without loss of generality.) N-way replication refers to data being replicated across N storage nodes. During a write, data is written to all N nodes, and, during a read, data is read from one of the N storage nodes. Figure 4.11 shows the storage demand required when replication is used for each of the representative workloads. In general, the storage demand increases linearly with the replication degree, since an increase in the replication factor results in an increase in the number of storage nodes accessed. The overhead increases for several reasons. First, there are more network

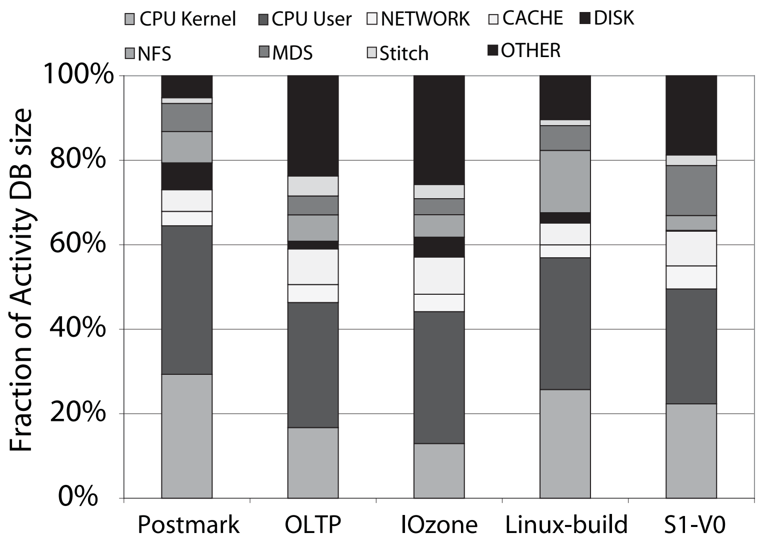


Figure 4.9. **Contents of Activity DB.**

interactions and thus more network records, since more storage-nodes are accessed (during writes). Second, there are more buffer cache accesses and thus more cache records, since now a data block resides on N nodes. Third, there are more disk accesses and thus more disk records, since a data block is written on N disks. In general, the number of trace records increases as replication increases, and the trend is similar for all workloads.

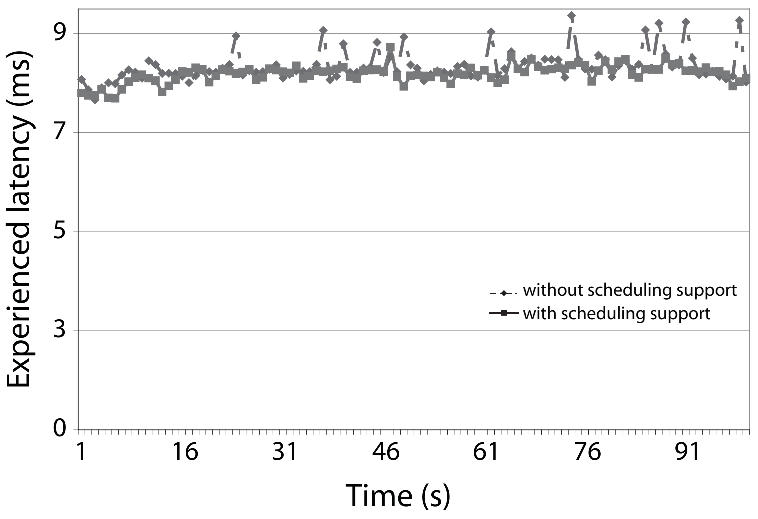


Figure 4.10. **Network scheduling support for Stardust.** Latency as seen by a sequence of Postmark NFS_CREATE requests, with and without scheduling support. The average latency with and without scheduling support is 8.2 ms and 8.3 ms respectively. The standard deviation is 0.18 ms and 0.3 ms respectively.

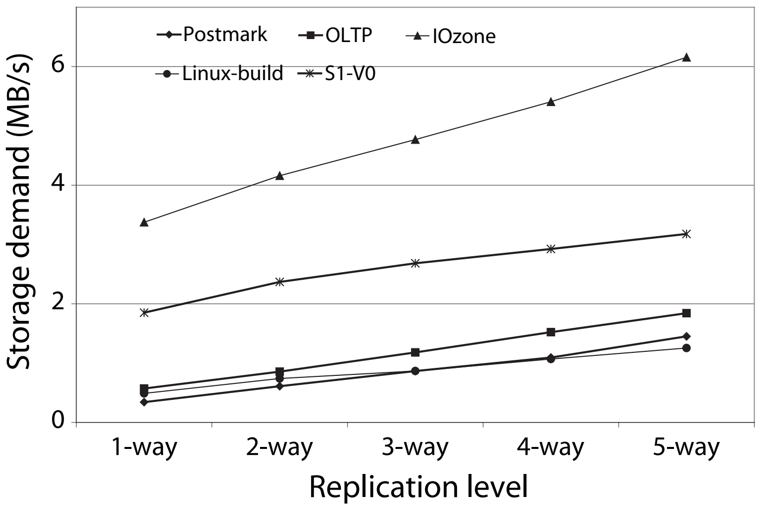


Figure 4.11. **Amount of trace data as a function of replication level.**

	Workload runtime (s)	**Read path**	**Write path**	**Memory needed (MB)**
Postmark	9792	8.5	19	220
OLTP	636	8.5	135	103
IOzone	329	11.4	109	323
Linux-build	1786	8.2	42.9	103
S1-V0	670	4	12.5	263

Table 4.4. **Number of SQL queries for creating a single request path.** 10000 causal paths are created and averaged to give the average number of queries needed for a *single* request path. In all cases, the information needed to create *all* paths for all requests can fully reside in memory for the duration of the run. The standard deviation for all the metrics, except for the read path, is negligible. However, the standard deviation for the read path is 5, 5, 3, 5, 0 for each of the workloads, respectively.

4.4.3 Querying framework efficiency

In addition to trace capture overheads, 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 4.4 shows the average number of SQL queries required to create such a path for each of the workloads (these queries are issued internally). Providing the number of queries is more useful than providing the time required to create the path, since that time depends on factors such as the database used, amount of buffer cache dedicated to the database, whether the database is over a network link or is local. The number of SQL queries provides a more standard way to compare the work needed to create a path. All queries are simple SELECT queries that just select all columns of a row with a given breadcrumb. 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 breadcrumb column has an index on it.

The time to create a path depends on two main workload factors. First, the deeper a request flows into the system (e.g., when it misses in the NFS server 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 the buffer cache. Second, coalescing (many-to-one) and splitting (one-to-many) requests causes the path of a request to include sub-paths of other requests. Re-creating the full path of an original request currently re-creates the path of all other requests that were coalesced with it. Thus, the cost of re-creating the write path is usually larger than for reads. Some workloads do not exhibit sequential behavior (e.g., Postmark), and little or no coalescing happens for them.

Table 4.4 also shows the amount of data (in form of trace records) needed to construct the request path. The information can usually reside in memory for the duration of the workload run, which speeds up the path construction algorithm. In practice, we have seen individual path creation times ranging from a few microseconds, when the breadcrumbs are still in the buffer cache of the Activity DBs, to a few milliseconds when the breadcrumbs 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). The rather large time to construct write flows is due to write coalescing. As another example, 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). For both benchmarks, the time to create the full request flow graphs can be an order of magnitude or more higher if the database resides on disk or over the network. We believe that techniques such as sampling could be used to create request flow paths for a percentage of the requests, thus further reducing the path construction time. However, we have not experimented with such techniques.

In addition to the main read and write calls, request flow graphs can be created for other NFS calls as well. For example, Figure 4.12 shows the request flow graph taken by NFS_Create calls from a specific client for the Postmark workload. A graph is created for 140 randomly chosen requests and the time between instrumentation points is averaged, as first described

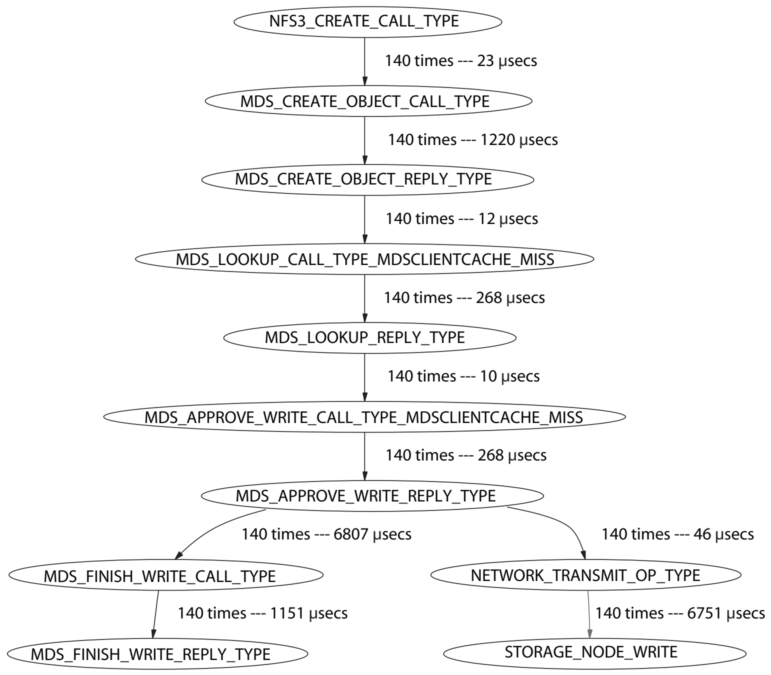
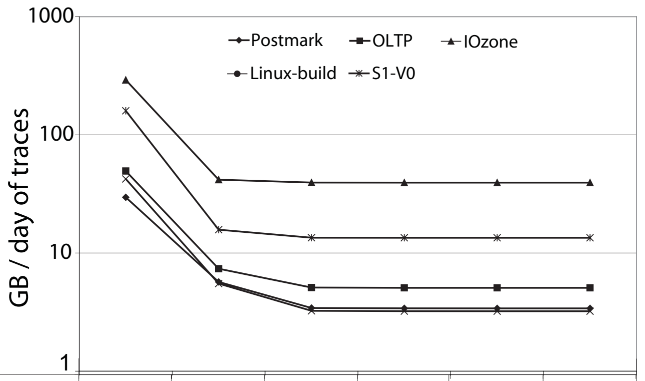


Figure 4.12. **Example (simplified) path of NFS_Create calls.** This path was obtained using the Postmark workload and averaging 140 paths. The nodes contain information such as the unique identifier of the component posting the record and the string name of the record. The edges contain latency information between two instrumentation points. Some requests may be processed in parallel. These graphs can be generated using the trace data and a readily available visualization tool called DOT [GraphViz, 2006].

in Section 4.2.3. Such graphs can be reconstructed offline by querying the ActivityDBs.

4.4.4 Trace pruning methods

The trace-based approach to measuring performance allows for easy integration into a system. However, the system is not expected to maintain all raw traces forever, since they consume storage space. From our current experience with using these traces, we have found that pruning the CPU, network, and disk traces to generate per-client performance resource and latency in-



Pruning interval T	no pruning	10 ms	1 sec	1 min	1 hour	1 day
Resource demands	N/A	0.3 GB	3 MB	0.05 MB	0.8 KB	0.034 KB
Latency graph	N/A	1.9 GB	19 MB	0.33 MB	5.5 KB	0.2 KB

Figure 4.13. **Space savings from pruning.** The graph illustrates the average storage savings from pruning the CPU, network and disk traces to get demand and latency information. However, full traces of buffer cache and disk accesses are still kept and their cost dominates. The table beneath shows the amount of data needed for just resource demands and latency information as a function of pruning interval T (which serves as the graph's x-axis as well). The standard deviation is negligible in all cases.

formation is acceptable for performance predictions. However, we keep full buffer cache and disk traces as well.

Figure 4.13 shows the storage demand *after* the traces derived from the workloads are pruned every T units of time. The pruning happens offline, after the full traces have been collected. As T increases, the amount of trace data decreases proportionally. The graph shows the amount of trace data from keeping pruned CPU, network, and disk traces and also full buffer cache and disk traces (pruned disk traces reveal disk utilization metrics, whereas the full traces can be used for trace re-play). The table beneath the graph shows that further savings can be made if the full buffer cache and disk traces are not kept. The table shows just the capacity needed if only the per-client, per-resource demands are kept, or if only the latency graphs are kept. Chapter 5 discusses cases when we have found it useful to keep the full buffer cache and disk traces, in addition to the resource and latency information. In addition, that chapter discusses how models can *predict* future resource demands. In those cases, traces can be immediately discarded once the models have learned how to predict well.

4.5 Summary, lessons learned, and limitations

Stardust is a measurement infrastructure for shared, distributed systems. It is based on monitoring entry and exit points from software modules that make use of the four common resource types (CPU, network, buffer cache and disks). It can be easily extended to monitor other resource types, such as lock resources. Stardust should be ON at all times, and the collection of activity records is relatively lightweight. Querying the wealth of information is done through the SQL language on Activity DBs, which are relational databases.

We learned several things in the process of working with Stardust traces. First, the management of the instrumentation points has proven to be a headache. The instrumentation is inserted manually in the code, and needs to be maintained as a first class aspect of each instrumented software module. Every time the code base is changed, there is the potential that the request causal paths change. Hence, the person that introduces the change must also worry about maintaining the instrumentation points. In practice, we have observed that most instrumentation points are not affected most of the time. But, sometimes, some are. Ideally, the instrumentation infrastructure would be self-maintaining and discover when request causal paths change and instrument the code automatically. Some dynamic programming analysis techniques have been successful in automatically instrumenting programs that reside on a single server (e.g., see papers from the Paradyn group [Hollingsworth et al., 1994]). A very recent paper discusses extending these techniques to a distributed environment [Chanda et al., 2007].

Second, Stardust was designed with performance prediction in mind, and there is still room for improvement if it is to be used for performance anomaly detection and diagnosis. For the latter, which often seeks to discover a problem's root cause, much more fine-grained instrumentation may be needed. For example, Stardust maintains an activity record each time a request goes to the disk subsystem (from the Ursa Minor file system to the kernel) and comes back (to our filesystem from the kernel). If a disk request takes unreasonably long, the trace records will not indicate whether the disk hardware or some module in the kernel slowed it down, because these components are lumped into one black box.

Third, there is room for maintaining the full request traces with improved encoding (while still maintaining the querying flexibility relational databases provide). For example, if a file is accessed 1000 times sequentially, we currently record the file name 1000 times. Doing so consumes significant storage space. Maintaining records in terms of well-defined "diffs" would be a good approach [Samples, 1989; Verbowski et al., 2006], but may not easily fit a database storage constraint.

Fourth, there is much room for improved visualization tools that display different "views" obtained from the measurement framework. Such views could be useful for anomaly detection, diagnosis and so on. Challenges include building automatic tools that show only relevant changes to the system state during an anomaly. A recent white paper describes efforts within our research group to build visualization tools on top of Stardust [Sambasivan et al., 2007].

5 Case studies with expectation-based models

Whereas Chapter 4 evaluated the efficiency of the measurement framework, this chapter evaluates the efficacy of the modeling framework in answering several important *What...if* questions. The testbed used in this chapter is a system that we designed from scratch, Ursa Minor. Hence, we had the opportunity to design into it the modeling infrastructure. Appendix A outlines an experience with incorporating such a framework in a legacy system.

Ursa Minor, first briefly discussed in Section 4.3, is a cluster-based storage system that exposes a large number of performance tuning knobs and choices. Thus, it provides an excellent testbed for the modeling infrastructure. This section overviews the knobs and the difficulties administrators have with tuning them.

5.1 Overview of Ursa Minor's versatility

Traditional storage systems are built around a single-vendor, monolithic disk array design. Such systems provide high performance and availability, but they are expensive and do not scale easily. Incremental scaling is not an option with such systems, as a client must buy and manage another monolithic system when scalability requirements slightly exceed the existing array system. Ursa Minor is a cluster-based storage system, built from commodity hardware, that has been developed to address these scalability and cost issues [Abd-El-Malek et al., 2005]. The individual servers are often called *storage-nodes*, and each provides a certain amount of CPU, buffer cache and storage. Incremental scalability is provided by addition of storage-nodes into the system. These components can be inexpensive mass-produced commodities and often contain heterogeneous resources. Even when buying from the same vendor, year after year, the components change (e.g., they may contain faster CPUs, more memory, and faster network cards). Ursa Minor is designed to be shared among many clients with potentially different workloads.

Commodity hardware is often less reliable than customized hardware, and individual storage-nodes usually have lower performance than customized disk arrays. To make up for the lower storage-node reliability and performance, data is strategically distributed to enable access parallelism and reliability in the face of node failures. A *data distribution* for a dataset describes how it is encoded (e.g., replication vs. erasure coding) and assigned to storage-nodes within the cluster.

There is no single data distribution that is best for all data. The data distribution choice has major impact on three crucial system metrics: availability, confidentiality, and performance. The data that a bank stores, for example, has different availability goals than the data of an online retailer [Keeton et al., 2004], and thus may require a different encoding. The online retailer may have a stricter confidentiality goal than an email provider and thus may require encryption. The online retailer may have more stringent performance requirements than the bank and may require that response times be kept below a threshold. Ursa Minor is versatile, in that it allows each client to use a different data distribution.

In face of system and workload heterogeneity and system scale, the tradeoffs involved when choosing the data distribution for a client or choosing how to upgrade the system, are complicated.

5.1.1 Challenges in predicting data distribution consequences

Predicting performance for a data distribution is a complex function of I/O workload characteristics and storage-node attributes. Selecting the right encoding requires knowledge of the access patterns and the bottleneck resources. For example, small random-access writes often interact poorly with erasure coding, but large streaming writes benefit from the reduced network bandwidth relative to replication. Data placement requires similar knowledge plus information about how workloads will interact when sharing storage-nodes. For example, two workloads that benefit from large caches may experience dramatic performance decreases if assigned to the same storage-node. Answering *What...if* questions about data distribution choices requires accounting for all such effects.

Predicting data encoding consequences: A data encoding specifies the degree of redundancy with which a piece of data is encoded, the manner in which redundancy is achieved, and whether or not the data is encrypted. Availability requirements dictate the degree of data redundancy. Redundancy is achieved by replicating or erasure coding the data [Chen et al., 1994; Rabin, 1989]. Most erasure coding schemes can be characterized by the parameters (m, n). An m-of-n scheme encodes data into n *fragments* such that reading any m of them reconstructs the original data. Figure 5.1 shows an example 3-of-5 encoding that tolerates up to two storage-node crashes. Other schemes that tolerate the same number of crashes include 1-of-3 replication, 4-of-6 and 10-of-12 erasure coding schemes. Confidentiality requirements dictate whether or not encryption is employed. Encryption is performed prior to encoding and decryption is performed after decoding. Once the availability and confidentiality requirements are known, we need to know the performance implications of picking any candidate encoding scheme.

There is a large trade-off space in choosing an encoding scheme. Figure 5.2 illustrates several of these tradeoffs. For example, as n increases, relative to m, data availability increases. However, the storage capacity consumed also increases, as does the network bandwidth required during data writes. As m increases, the encoding becomes more space-efficient; less storage capacity is required to provide a specific degree of data redundancy. However, availability decreases; more fragments are needed to reconstruct the data during reads. When encryption is used, the confidentiality of the data increases, but the demand on CPU increases (to encrypt the data).

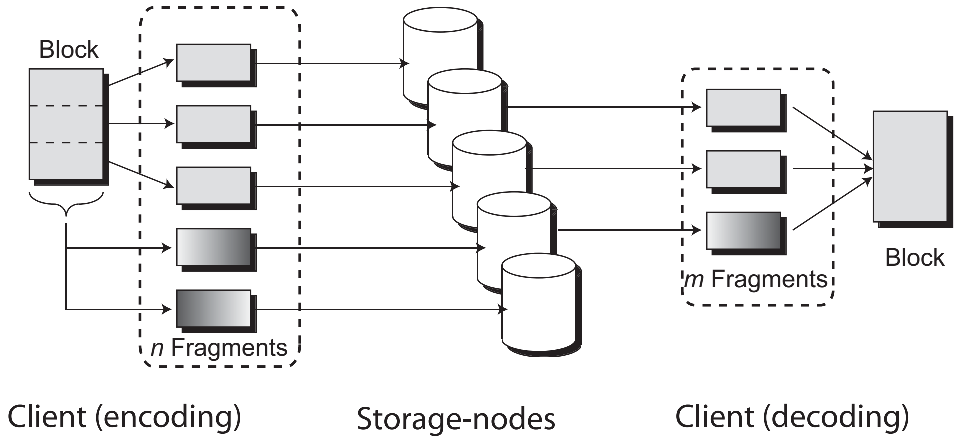
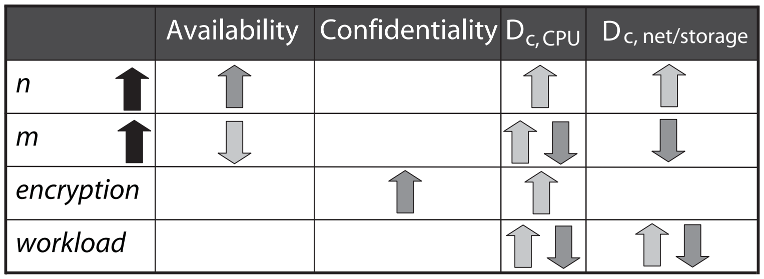


Figure 5.1. **Example 3-of-5 encoding scheme.** This diagram illustrates a client encoding a block of data using the 3-of-5 encoding scheme and writing to 5 storage-nodes. Another client is decoding the data from 3 storage nodes. 3-of-5 encoding tolerates loss of up to 2 storage-nodes.

The workload for a given piece of data should also be considered when selecting the data encoding. For example, it may make more sense to increase m for a write-mostly workload, so that less network bandwidth is consumed. As the evaluation section shows, 3-way replication (i.e., a 1-of-3 encoding) consumes approximately 40% more network bandwidth than a 3-of-5 erasure coding scheme for an all-write workload. For an all-read workload, however, both schemes consume the same network bandwidth. Others have explained these trade-offs in significant detail [Weatherspoon and Kubiatowicz, 2002; Wylie, 2005].

Because of this large trade-off space and the dependence on workload characteristics, it is very difficult for an administrator to predict the consequences of an encoding change — hence the need for system self-prediction. This chapter shows that Observer can answer high-level performance questions related to throughput and latency by answering and combining sub-questions of the form "*What* would be the CPU/network/storage demand of client c, *if* data is encoded using scheme E?".

Predicting data placement consequences: In addition to selecting the data encoding, the storage-nodes on which encoded data fragments are



	Availability	Confidentiality	$D_{c,CPU}$	$D_{c,net/storage}$
n ⬆	⬆		⇧	⇧
m ⬆	⇩		⇧⬇	⬇
encryption		⬆	⇧	
workload			⇧⬇	⇧⬇

Figure 5.2. **Example tradeoffs when choosing the encoding scheme.** The administrator currently has to understand the direction, size and importance of these arrows in order to pick the right encoding scheme.

placed must also be selected. When data is initially created, the question of placement must be answered. Afterwards, different system events may cause the placement decision to be revisited — for example, when new storage-nodes are added to the cluster, when old storage-nodes are retired, and when workloads have changed sufficiently to warrant re-balancing load. Quantifying the performance effect of adding or subtracting a workload from a set of storage-nodes is non-trivial. Each storage-node may have different physical characteristics (e.g., the amount of buffer cache, types of disks, and network connectivity) and may host different pieces of data whose workloads lead to different levels of contention for the physical resources.

Workload movement *What*...*if* questions (e.g., "*What* is the expected throughput/response client c can get *if* its workload is moved to a set of storage-nodes S?") need answers to several sub-questions. First, the buffer cache hit rate of the new workload and the existing workloads on those storage-nodes need to be evaluated (i.e., for each of the workloads the question is "*What* is the buffer cache hit rate *if* I add/subtract client c's workload to/from this storage-node?"). The answer to this question will depend on the particulars of the buffer cache management algorithm the storage-node uses. Second, the disk demand (or service time) for each of the I/O workloads' requests that miss in buffer cache will need to be predicted (i.e., for each of the workloads, the question is "*What* is the average I/O service time *if* I add/subtract client c's workload to/from this storage-

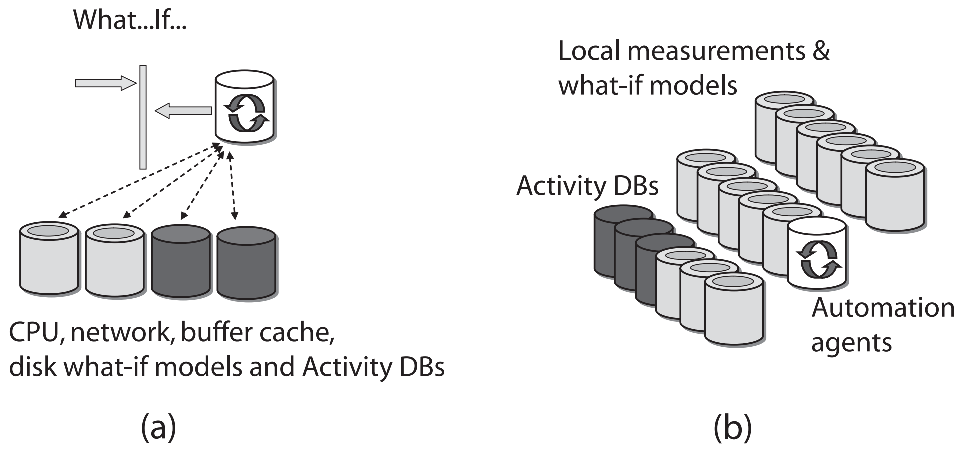

What...If...
CPU, network, buffer cache, disk what-if models and Activity DBs
(a)
Local measurements & what-if models
Activity DBs
Automation agents
(b)

Figure 5.3. **Automation agents and low-level *What*...*if* models.** A logical and physical view of the modeling and monitoring infrastructure that is weaved within the system. Automation agents receive *What*...*if* questions and answer them by consulting and combining resource-specific *What*...*if* models.

node?"). Third, the network load on each of the storage-nodes that results from adding/subtracting workloads needs to be predicted as well.

It is challenging for administrators to answer *What*...*if* questions such as the above. Doing so requires one to understand the system internals (e.g., buffer cache replacement policies) and to keep track of the workloads each resource is seeing (e.g., buffer cache records for each workload and storage-node).

5.2 Incorporating expectation-based models in Ursa Minor

This section describes the integration of parts of Observer (the expectation-based models) in Ursa Minor. Figure 5.3 shows two views of the way the models are incorporated in Ursa Minor. The first (a) is a logical view. The administrator (or internal tuning modules) interacts with *automation agents* through high-level *What*...*if* questions. The automation agents have an understanding of the underlying queuing network in the system and solve the necessary queuing equations to answer the *What*...*if* questions. At the ser-

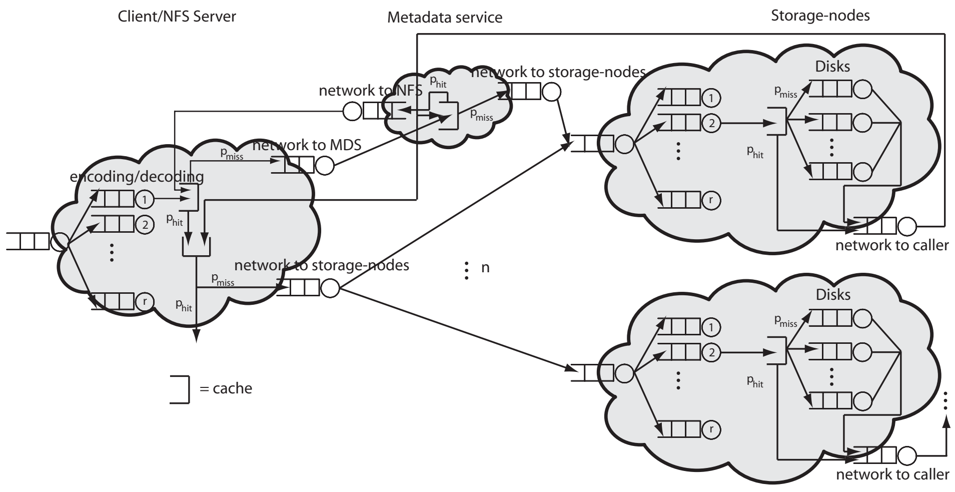


Figure 5.4. **Simplified queuing network in Ursa Minor.** This queuing diagram shows the path for writes, from a single client. The clouds represent major software components in Ursa Minor.

vice center level, each service center answers its own set of low-level *What...if* questions and keeps its own statistics in Activity DBs, as first described in Chapter 4. The second (b) is a view of the physical system. Each server in the cluster, in addition to hosting specific services, monitors itself and exposes a set of low-level *What...if* questions.

An important message to take away is that Observer and Stardust are built into Ursa Minor (although most of the modeling analysis is currently done offline). Each server has dedicated resources (CPU, memory, network, disk) to host this infrastructure.

5.2.1 Queuing network structure

Figure 5.4 shows a simplified queuing network (only for write requests) inside Ursa Minor, mirroring the system diagram shown in Figure 4.4. 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 small fraction of the energy already spent verifying the correctness of the various algorithms and protocols in the system. Once the correctness of the protocols has been verified, the “extra work” needed to codify expectations in the form of a queuing network is minimal. Concretely, several months were spent by several people designing and reviewing how requests would flow in Ursa Minor, whereas translating that previous work into a queuing network took me less than a month.

5.2.2 Performance expectations

This section describes the main service center types in Ursa Minor and how each of them uses built-in models to answer specific What...if questions.

CPU encode/decode What... if model: direct measurement

The goal of the client CPU model[1] is to answer sub-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. Inputs to the CPU model are the hypothetical encoding E and the measured read:write ratio of the workload. 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 (Chapter 6 shows how the answer can be remembered after the first time, however, the rationale for doing so will be deferred to that chapter).

As Section 6.4.3 elaborates[2], a secondary CPU-consuming service center, which we initially did not consider, models the CPU consumed by TCP network stack processing inside the kernel. There are regions of the workload-system operation space for which CPU consumed by the network stack is

[1]There is CPU consumed at the storage-nodes and MDS as well (e.g., during data copying). However, the storage-node and MDS CPUs do not become a bottleneck in practice, so we focus on the client CPU, which is used for encoding/decoding and encryption.

[2]Section 6.4.3 describes how we initially discovered that we were not modeling the CPU consumption correctly.

non-negligible. 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 [Iperf, 2007] generates network-bound workloads and is run with various request sizes to record the CPU consumption incurred.

Network What...if model: analytical model

The goal of the network model is to answer sub-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?". To capture first-order effects, the network model uses a simple analytical function to predict network demand based on the number of bytes transmitted. 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 network demand for a single request is the minimum time needed to transmit the data for a request (i.e., if that request was the only one using the network) plus a fixed cost for the network stack processing. The time to transmit data equals the amount of data transmitted divided by the network bandwidth. 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:

$$\text{Bytes sent} = p_{read} BlockSize + p_{write} BlockSize \frac{n}{m} \quad (5.1)$$

$$Time = \frac{\text{Bytes sent}}{Bandwidth} + \text{network setup time} \quad (5.2)$$

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 [He et al., 2005]. However, the above analytical model is a good starting point, and it can be refined by the observation-based models (e.g., the network setup time may be server and switch dependent).

Buffer Cache What...if model: simulation-based model

The goal of a general buffer cache model is to answer sub-questions of the form "*What* is the probability $p_{c,HIT}$ that requests from client c are absorbed[3] in cache, *if* the cache size is X MB?" As shown in Figure 5.4, there are caches at many points in the system. In this dissertation, we focus the discussion on the buffer cache at the storage-nodes.

A disk access requires orders of magnitude more time than a buffer cache access, hence the performance that the client sees is very much dependent on the storage-node's buffer cache. The buffer cache behavior of a workload depends on its access patterns, working set size, and the cache size and replacement policy.

The buffer cache model uses simulation to make a prediction. 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 (as further described in Section 5.3.4). Hence, the analysis above can be done independently for every workload W_i, without considering how other workloads may interfere with it (because the interference is explicitly disallowed — there are potential benefits of interference, such as sharing cache blocks, but, in practice, sharing happens rarely.) The *What...if* model computes, for each workload, a *cache curve* that indicates the cache absorption rate as a function of the cache size.

The output from this model is the absorption rate and a trace of requests that have to go to disk for each workload. Simulation is used, rather than an analytical model, because buffer cache replacement and persistence policies are often complex and system-dependent. In Ursa Minor, they cannot be accurately captured using analytical formulae. The storage-node buffer cache

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

policy in Ursa Minor is a variant of least-recently-used (LRU) with certain optimizations.

Disk What. . . if model: simulation-based and analytical model

The goal of the disk model is to answer sub-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 NVRAM, use a log-structured layout on disk [Soules et al., 2003], and prefetch aggressively [Wachs et al., 2007]. Request scheduling is round-robin across each workload, where the length of the scheduling quanta for each round is determined analytically as described by Wachs et al. [2007]. The quanta is determined using a combination of the very buffer cache and disk models this section describes; a sketch of the approach is given in Section 5.3.4, which also sketches the approach taken to determine how deep prefetching must be.

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. The service time is based on the four extracted disk parameters, the length of the scheduling quanta, and the prefetching policy. Within a quanta, each request gets a service time as follows. Let RUN be an ordered set of requests that are consecutively sequential (reads or writes respectively), and let $RUN[i]$ be the i^{th} request in the run. Then, within a quanta Q_c:

$$D^{'}_{c^i,DISK} = \begin{cases} SEQ_{read} & \text{if } c^i \in \text{read RUN} \wedge c^i \neq \text{RUN[1]} \\ SEQ_{write} & \text{if } c^i \in \text{write RUN} \wedge c^i \neq \text{RUN[1]} \\ RND_{read} & \text{if } c^i = \text{read RUN[1]} \vee c^i \not\subseteq \text{RUN} \\ RND_{write} & \text{if } c^i = \text{write RUN[1]} \vee c^i \not\subseteq \text{RUN} \end{cases} \tag{5.3}$$

Thus, when considering the quanta length $|Q_c|$ for each of the clients, the expected service demand for each client request, assuming request from all clients arrive uniformly distributed during the quanta times, is:

$$D_{c^i,DISK} = \left(\frac{|Q_c|}{\sum_j |Q_j|}\right) D^{'}_{c^i,DISK} + \left(\frac{\sum_{j\neq c} |Q_j|}{\sum_j |Q_j|}\right)\left(\frac{\sum_{j\neq c} |Q_j|}{2}\right) \tag{5.4}$$

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 thesis, and is explained by Wachs et al. [2007]. The mechanism is sketched in Section 5.3.4.

5.2.3 Answering high-level What. . . if questions

To predict client c's throughput after a data distribution change or resource upgrade, the automation agents solve a queuing network by consulting the resource-specific *What...if* models to determine which of the resources will be the bottleneck resource. Client c's peak throughput will be limited by the throughput of that resource. As first described in Section 3.3.1, a tool like the one provided by Gunther [2005], provides exact throughput and response time solutions.

For example, a workload movement *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 to several sub-questions. First, the buffer cache hit rate of the new workload and the existing workloads on those storage-nodes need to be evaluated by the buffer cache model. Second, the disk service time for each of the I/O workloads' requests that miss in buffer cache will need to be predicted by the disk model. Third, the network load on each of the storage-nodes that results from adding/subtracting workloads needs to be predicted by the network model. Using the throughput bounding formula in Table 3.2, bounds can be provided on peak throughput that each workload could achieve if sharing the storage-nodes S.

5.3 Evaluation

This section evaluates the use of expectation-based queuing models in answering several *What*...*if* questions on data distribution changes and component upgrades. Initially experiments are shown to evaluate the individual *What*...*if* models in isolation. Then, the complexity of the experimental setup increases gradually and multiple *What*...*if* models are used to answer high-level *What*...*if* questions.

5.3.1 Experimental setup

The experimental setup and workloads used are identical to the ones first described in Section 4.4.1. In addition to the benchmarks described there, we use the following benchmark in this section:

SSIO_BENCHMARK: This micro-benchmark allows control of the workload read:write ratio, access patterns, and number of outstanding requests. Performance is reported in terms of requests/second, MB/s, and response time per request. The access size is 32 KB for this benchmark. The number of outstanding requests is $N_c = 8$ and the file size is 32 MB, unless otherwise mentioned. This workload is closed-loop.

For conciseness, we present how Observer chooses the right data distribution by using six data encodings. These results are indicative of the many other encodings we explored. "1-of-1" refers to 1-way replication. "1-of-1 encr" is 1-way replication where the data is also encrypted to ensure confidentiality. For encryption, we use the AES cipher with a key size of 128 bits in CBC mode. "1-of-3" is 3-way replication, which tolerates two storage-node crashes. "1-of-3 encr" is 3-way replication with encryption. "3-of-5" is an erasure coding scheme that tolerates two storage-node crashes, but is more storage efficient than “1-of-3”. “3-of-5 encr” is the “3-of-5” scheme with encryption.

Unless otherwise mentioned, all experiments are run ten times, and the average and the standard deviation are reported. The average client think time Z_c is zero in all experiments.

5.3.2 Individual resource models

This section evaluates the resource-specific *What*...*if* models in isolation. The CPU and network *What*...*if* models are based on direct measurements, hence the prediction accuracy is usually perfect. However, we defer to Section 6.4 discussion of how surprising, non-linear behavior for both resources manifests itself. For those two resources, we just show how resource consumption changes as a function of encoding choice. The memory and disk *What*...*if* models are based on simulation and analytical models, respectively, and so we concentrate on the prediction accuracy of those models.

CPU *What*...*if* model: Recall from Section 5.2.2 that the goal of the CPU model is to 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?”. Figure 5.5 shows how the CPU demand varies based on the encoding scheme used. The model runs 100 encode/decode operations and reports the average. Some encoding schemes differ from others in CPU demand by more than an order of magnitude, and as we show later in this evaluation, the client CPU can become a bottleneck. Two trends are worth noting. First, the cost of encryption dominates the cost of data encoding/decoding. Second, erasure coding requires more CPU than replication for encoding data.

Network *What*...*if* model: Recall from Section 5.2.2 that the goal of the network model is to answer sub-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?”. Figure 5.6 shows how the network demand varies based on the encoding scheme used. A trend worth noting is that replication places a larger demand on the network than erasure coding, for the same number

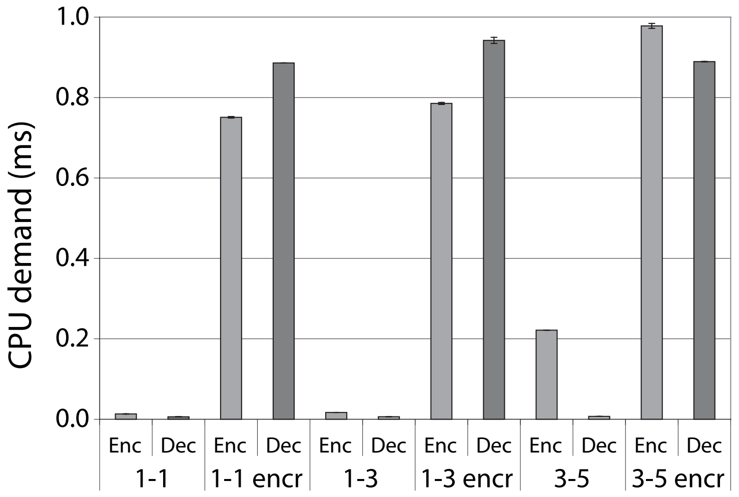


Figure 5.5. **CPU *What*...*if* model output.** This figure illustrates how the CPU demand per request differs based on the chosen data encoding. The cost for encoding data (during a write) and decoding it (during a read) are shown for six encoding choices.

of storage-node crashes tolerated. We defer discussion of variance in network demand to Section 6.4.3, because it is surprisingly specific to the network switch type.

Buffer Cache *What*...*if* model: Recall from Section 5.2.2 that the goal of the buffer cache model is to answer sub-questions of the form "*What* is the probability $p_{c,HIT}$ that requests from client c are absorbed in cache, *if* the cache size is X MB?" Figure 5.7 illustrates the accuracy of the buffer cache model for predicting read hits under three workloads of varying working-set size and access patterns. The encoding for these workloads is 1-of-1. This experiment illustrates what would happen if, for example, another workload was added to the storage-node such that the amount of buffer cache available to the original one shrank. Or, if a workload was removed from the storage-node, and the amount of buffer cache available to the original one increased.

For each of the workloads, Stardust collected the original buffer cache reference trace when the buffer cache size was 512 MB, and the *What*...*if* model predicted what would happen for all other buffer cache sizes. (The

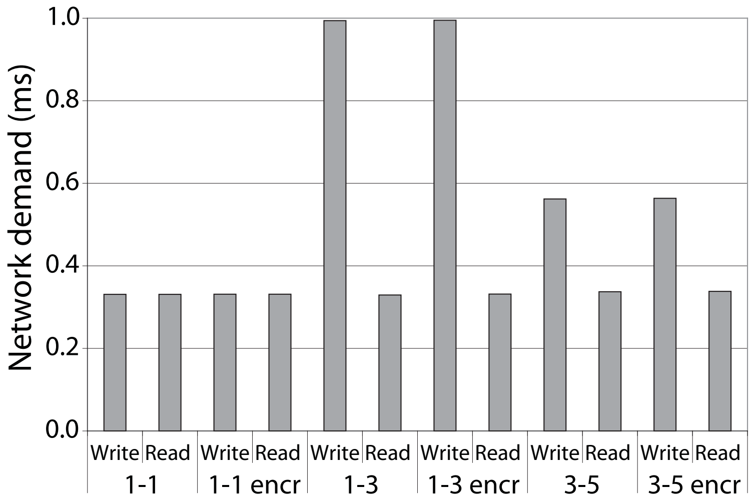


Figure 5.6. **Network *What*...*if* model output.** This figure illustrates the network demand per request as a function of the chosen data encoding.

choice of 512 MB is rather arbitrary, but we have verified that any other size in the range shown gives similar results).

An important metric for evaluating the efficiency of the buffer cache *What*...*if* model is the simulator's throughput, in terms of requests that can be simulated per second. We have observed that for cache hits the simulator and real cache manager need similar times to process a request. The simulator is 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.

Disk *What*...*if* model: Recall from Section 5.2.2 that the goal of the disk model is to answer sub-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?" Figure 5.8 illustrates the accuracy of the disk model. The buffer cache model produces a disk reference trace (for requests that miss in buffer cache). The disk model takes those requests, analyzes their access patterns, and predicts individual request service times. The model captures

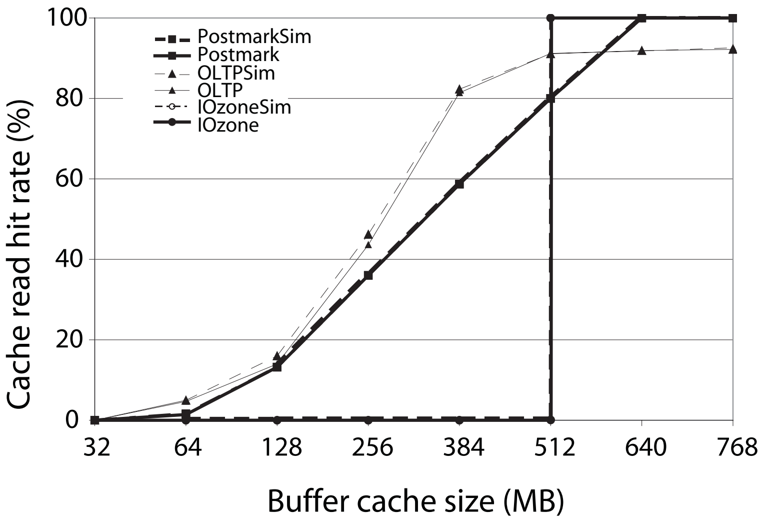


Figure 5.7. **Buffer Cache *What...if* model output.** This figure illustrates the accuracy of the buffer cache simulator in predicting the storage-node buffer cache read hit rate under various workloads. For Postmark and IOzone, the measured and predicted hit rate are almost indistinguishable, indicating excellent prediction accuracy. Error bars in all cases are invisible, since the variance is negligible.

well the service time trends, but there is room for improvement, as seen in the Postmark case. The rather large inaccuracy at the 512 MB buffer cache size shows a limitation of expectation-based models. It occurs because more requests are hitting in the buffer cache, and the few requests that go to disk are serviced in FIFO fashion, thereby reducing the efficiency of the disk head scheduler. Recall, from Section 5.2.2, that the disk model is built using four parameters extracted from the disk: random-access read and write service times and streaming read and write service times. The reported service times usually assume that the disk queue is full, which allows for more efficient disk scheduling. Thus, our disk model also ends up assuming that the disk queues are always full when making a prediction. That leads to inaccuracies when the disk queue is not full, as is the case for Postmark at 512 MB.

In general, predicting the size of the disk queue requires assumptions about arrival patterns (e.g., Poisson arrivals) that we do not wish to make.

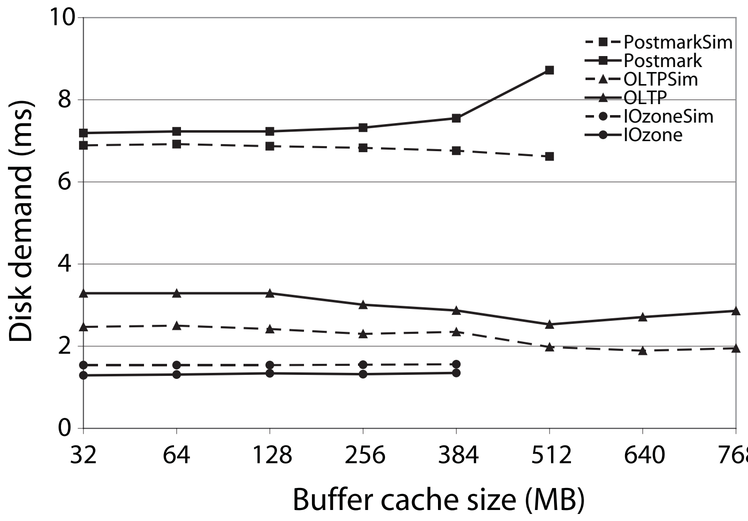


Figure 5.8. **Disk *What...if* model output.** This figure illustrates the accuracy of the disk model in predicting request service times for several workloads with different access patterns. Error bars in all cases are invisible, since the variance is negligible.

However, as described in Section 6.3, statistical observation-based models may be able to specialize their predictions by "discovering" correlations between attributes that the designer could not build expectation-based models for (like queue size) and performance.

5.3.3 Answering high-level What...if questions

This subsection evaluates the accuracy of the automation agents in predicting throughput and response time using several of the *What...if* models in combination.

Predicting cost of encryption: The first experiment is illustrated in Figure 5.9. The high-level performance question that this experiment answers is "*What* is the peak throughput client c can get *if* its workload's encoding changes from 3-way replication to 3-way replication with encryption (or the other way around)?". The workload hits fully in the storage-nodes' buffer caches, and, hence, the two resource types that could be a bottleneck

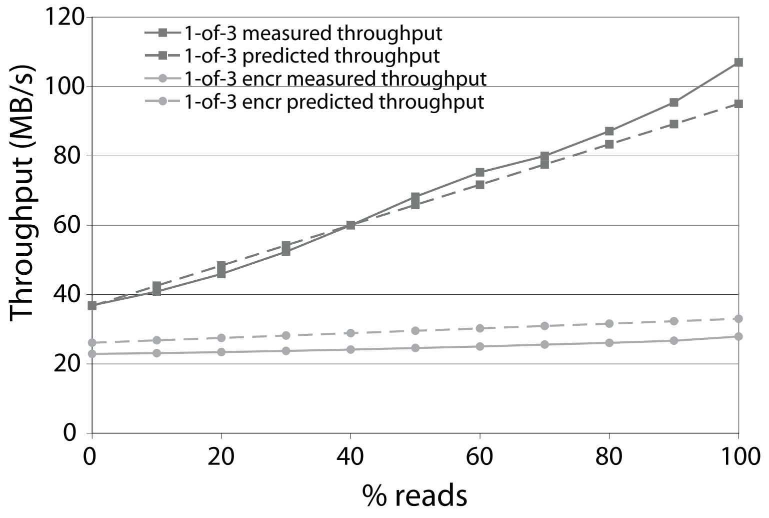


Figure 5.9. **Predicting peak throughput for CPU-bound workloads.** SSIO_BENCHMARK is used to measure throughput for different read:write ratios. The highest standard deviation seen among all measured data points was 2 MB/s.

are the CPU and network resources. For each of the predicted datapoints in the graph, the CPU and network model report on the demands placed on those resources respectively. The automation agents then use the throughput bounding formula in Table 3.2 to determine the peak throughput resulting from the bottleneck resource.

There are several trends worth noting. First, the predictions track well the actual throughput lines. Second, when using encryption, the client's CPU is the bottleneck resource. Third, although the CPU cost of encoding is higher than that of decoding when using encryption, the throughput increases slightly as the read percentage increases. This is because writes are sent to three machines, thus requiring more network bandwidth than reads. As described in Section 5.2.2, the higher the network bandwidth required, the higher the CPU demand needed for TCP processing. Thus, less CPU time is available for the encoding and encryption of data. Fourth, as the read percentage increases, the throughput for the encoding without encryption increases, since reads obtain data from only one of the storage-nodes.

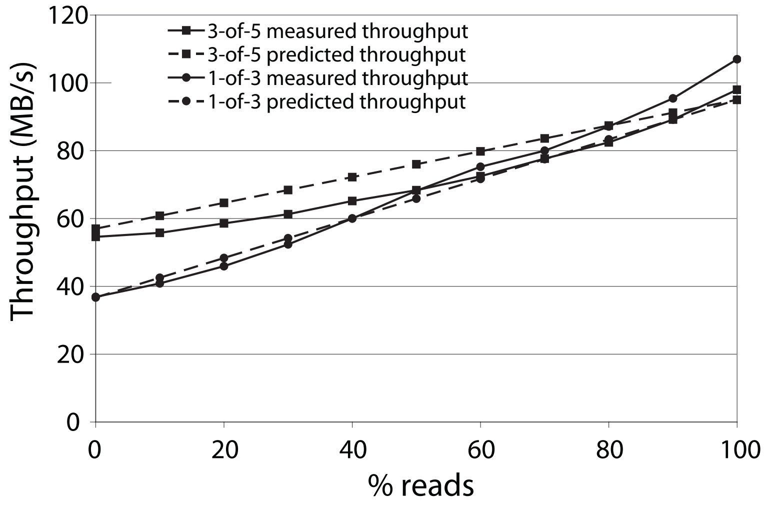


Figure 5.10. **Predicting peak throughput for network-bound workloads.** SSIO_BENCHMARK is used to measure throughput for different read:write ratios. The highest standard deviation seen among all measured data points was 1.75 MB/s.

Writes, instead, need to update all three storage-nodes, thus placing more load on the network and CPU.

Replication vs. erasure codes: The second experiment is illustrated in Figure 5.10. The high-level performance question that this experiment answers is "*What* is the peak throughput client c can get *if* its workload's encoding changes from 3-way replication to a 3-of-5 erasure coding scheme (or the other way around)?". A 3-of-5 scheme is more storage efficient than 3-way replication, while tolerating the same number of storage-node crashes (two).

The workload hits fully in the storage-nodes' buffer caches, and, hence, the two resource types that could be a bottleneck are the CPU and network resources. For each of the predicted datapoints in the graph, the CPU and network model report on the demands placed on those resources respectively. The automation agents then use the throughput bounding formula in Table 3.2 to determine the peak throughput resulting from the bottleneck resource.

A trend worth noting is that, for a mostly-write workload, the 3-of-5 encoding performs best, since the workloads are network bound. For 3-way replication the amount of data that needs to be transmitted to tolerate two crashes is three times more than the data that needs to be transmitted when no crashes are tolerated. But, the 3-of-5 scheme only transmits $\frac{5}{3}$ times more data. Hence, the network demand is less for that scheme.

Data placement: The next experiment answers the question "*What* is the peak throughput client A can get *if* its workload is moved to a new set of storage-nodes?". Client A's workload is encoded using 3-way replication. Two sets of possible storage-nodes are considered for data placement. The first set S_1 is currently not utilized. However, one of the nodes is behind a 100 Mbps network (the other nodes are behind a 1000 Mbps, or gigabit, network). The second set S_2 currently services a second workload, and Stardust measures a utilization of 50% on the network of the set S_2 nodes.

The workload hits fully in the storage-nodes' buffer caches, and, hence, the two resource types that could be a bottleneck are the CPU and network resources. For each of the two choices, the CPU and network model report on the demands placed on those resources respectively. The automation agents then use bottleneck analysis to determine the peak throughput resulting from the bottleneck resource. Figure 5.11 shows the accuracy of the predicted performance.

Effects of N_c on throughput and response time: The next experiment answers the question "*What* is the distribution of throughput and response time *if* the number of outstanding requests N_c from client c changes?". It is intuitive that the client's throughput will peak after a certain number of outstanding requests, while the response time may continue to increase after that point as more requests are queued. Our models quantify the change in both metrics.

Figure 5.12 illustrates the prediction accuracy for a client that is using the 3-of-5 scheme and is network-bound. After the request pipeline fills up ($N_c^* = 3$), the throughput peaks, while the response time increases linearly as the formulae in Section 3.3.1 predict.

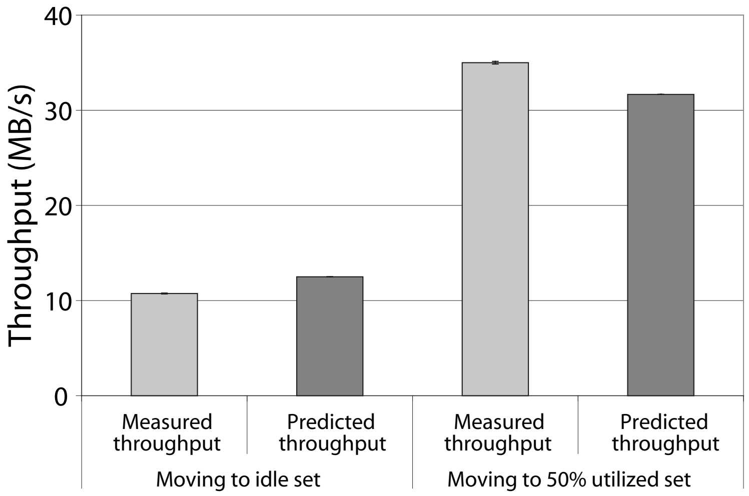


Figure 5.11. **Predicting peak throughput for workload movements.** The high-level performance question that this experiment answers is "*What* is the peak throughput client A can get *if* its workload is moved to a new set of storage-nodes?" In this experiment, the first set of storage-nodes is not utilized, but one of the storage-nodes in that set is behind a slow network. The second set of storage-nodes contains a workload that places a 50% utilization on the network. Both SSIO_BENCHMARK workloads consist entirely of writes.

5.3.4 Modeling scalability and complexity

This next experiment illustrates the implications of having no support in the system for separability of analysis when analyzing the performance of workloads. Lack of this property is a show-stopper for queuing analysis, as first discussed in Section 3.3.3. Without separability of analysis, each workload's prediction needs to be re-evaluated each time another workload is added by considering every possible way requests from those workloads might interfere. Such re-evaluation is expensive (sometimes prohibitively so, such as in the case of the buffer cache simulator. It would have to be invoked during each re-evaluation step to examine *all possible interleavings* among workload requests). Often, to sidestep the need to re-evaluate all possible interleavings, either worst-case predictions are made that assume

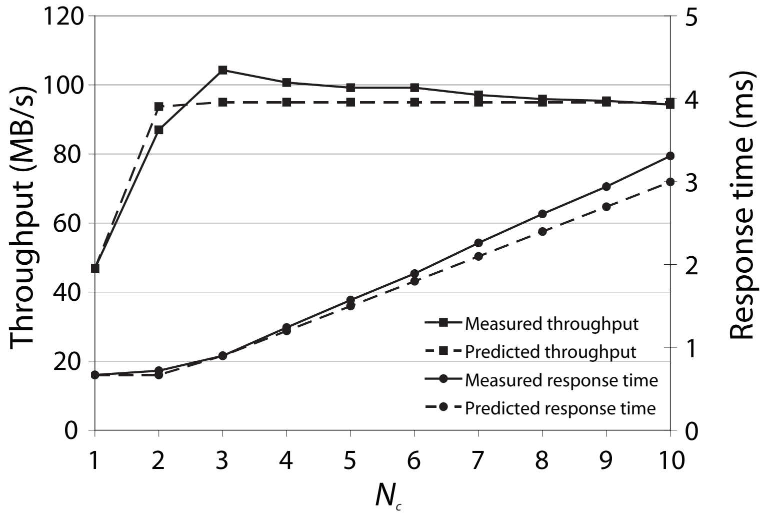


Figure 5.12. **Predicting throughput and response time as a function of N_c.** The high-level performance question that this experiment answers is "*What* is the distribution of throughput and response time *if* the number of outstanding requests N_c from client c changes?" SSIO_BENCHMARK is used to control the number of outstanding requests. The workload consists entirely of reads. The highest standard deviation seen among all measured throughput data points was 2.2 MB/s, and among measured response time data points was 0.1 ms.

requests always interfere in the worst possible way with one another, or only a few interleavings possibilities are examined. For example, for a disk, worst-case interleaving happens when two requests require data that resides furthest apart in terms of disk seek distance and rotational distance. It is clear that, in storage-systems, worst-case predictions can be one to two orders of magnitude worse than best-case predictions.

A mechanism that ensures performance insulation ensures, among other things, that once a workload gets a "share" of a resource, no other workload can steal that share away ("share" is used loosely here. It could be a scheduling quanta for the CPU, network and disk resources, or a fraction of the buffer cache size).

The experiment in this section is also concerned with the data placement question "*What* is the peak throughput client B can get *if* its workload

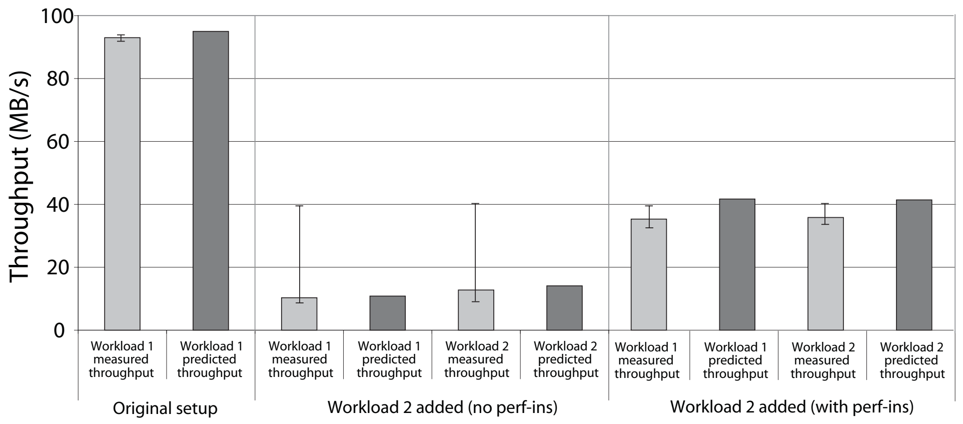


Figure 5.13. **Predicting peak throughput from workload interference.** The high-level performance question that this experiment answers is "*What* is the peak throughput client A can get *if* its workload is moved to a new set of storage-nodes?" SSIO_BENCHMARK is used to control the workload access patterns. The error bars indicate the 95% percentile of throughput.

is moved to a new set of storage-nodes?". The encoding is again 3-way replication but there are several setup differences. The first set of nodes S_1 is currently being used by a streaming workload A that hits in the buffer cache of the storage-nodes. Workload B is also a streaming workload, and the administrator wants to know the performance implication of moving that workload to the S_1 set of storage-nodes. Figure 5.13 shows the results.

For this particular experiment, the models have two options. The first option is to predict worst-case behavior. The models would assume all requests miss in cache and all requests incur a large seek and rotational time from the disk. The second option, shown in Figure 5.13, is to consider only one of the many interleaving options and predict for that interleaving. Both options are unsatisfactory, but it would be prohibitively expensive to evaluate the buffer cache and disk models for all request interleavings.

How performance insulation helps to keep models simple: Rather than going with the above options, we chose to change the system algorithms to reduce the interference to start with. Our solution, performance insulation, hinted at throughout Section 5.2.2 as key to making the *What...if* models work, was developed in Ursa Minor by Wachs et al. [2007]. The complete work on making the necessary system changes to enable performance insulation is not a direct contribution of this dissertation. However, the models developed in this dissertation were the underlying technology to help with performance insulation; in turn, performance insulation allowed the models to stay simple, by reducing complexity from uncontrolled interference in the system. Below, we sketch how the models helped the system isolate the workloads from one another at the storage-nodes.[4] The interested reader is advised to consult [Wachs et al., 2007] for the full details.

Disks involve mechanical motion in servicing requests, and moving a disk head from one region to another is slow. The worst-case scenario is when two sequential access patterns become tightly interleaved causing the disk head to bounce between two regions of the disk; performance goes from streaming disk bandwidth to that of a random-access workload. Likewise, cache misses are two orders of magnitude less efficient than cache hits. Without proper cache partitioning, it is easy for one data-intensive service to dominate the cache with a large footprint, significantly reducing the hit rates of other services. Two consequences of disk and cache interference are significant performance degradation and lack of performance predictability (e.g., to predict the performance of a streaming workload, one must analyze all possible ways in which other workloads may interfere with it). As a result, interference concerns compel many administrators to statically partition storage infrastructures among services.

Our implementation of performance insulation combines three mechanisms plus automated configuration to achieve the above goal. First, detecting sequential streams and using sufficiently large prefetching/write-back ranges amortizes positioning costs to achieve the configured efficiency value of streaming bandwidth. The disk model helps with automatically determining the prefetch size (e.g., it answers a series of *What...if* questions on how

[4]Others have described the mechanisms needed to insulate performance for other resources like CPU and network [Banga et al., 1998; Reumann et al., 2000].

performance would change as a function of the prefetch size and chooses the best answer).

Second, explicit cache partitioning prevents any one service from squeezing out others. To maximize the value of available cache space, the space allocated to each service is set to the minimum amount required to achieve the configured efficiency. For example, a service that streams large files and exhibits no reuse hits only requires enough cache space to buffer its prefetched data. The cache model performs on-line cache simulation to determine the required cache space (e.g., it answers a series of *What...if* questions on how performance would change as a function of the cache size and chooses the best answer).

Third, disk time scheduling quanta are used to separate the disk I/O of services, eliminating interference that arises from workload mixing. The length of each quantum is determined analytically by a combination of the cache and disk models.

The combination of cache partitioning, deep prefetching, and quanta-based disk scheduling leads to workloads being sufficiently insulated from one another. The current policy in Ursa Minor allows the performance of each of n workloads to be analyzed independently and provides soft guarantees that whenever the n workloads are combined, each gets $1/n^{th}$ of their stand-alone throughput. The models help with automatically choosing the right cache size, prefetching depth, and scheduling quanta.

Going back to Figure 5.13, each of the workloads is analyzed independently and assigned a cache size, prefetching size and scheduling quanta so that the efficiency when combined would be close to 1/2 of their original efficiency[5]. In this particular setup, the variance in performance is also reduced, however, performance insulation does not necessarily reduce variance at all times, as explained by Wachs et al. [2007].

[5] $1/n$ is the ideal efficiency. As Wachs et al. [2007] discusses, there are several reasons why the theoretical ideal may be slightly higher than what is practically possible to achieve.

5.4 Summary, lessons learned, and limitations

This chapter illustrated the usefulness of Observer, the modeling infrastructure, when answering *What*...*if* questions related to data distributions and resource upgrades in Ursa Minor. The first part of Observer (the expectation-based models) made accurate predictions in common regions of operation. The expectation-based models are treated as first-class citizens in Ursa Minor.

A main lesson learned in this chapter is that, if the models appear to get too complex, it could be time to look at the system algorithms and simplify them. Subtle changes in the system algorithms can make a large difference in inherent system predictability. In particular, performance insulation in Ursa Minor allowed the behavior of the individual workloads to be more predictable, by allowing each workload to be analyzed independently. In turn, that allowed the models to remain simple, and still predict well.

An interesting area worth exploring in the future is extending the *What*...*if* models to software components. For example, it would be interesting to examine whether questions of the form "*What* happens to performance *if* I upgrade a software component (e.g., hash algorithm)?" can be answered. Currently the models only predict the effects of hardware upgrades.

A discussion of other limitations of expectation-based models, together with ways to address some of them, is deferred to Chapter 6.

6 Towards robust modeling for the unexpected

Our experiences to date with using the expectation-based models presented in Chapter 3 and Chapter 5 have been positive. However, there have been times when the models proved to be too brittle and failed to predict well. From our initial experience, we found several reasons for being cautious with the initial models. These reasons are illustrated in Figure 6.1.

Misconfigured or buggy components: Systems may be designed and implemented well, but could be misconfigured when deployed. Also, sometimes systems are designed well, but the implementation may not perform as expected in certain circumstances. In these cases, the models' predictions, which contain simplified design invariants, may be different from the system's behavior. Misconfigurations and performance bugs that we have seen include DB table placement misconfigurations causing excessive locking, buffer cache misallocations, routing errors, incorrect implementation of disk drivers that leads to poor performance, etc.

Limited or incorrect behavioral models: The models I initially built were not flawless either. In general, most individual resource models 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 non-deterministic behavior that can only be statistically modeled. In addition, incomplete understanding by the model creator on how the system behaves under certain conditions can lead to incorrect behavioral models (e.g., behavior of a RAID

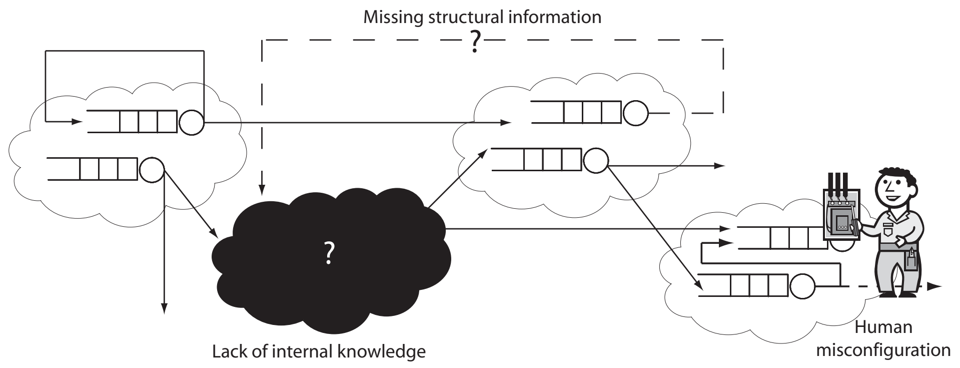


Figure 6.1. **Brittleness of expectation-based models.** In practice, some system components may be buggy or misconfigured, models may have limited regions of operation, and some system components may be black-box and not expose enough measurements.

array when one of the disks has failed is more difficult to model than the non-failure behavior).

Limited inputs to the models: We had complete access to the source code of Ursa Minor, hence Stardust, as Chapter 4 described, was able to provide the models with good measurements from it. Even in Ursa Minor, however, Stardust only monitors the service centers along the critical path of requests; it does not monitor the many background maintenance activities, such as the buffer cache syncer daemon, garbage collection thread(s), etc. This reflected my perception that time is saved by monitoring only the critical path of requests. In many scenarios, maintenance activities do not change the predictions significantly; however, when they did, we did not have enough information to find the root cause of the problem. Ursa Minor also makes use of legacy components (e.g., the Linux kernel), which we treat as black-box. These legacy components do not provide end-to-end measurements to the mathematical tools.

This chapter describes a second (logical and physical) component to the modeling architecture presented in Chapter 3: observation-based models. Observation-based models are a step towards robust and meaningful predictions in the face of the above issues. First, such models can help localize the potential sources of a prediction mismatch by verifying system behavior. Localization can guide humans towards the root-cause of the problem. Second, in many cases, the models can re-train while deployed to adjust to unexpected workload-system attributes. Third, observation-based models can give fidelity estimates by keeping track of historical information about their predictions. System-workload regions of operation where the models being used have frequently mispredicted will have low fidelity.

6.1 Mismatch localization based on resource consumption

We believe that a robust model should detect when there is a deviation between the model expectations and observed behavior. Thus, in addition to answering hypothetical *What...if* questions, individual models should continuously self-check. There are two general types of deviations:

Structural deviations: Structural expectation mismatches happen when designer expectations do not cover all possible ways requests can flow through service centers (or when the designer misconfigures the service center topology). Concrete examples of when structural deviations might happen in a storage system include: re-entering a service center a second time because a sub-request timed out the first time, entering a data repair phase and having storage-nodes interact differently with one another, going to the Metadata Service several times because of a poor implementation of an access control protocol, etc.

For white-box systems, a starting point towards detecting structural deviations is having a good measurement infrastructure in place that keeps track of the requests at entry and exit points from each service center in the system. Stardust does just that. A causal graph similar to the one shown in Figure 4.12 can be created for every request. That graph can then be "diffed" with the graph of structural expectation that the designer first inputs.

The situation when few or no structural expectations exist is a special case of the above. In this case, initial expectations may be "bootstrapped" by obtaining the causal path of requests from the initial workloads running in the system.

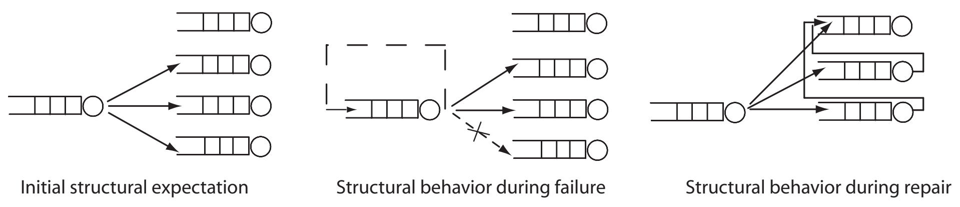

Initial structural expectation
Structural behavior during failure
Structural behavior during repair

Figure 6.2. **Structural deviations during failure and repair for 3-way replication.** The initial structural expectation for write requests indicates that a write contacts 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.

Structural deviations *within* black-box components cannot be obtained by the above method. A black-box component is treated as a large service center. The system monitors requests arriving and departing from this service center and only detects deviations *external* to the black-box component.

Performance deviations: Performance deviations occur when the initial models have limited regions of operation (e.g., there could be non-linear behavior that the designers did not foresee or the system could be configured in an unexpected way). Concrete examples in a storage system might include performance drops when striping data over more than five storage-nodes due to switch buffer congestion, misconfiguration of a hash table algorithm that hashes files using only the low bits of their ID (which, over time, leads to performance degradation), and an adaptive application that continuously changes its behavior depending on the speed of the system.

A starting point towards detecting performance deviations is to keep track of historical data while deployed in the field. The historical data should be kept for each service center to indicate how well it has been predicting in the past. For service centers for which there are expectation-based models, the system should compare the expectations with the historical data. For service centers for which there are no expectation-based models, the system could compare new observations with a historical average. For example, a metric could be considered to deviate if its value is outside the one-standard-deviation region (other policies could be used as well). Much previous work in the general field of anomaly detection (e.g., [Cohen et al., 2005; Sun et al., 2005]) can be used to analyze the strength of the correlation. Our goal is to investigate how we can use performance models to establish a state of "normalcy" that subsequent behavior can be compared against.

From a feasibility perspective, there are challenges. First, the self-checking must be managed in terms of when it should happen and how frequently. For example, self-checking might happen online when a service level objective is not met or offline during idle time (e.g., nightly). 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 [Moore, 2005], it may be justified to throw money at dedicated hardware for self-checking, but the costs need to be examined.

6.2 Model refinement and fidelity

After a successful mismatch localization, a relearning component should be applied to evolve the existing models or at least making educated suggestions to the model designers. Any such component must address two issues. First, it should discover new attributes of the workload-system interaction space that should be incorporated into the model. Second, it should discover regions of operation where the prediction confidence is high and regions where it is low. Hence, any subsequent model outputs can have a notion of confidence associated with them.

6.2.1 The general learning problem

In general, we want a learner 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 black-box disk resource, traditional attributes of interest include *disk type*, *request inter-arrival time*, *read:write ratio*, etc. Non-traditional, but still important attributes, could be things like *day of week* (workloads can behave differently

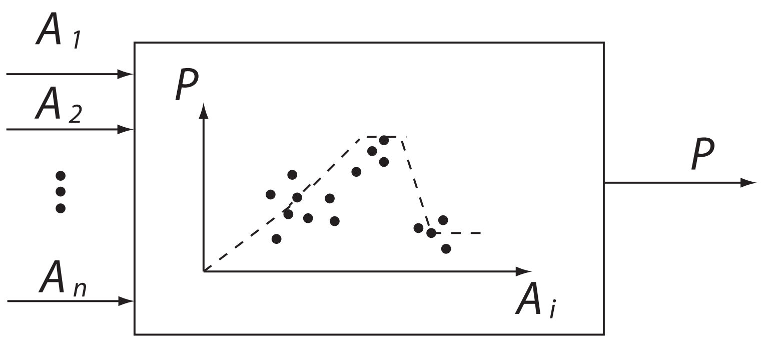


Figure 6.3. **Illustration of a learner.** The learner observes correlations between inputs and the output and fits a function to them. In a storage system, $\mathcal{A}_i$ could be the number of storage-nodes data is striped on, and $\mathcal{P}$ could be the observed throughput.

on Mondays than Sundays). $\mathcal{P}$ can be any metric of interest we want to validate, such as, throughput X_i. Figure 6.3 shows this generic learner.

The above classification problem translates into these primary requirements that a generic learner needs to satisfy:

- **Handling of mixed-type attributes.** 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 *day of week* has categorical attributes (Monday, ..., Sunday).
- **Handling of combinative associations.** A system's behavior may depend on combinations of attributes. For example, the expected throughput from a disk array may depend on both workload burstiness and request locality.
- **Reasonably fast.** The learner must be able to make predictions reasonably fast, and ideally train 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.
- **Adaptive.** The learner must efficiently adapt to new workloads and learn incrementally from each individual observation.

- **Cost-sensitive.** 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. The need for cost-sensitive models was first described in Section 3.1.
- **Interpretable.** 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 verifying the new rules with their intuition.

6.2.2 Z-CART: combining domain expertise with observations

Many algorithms have been developed in the machine learning community. We chose to use classification and regression trees (CART). Trees handle mixed attributes easily, and combinations of related attributes (i.e., AND and OR) are naturally captured. Moreover, trees require minimal storage, quickly make predictions, are easy to interpret, and can learn incrementally over time [Utgoff et al., 1997]. Training time is $O(nHeight(n))$ time, but predictions only 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 3 and 10).

We augment CART to be zero-training (Z-CART). Traditional CART models (much like other models, like Bayes Nets or neural nets) suffer from requiring much training data before even simple classifications can happen. Z-CART derives its initial structure from the expectation-based models (think of the latter as *domain knowledge* in this setting) and, hence, does not require any training data to start making predictions. This section describes how Z-CART works.

Figure 6.4 shows an example starting point for a Z-CART model. The goal of the model is to predict the maximum throughput from the CPU resource $X_{c,CPU}$. To construct the initial model, the CPU *What*...*if* model, first presented in Section 5.2.2, is asked the question "*What* is the CPU

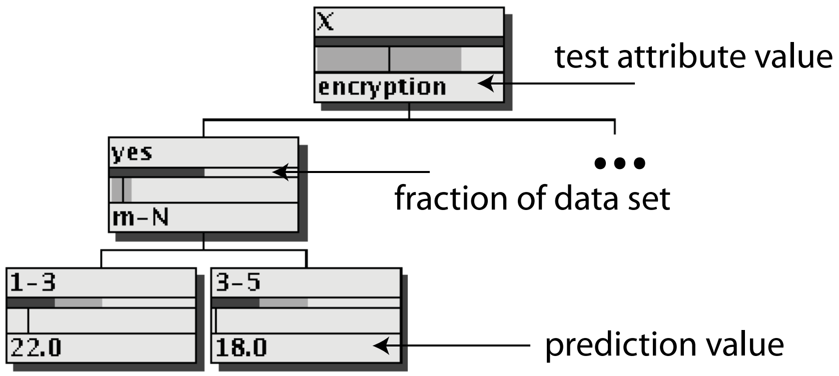


Figure 6.4. **Initial Z-CART model.** The initial model is built using expectation-based models (domain expertise) and requires no training data. DTVIEW is used to generate this diagram [Borgelt, 2007].

demand $D_{c,CPU}$ for requests from client c *if* the data is encoded using scheme E?", for each of the hypothetical schemes the client is interested in (e.g., 1-of-3 with encryption, 3-of-5 with encryption, etc.) The CPU model uses direct measurements to answer that question.

Imagine a situation in which, after the encoding scheme has been chosen, a certain workload gets less than half of its predicted throughput in the field. After successful mismatch localization, the CPU encode/decode service center might be identified as the culprit. 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. To see if any of these attributes are correlated to the unexpected drop in performance, Z-CART uses the original CART algorithm for selecting the most relevant attribute. 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 is:

$$Gain(S, A_i) = Entropy(S) - \sum_{a \in A_i} \frac{|S_a|}{|S|} Entropy(S_v) \tag{6.1}$$

where S is a set of classification instances with probability distribution $P(s)$ and

$$Entropy(S) \equiv - \sum_{\text{all classes } s} P(s) log_2 P(s) \tag{6.2}$$

In general, a very difficult problem for any machine learning algorithm is the *attribute selection problem*. It is best if all important attributes are available for the models to choose from, hence systems should be built to expose as many potentially important attributes to the tools as possible. However, there is a risk of having the machine learning algorithm "confused" by having too many attributes to choose from (this is known as the *curse-of-dimensionality* [Mitchell, 1997]). Initially, we exposed the machine learning tool to the workload attributes that are collected through the request flow traces collected by Stardust (i.e., those shown in Table 4.1).

Figure 6.5 shows the modified Z-CART model after the block size attribute has been incorporated into it. The leaves of the Z-CART model contain classification information and confidence information. Confidence 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 $D_{c,CPU}$ less than 1 ms?"), 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 $D_{c,CPU}$?"), the leaf maintains a histogram of observations. At one extreme, the histogram could keep a bucket for each unique observation. That, however, may be expensive, hence, in practice a limited number of buckets (e.g., 10) is kept.

Observer's architectural diagram, shown in Figure 3.2, illustrates how expectation-based and observation-based models interact with one another. Every $\underline{What}...\underline{if}$ question is answered by the observation-based model. The first time the question is posed, the observation-based model checks with the expectation-based model (domain expertise). 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 $\underline{What}...\underline{if}$ question. After observations in the field, the observation-based models further refine their predictions.

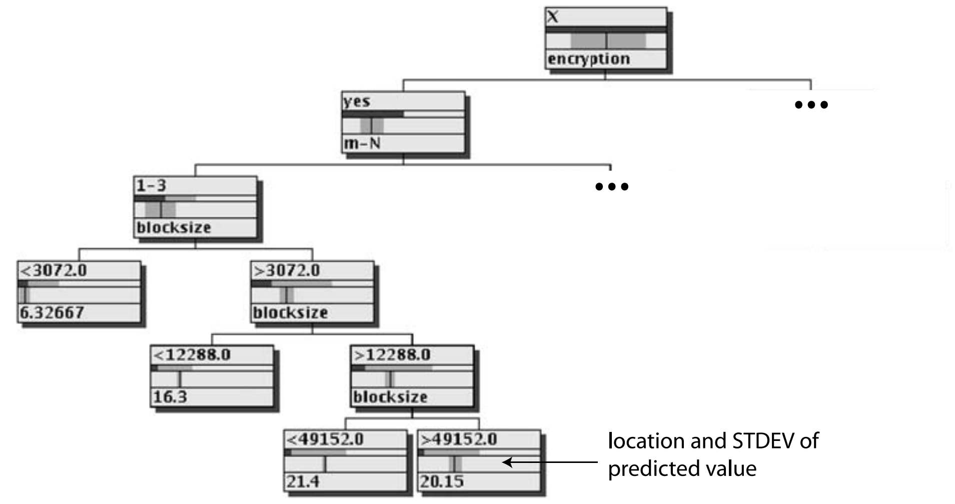


Figure 6.5. **Modified Z-CART model.** After several observations with new experiments, the initial Z-CART model automatically evolves to incorporate the new block size attribute. The predicted value (vertical line) together with one standard deviation (shaded area around vertical line) is shown by the GUI tool. 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).

Any subsequent *What*...*if* questions obtain answers with confidence metrics associated with them.

6.2.3 An aside: why learning is not sufficient

Learner models based on observations can, in theory, "discover" expected system behavior, and hence it seems there would be no need for designers to define expectation-based models. Indeed, most of the queuing formulae described in Chapter 3 can be discovered by fitting functions to observed data while deployed. So, why not use only learner models and sidestep the need to have system designers think about structural and performance expectations for their systems? There are several answers to this question. Most of them focus on efficiency and correctness, and one is more subjective.

First, there is an efficiency issue. Observation-based models require a large set of distinct observations, an issue that 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 in order to make a reasonable performance prediction (which can be made in a straightforward manner when employing expectation-based models that use queuing analysis coupled with domain knowledge on each service center).

A simple analogy that illustrates the efficiency issue exists in the game of chess. 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 obtained from all chess games played in the last 100+ years. Another option is to make the move by using a model that has the chess rules embedded (i.e., white-box). Systems and workloads change over time (whereas the chess rules remain the same), so purely black-box observations in systems could not only take a long time to observe enough unique interactions, but they also risk becoming obsolete quickly (e.g., every time a new device is added). Although several research groups have used purely observation-based models [Aguilera et al., 2003; Cohen et al., 2004; Mesnier et al., 2007; Wang et al., 2004], no analysis of their long-term behavior and "refresh" requirements has been clearly shown.

Second, there is a correctness issue. Just because an observation is true most of the time (and thus strongly correlated), it does not mean the observation matches the intended system behavior. Figure 6.6 shows how a learner model fits the data in the field. This graphs shows throughput as a function of number of storage-nodes data is striped on. The learned behavior differs from the expected behavior, especially as data is striped over a large number of storage nodes. It is *very good* that the designer took time to define the expected behavior in this case, since the discrepancy reflects

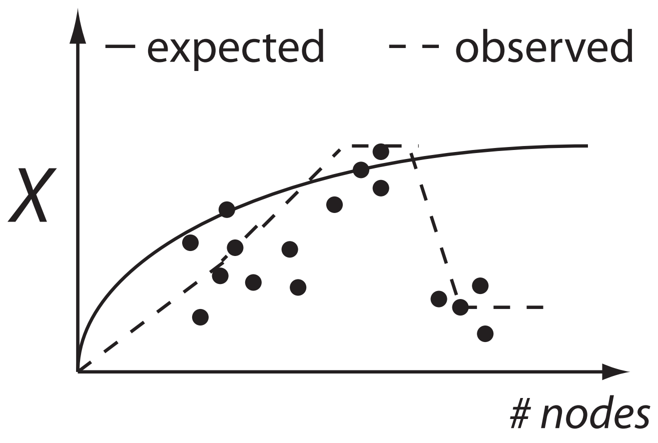


Figure 6.6. **Observed versus expected behavior.** Discrepancies in observed vs. expected behavior could indicate serious anomalies in the system. This particular figure illustrates how throughput changes as a function of the number of storage-nodes data is striped onto. Observations indicate a potential problem. The human is still needed to find the root-cause of the problem when it first occurs.

a serious performance anomaly in the system (known as the TCP-incast problem [Nagle et al., 2004], and further discussed in Section 6.4.3). Simply looking at observed data would not expose the anomaly.

The third reason designers should provide expectation-based models is more of a personal belief. I believe that thinking of models may force the designer (and programmer) to write more predictable algorithms and better code. Designing systems in an ad-hoc way and hoping that a learning algorithm will figure out how it works is wishful thinking. (There are cases, however, especially with legacy systems, when it is not possible to look at the source code to build good expectations. In those cases, purely statistical observations are the only option.)

The counter-argument is "why design models if discovering the system properties can be (eventually) accurate?" Proponents of this argument may point out that the system is probably behaving correctly most of the time [Kremenek et al., 2006], hence, structural expectations can be inferred by running a few workloads on the system and observing how requests flow through it. Others may have scenarios where only eventual accuracy matters, rather than the ability to predict well on day 1. Thus, performance expectations can be discovered over time, after many workload mixes have been observed. In general, the right answer depends on the client requirements and uses of the models. However, if a system is designed with a clean slate (like Ursa Minor was), we believe that it is natural to incorporate domain expertise into the models. Appendix A describes our experiences with a legacy system (Microsoft's SQL Server).

6.3 Incorporating observation-based models in Ursa Minor

The observation-based models in Ursa Minor are responsible for helping model designers detect new structural and performance behavior in the system. The key to detecting new structural behavior is having a good measurement infrastructure in place that keeps track of the requests at entry and exit points for each service center in the system. Stardust does just this.

Every time a request passes through an instrumentation point, a record is stored in the appropriate Activity DB table. Using the information contained in the Activity DB 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 as a DOT graph data structure [GraphViz, 2006].

A simple example illustrates how the structural path can be discovered. Assume no expectation-based structural models exist for Ursa Minor. One way to bootstrap the discovery process is to run a few workloads in the system. Each time a request from these workloads propagates from service center to service center, the system posts activity records at their entrances and exits. These records are stored in the Activity DB tables, and they are sufficient to understand how most requests flow in the system.

The prototype in Ursa Minor for localizing performance deviations currently only works offline. It calculates various statistics based on observed

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

Table 6.1. **Relevant system-workload attributes.** In bold are the attributes we *felt sure* had a direct relationship with performance, primarily because there is theoretical backing of their correlation. The system itself discovers the strength of the correlation with the various attributes.

throughput and response time and compares those to what expectation-based models predict.

Model re-learning is done through Z-CART, which currently uses the ITI package as the underlying classification and regression tree structure [Utgoff et al., 1997]. In addition to the attributes exposed by Stardust (summarized in Table 4.1), Table 6.1 shows several other attributes for which we knew the correlation with performance was important, though we did not know the exact nature of the correlation. We exposed these attributes at Z-CART, which then discovered the strength of the correlation between those attributes and performance. 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.

6.4 Evaluation

This section evaluates the efficacy of the observation-based models to locate mismatches with expectations and learn revised expectations. The performance problems illustrated have been encountered in Ursa Minor during the last two years. Because self-checking has only recently been prototyped into the system, we were not able to evaluate its usefulness when the problems first arose. Instead, we replicated the problems as best as possible with the current code base.

6.4.1 Experimental setup

The experimental setup and workloads used are identical to the ones first described in Section 4.4.1 and Section 5.3.1.

6.4.2 Localizing sources of mismatch

This subsection experiments with cases when the localizer can and cannot be used to guide human attention and help with the diagnosis of problems. For the purposes of 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.

The overheads of self-checking are equivalent to the overheads of creating latency graphs, as first evaluated in Section 4.4.3. Currently, once a latency graph is created, a human needs to visually "diff" it with the expected latency graph. As part of future work, we are considering building tools that "diff" general graphs more efficiently than by visualization. Below, we discuss broad problem categories that the behavioral models can and cannot localize, generalizing from the experience so far.

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 have experienced are degraded hash table performance over

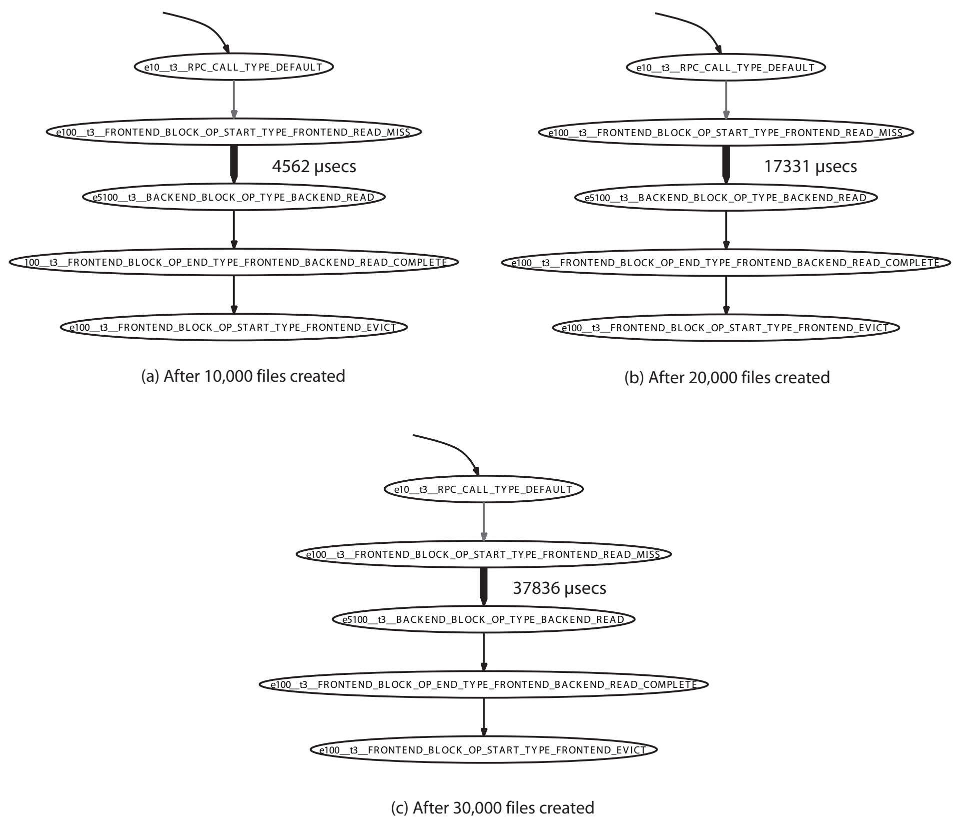

e10__t3__RPC_CALL_TYPE_DEFAULT
e100__t3__FRONTEND_BLOCK_OP_START_TYPE_FRONTEND_READ_MISS
4562 μsecs
e5100__t3__BACKEND_BLOCK_OP_TYPE_BACKEND_READ
100__t3__FRONTEND_BLOCK_OP_END_TYPE_FRONTEND_BACKEND_READ_COMPLETE
e100__t3__FRONTEND_BLOCK_OP_START_TYPE_FRONTEND_EVICT
(a) After 10,000 files created
e10__t3__RPC_CALL_TYPE_DEFAULT
e100__t3__FRONTEND_BLOCK_OP_START_TYPE_FRONTEND_READ_MISS
17331 μsecs
e5100__t3__BACKEND_BLOCK_OP_TYPE_BACKEND_READ
e100__t3__FRONTEND_BLOCK_OP_END_TYPE_FRONTEND_BACKEND_READ_COMPLETE
e100__t3__FRONTEND_BLOCK_OP_START_TYPE_FRONTEND_EVICT
(b) After 20,000 files created
e10__t3__RPC_CALL_TYPE_DEFAULT
e100__t3__FRONTEND_BLOCK_OP_START_TYPE_FRONTEND_READ_MISS
37836 μsecs
e5100__t3__BACKEND_BLOCK_OP_TYPE_BACKEND_READ
e100__t3__FRONTEND_BLOCK_OP_END_TYPE_FRONTEND_BACKEND_READ_COMPLETE
e100__t3__FRONTEND_BLOCK_OP_START_TYPE_FRONTEND_EVICT
(c) After 30,000 files created

Figure 6.7. **Performance degradation over time.** A hashing algorithm bug leads to unexpected performance drops over time (bold edges).

time and degraded disk performance as the disk gets fuller. For the purposes of this discussion, we focus on degradation of hash table performance. An initial hashing algorithm had a bug that led to non-uniform hashing. Figure 6.7 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, 20,000 and 30,000 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 other system bugs or misconfigurations we have seen, the mismatch is shown as an increase or decrease in edge latencies.

Model bugs: A representative problem in this category relates to non-linear behavior of the individual models that I did not correctly model initially. A concrete instance involves the CPU model. As described in Section 5.2.2, our CPU model 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). In experiments, we observed that an SSIO_BENCHMARK run 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 SSIO_BENCHMARK was configured to use small block sizes (512 B), and the kernel network stack consumed significant amounts of CPU per-block, as shown in Figure 6.8. Hence, it was impossible to keep the network pipeline full, since the CPU could not keep up. 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. In creating the model, we did not foresee the different behavior resulting from the small block sizes. 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 6.4.3 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: Localization of mismatches is most useful when the mismatch is found between two instrumentation points. There are times, however, when the usefulness of the localizer in understanding the source of the problem was diminished because of 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

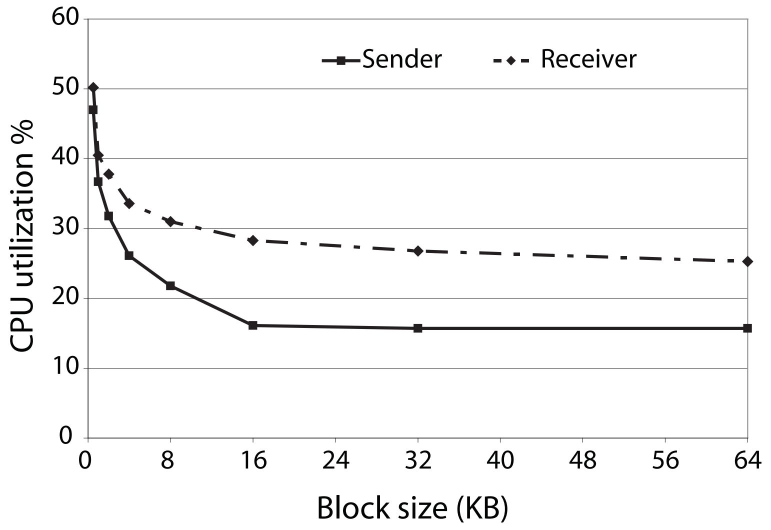


Figure 6.8. **Non-linear demand on the CPU.** When keeping the network pipeline full, a large fraction of the CPU consumption goes towards handling network traffic. The variance is negligible in all cases.

self-checked to see if the expectations matched the observations. (When this particular upgrade happened we did have the models in place.)

Figure 6.9 shows two latency graphs obtained when the Postmark workload was run on the 2.4 and 2.6 versions, respectively. These graphs only illustrate the changes for one of the many request types that experienced performance changes (create calls) and is shown for only the NFS Server (i.e., the full graph is much larger and is edited to reduce clutter). Bold edges indicate discrepancies from version 2.4 to 2.6 (for this example, a discrepancy is noted whenever a new value falls outside the one-standard deviation range).

In a system with many services, localizing the problem to four graph edges on a single server does indeed reduce the search space (for example, we knew the problem was not related to the buffer cache or disk resources in the system). However, understanding the root cause of the problem would have been easier if only one of the edges had a discrepancy. In this particular case, several edges had discrepancies, since a change in the TCP stack processing

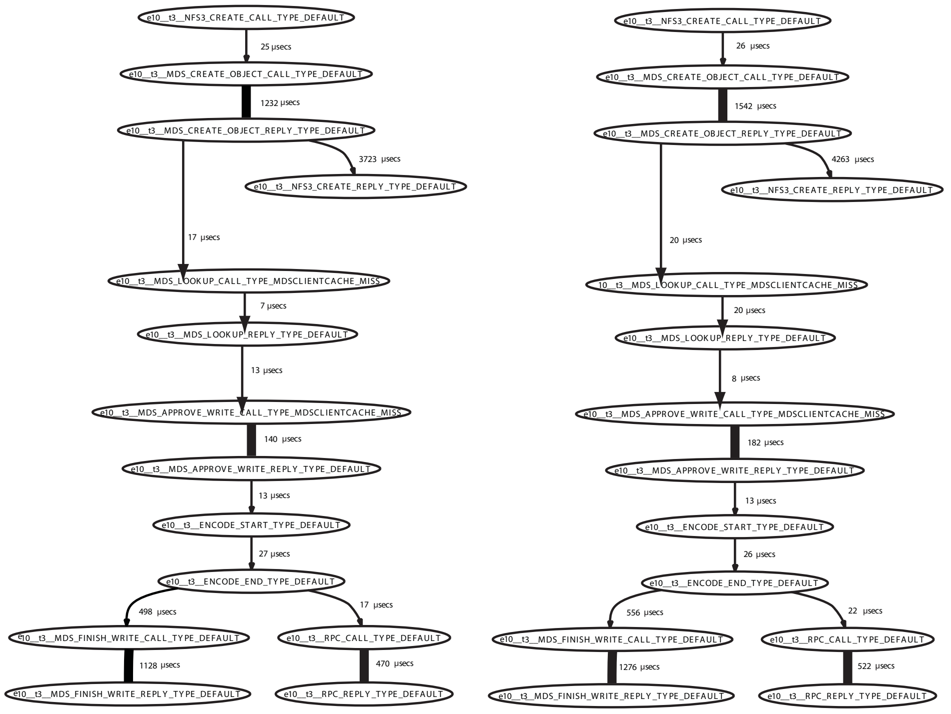

e10__t3__NFS3_CREATE_CALL_TYPE_DEFAULT
25 µsecs
e10__t3__MDS_CREATE_OBJECT_CALL_TYPE_DEFAULT
1232 µsecs
e10__t3__MDS_CREATE_OBJECT_REPLY_TYPE_DEFAULT
3723 µsecs
e10__t3__NFS3_CREATE_REPLY_TYPE_DEFAULT
17 µsecs
e10__t3__MDS_LOOKUP_CALL_TYPE_MDSCLIENTCACHE_MISS
7 µsecs
e10__t3__MDS_LOOKUP_REPLY_TYPE_DEFAULT
13 µsecs
e10__t3__MDS_APPROVE_WRITE_CALL_TYPE_MDSCLIENTCACHE_MISS
140 µsecs
e10__t3__MDS_APPROVE_WRITE_REPLY_TYPE_DEFAULT
13 µsecs
e10__t3__ENCODE_START_TYPE_DEFAULT
27 µsecs
e10__t3__ENCODE_END_TYPE_DEFAULT
498 µsecs
17 µsecs
e10__t3__MDS_FINISH_WRITE_CALL_TYPE_DEFAULT
e10__t3__RPC_CALL_TYPE_DEFAULT
1128 µsecs
470 µsecs
e10__t3__MDS_FINISH_WRITE_REPLY_TYPE_DEFAULT
e10__t3__RPC_REPLY_TYPE_DEFAULT
e10__t3__NFS3_CREATE_CALL_TYPE_DEFAULT
26 µsecs
e10__t3__MDS_CREATE_OBJECT_CALL_TYPE_DEFAULT
1542 µsecs
e10__t3__MDS_CREATE_OBJECT_REPLY_TYPE_DEFAULT
4263 µsecs
e10__t3__NFS3_CREATE_REPLY_TYPE_DEFAULT
20 µsecs
10__t3__MDS_LOOKUP_CALL_TYPE_MDSCLIENTCACHE_MISS
20 µsecs
e10__t3__MDS_LOOKUP_REPLY_TYPE_DEFAULT
8 µsecs
e10__t3__MDS_APPROVE_WRITE_CALL_TYPE_MDSCLIENTCACHE_MISS
182 µsecs
e10__t3__MDS_APPROVE_WRITE_REPLY_TYPE_DEFAULT
13 µsecs
e10__t3__ENCODE_START_TYPE_DEFAULT
26 µsecs
e10__t3__ENCODE_END_TYPE_DEFAULT
556 µsecs
22 µsecs
e10__t3__MDS_FINISH_WRITE_CALL_TYPE_DEFAULT
e10__t3__RPC_CALL_TYPE_DEFAULT
1276 µsecs
522 µsecs
e10__t3__MDS_FINISH_WRITE_REPLY_TYPE_DEFAULT
e10__t3__RPC_REPLY_TYPE_DEFAULT

Figure 6.9. **Multiple mismatches due to software changes.** Latency graphs created under the 2.4 and 2.6 Linux kernel. Only part of the full graph is shown in each case. The graphs cover the movement of requests through a single NFS server. Edges in bold indicate discrepancies between the runs on the different kernels.

(which was new in 2.6) affected both network transmission times and CPU consumption.

As part of future work, we plan on exploring orthogonal diagnostic methods that 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.

Handling anomalies external to the system: There were several performance problems our benchmarks had that could not be localized within Ursa Minor. In one concrete instance, Ursa Minor had a thread exhaustion problem at the NFS server that implicitly put pressure on the client/benchmark to send fewer requests. The symptoms of this problem were that fewer outstanding requests were observed in Ursa Minor. However, the number of outstanding request is an input to the models, not a metric that they can predict. In another instance, the client running the "S1-V0" scientific workload was having a problem with the network link just before entering Ursa Minor. From Observer's perspective, the requests were behaving as expected, once in the system.

In these instances, the models could report back and inform the client to look for the problem elsewhere.

Handling adaptive workloads: Adaptive workloads, first discussed in Section 3.1.1), might change their behavior depending on the performance of the underlying system. For example, Observer assumes 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 may experience a different workload depending on the speed of the underlying storage system. That happens, for example, when a web user may depart from the site (e.g., if it is too slow) and go to another.

As a concrete instance, we set up a synthetic OLTP database workload that kept changing its indexing behavior as a function of Ursa Minor's performance. Observer's models assume that historical characteristics of the workload do not change, 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. The predicted and observed hit rate would significantly differ.

Currently, if discrepancies at the buffer cache accesses are observed for a given workload, Observer chooses not to make predictions for that workload.

6.4.3 Relearning and deriving confidence

This section illustrates how observation-based models could help model designers discover new workload-system correlations. In addition, this section describes how confidence can be reported with each prediction. Confidence about past predictions could then be used when making predictions about new changes.

Unexpected CPU bottleneck: The localization of this problem instance was first described in Section 6.4.2. 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, cache, disks) self-check, the CPU model is found to be the culprit (i.e., it under-predicted the CPU demand). Z-CART noticed that the attribute "block size" is significantly smaller than in the test cases and eventually incorporates that attribute in the decision tree (the resulting tree and confidence values were first shown in Figure 6.5). Of course, "block size" is an attribute that the programmer had to expose to Z-CART, for Z-CART to discover the correlation. Figure 6.10 shows the improvement in accuracy from the hybrid modeling technique.

When striping goes wrong: As described in Section 5.2.2, 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 SSIO_BENCHMARK configuration led to the benchmark 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).

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 [Nagle et al., 2004], and is rather unique

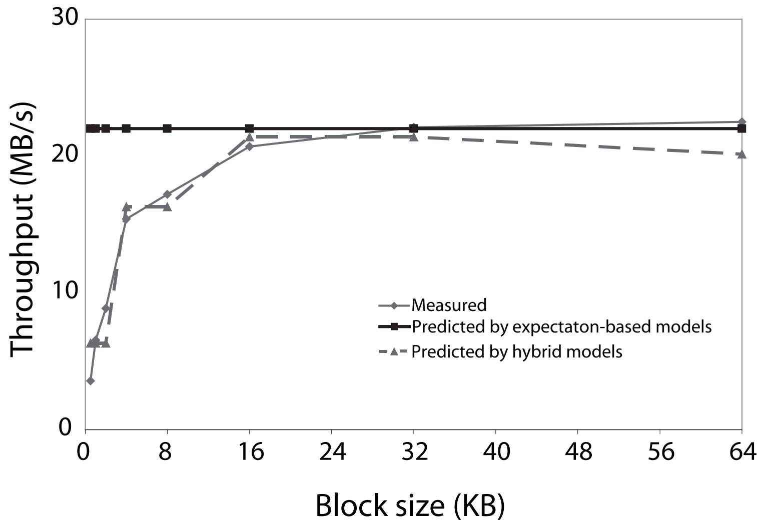


Figure 6.10. **Improved prediction from hybrid models.** After learning the new correlation from observations on a particular server, the results from the Z-CART model generalize on any server with the same CPU type. The standard deviation for the measured performance is negligible, whereas for the predicted performance it is shown in Figure 6.5.

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).

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 Observer considers them them to be black-box (i.e., Observer has no understanding of their internal algorithms). To explore how Z-CART could help, we extended it to incorporate the *switch type* as one of the attributes to check. We ran 825 experiment instances with erasure coding schemes

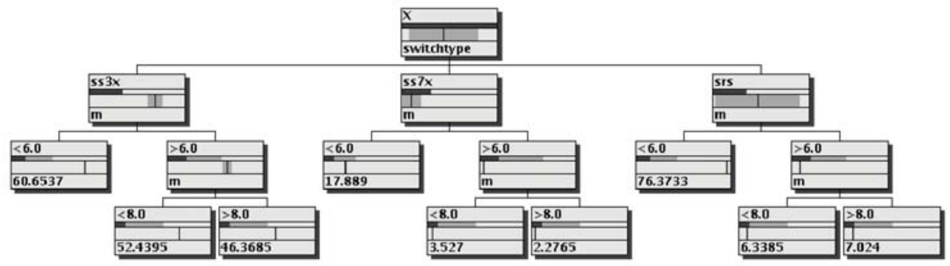


Figure 6.11. **Z-CART adapts the predictions to the switch type.** Ursa Minor has three types of donated switches (anonymized to srs, $ss3x$, $ss7x$), and each of them behaves differently as a function of m for erasure coding schemes. We treat them as black-box and Z-CART discovers their behavior over 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).

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 6.11. Currently, we are living with this problem ("fixing" it would require buying new switches). Hence, the current network model is adjusted to account for the switch type.

A remaining issue is how to handle predictions for an unseen switch type. For example, if a switch of type $SwitchNew$ is to be purchased, we would like the system to make a prediction about its performance (together with providing confidence values). To solve this issue, a new Z-CART model can be built, this time without the switch type as an attribute. Figure 6.12 plots the average throughput predicted, together with standard deviation. It is up to a higher policy layer to make the decision on what m to use based on such predictions.

6.5 Summary, lessons learned, and limitations

Figure 6.13 summarizes the main idea of this chapter. By using robust models, the human's job is simplified. Models are used not only to predict, but also to localize deviations from expectations. Humans ultimately need to fix

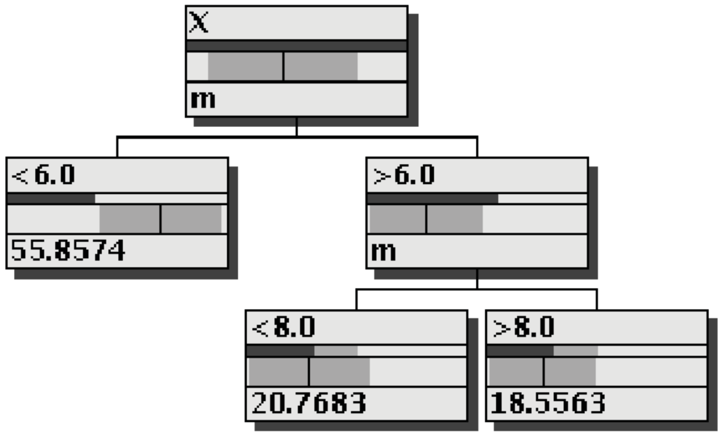


Figure 6.12. **Confidence when making predictions for a new switch type.** When buying a new switch, the best the models can do is provide confidence on their predictions based on m only. The location of the predicted value is relative to the minimum and maximum values observed (in this case 2.1 MB/s and 77 MB/s respectively).

the root cause of a problem, but the models help them focus their attention. Over time, machine learning techniques discover new, unforeseen correlations between attributes of the workload-system space and performance metrics of interest. 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. Hence, systems are designed to expose many of these attributes, and the models automatically calculate the correlation strength of each of them.

The main lesson learned is that queuing-based expectation-based models are brittle. System or model misconfigurations, coupled with limited inputs to the models, happened often enough in Ursa Minor to lead us to develop observation-based models to partially mitigate them.

Although combining statistical techniques with queuing analysis has proven useful, there are several limitations that this dissertation does not address and leaves as future research.

Zooming-in on root cause: The simple mismatch localization method

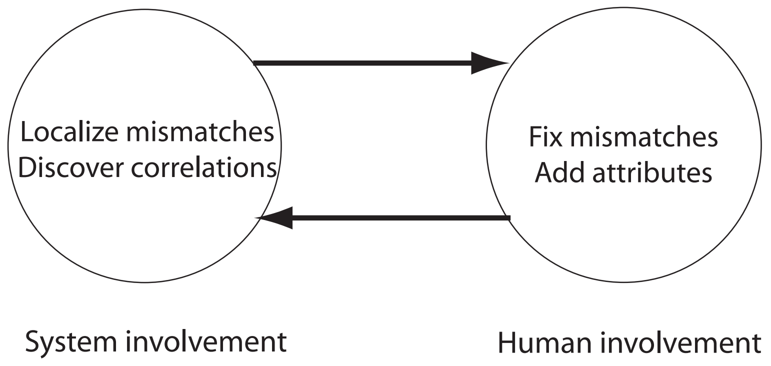


Figure 6.13. **The synergy between robust models and humans.** Models localize mismatches between models and the components they model and guide the human's attention. Models also test for correlation strengths and continuously reevaluate their predictions. Humans ultimately fix the bugs that cause the mismatches and build systems to expose a rich pool of attributes to the models.

used to illustrate the approach can be improved. Currently, it localizes problems that lead to resource consumption deviations, and the localization granularity is limited by the granularity of measurements collected by Stardust. Getting close to root cause analysis will involve collecting many more attributes, at different abstraction/semantic levels in the system. For example, in Ursa Minor, we have started to collect *environmental data* (temperature, humidity, etc), *hardware demands* (per-request, per-machine), *error messages*, *request flow traces*, hardware and software *configuration data* (components, topology, users), and annotations of *human activity*.

Ideally, one would have a measurement mechanism in place that, on demand, allows for the collection of more statistics at lower layers, if a problem in the upper layers is detected. The Paradyn project [Hollingsworth et al., 1994] has shown how self-propelling instrumentation can do just that (however, the technique does not currently extend to distributed systems). It would be interesting to collect case studies that show what is gained by finer grained instrumentation. As Section 6.2.2 cautioned, however, machine learning tools are not mature enough to handle many attributes because of the curse of dimensionality. This chapter examined the benefits from using hand-picked attributes that the system designer exposes through Stardust.

Active probing and the big, red button: We believe that two approaches could 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) could 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, and when these probes should run. Currently, Observer does not actively probe in Ursa Minor.

Second, there needs to be a way to involve a human in directing and shaping the algorithm's focus. There may be 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 could perhaps 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.

7 Conclusions

A self-predicting system monitors itself and answers *What...if* questions about hypothetical changes. This dissertation describes and evaluates our experiences with making Ursa Minor self-predicting. Such self-prediction has the potential to reduce the amount an administrator must understand about complex workload-system interactions and can be a step towards the goal of self-managing distributed systems. The mathematical models within Ursa Minor can answer *What...if* questions about the performance impact of data encoding changes, adding or removing datasets/workloads, and adding or removing storage-nodes. The results demonstrate the feasibility of self-prediction in a real system, and we believe that the same modeling architecture could work in general. (Appendix A provides early experience with a legacy database system upon which we base this generalization.)

7.1 Contributions

The key contribution of this thesis is identifying and handling show-stoppers that prevented mathematical models from making predictions in many shared, distributed systems. From a theoretical perspective, these show-stoppers were surprising because the mathematical models appear so simple. From a practical perspective, these show-stoppers needed to be addressed to make even simple predictions. Discovering them required deploying prototype models in realistic systems and performing case studies. In particular, mathematical models explored in this dissertation require system support for end-to-end monitoring, performance insulation, and collection of historical data on system behavior.

End-to-end monitoring can provide models with the right measurements in distributed, shared systems. Such measurements include per-client, per-resource demands and per-client request flow graphs. I identified and instrumented key service centers in a cluster-based storage system and in a commercial database system. These service centers represent four key resource types (CPU, network, buffer cache, and disks) that are common in many systems. I demonstrated that end-to-end monitoring can be efficient in both collection and querying phases, allowing it to be ON at all times.

Performance insulation allows independent analysis of workloads in the system, thus side-stepping the need for complex modeling of workload interactions. I built models that help the performance insulation algorithms work. In particular, the buffer cache and disk models answer a series of *What...if* questions to determine good cache partitioning, prefetching and quanta-based scheduling policies. In turn, the models I built stayed simple because of the guarantees that performance insulation provided.

Collection of historical data helps in localizing sources of system and model bugs and misconfigurations and allows for fidelity metrics to be provided with predictions. I demonstrated how combining queuing operational laws with a machine learning algorithm, like CART, provides a good starting point towards the vision of robust system models that handle unforeseen behavior. Models need to be robust to minimize the human effort needed to maintain and update them.

7.2 Thoughts on future work

There are several directions in which this work could be extended. Currently, three concrete directions have emerged and are being actively pursued. First, the observation-based models, discussed in Chapter 6 are being researched further now that they are included in Ursa Minor. We want to create a taxonomy of problems they can address and their limitations. Ideally, to create a good, complete taxonomy, several months to years of deployment would be necessary.

Second, the predictive infrastructure presented in this thesis creates a building block on top of which optimization tools can be built. Iterating over *What...if* questions (e.g., one for each possible option) is one way such optimizers can work. However, building optimization tools was not in the scope of this thesis. Other people in the Parallel Data Lab (e.g., John Strunk) have currently built a prototype optimizer, that incorporates not only my performance models, but also models built for other system metrics, such as availability, reliability, capacity, and power. John's research, in particular, analyses how to combine the outputs from all models to a single answer that the client can reason about.

Third, the infrastructure is being specialized to help with anomaly detection in the system. Stardust is a key enabler, and visualization techniques are being developed to cluster and analyze anomalies in system behavior [Sambasivan et al., 2007].

In the short- to medium-term, investigating ways to layer two or more self-predicting systems in a decentralized way is warranted. Some recent work has shown that another discipline, control theory, is useful in guiding local optimizations to a stable global state in a multi-tiered system, at least when there exists a centralized decision-making module [Padala et al., 2007]. Without that control, two self-predicting system tiers may keep making local optimizations without ever reaching a good overall system configuration. When two or more systems in separate administration domains are layered (e.g., a DB on top of Ursa Minor), however, there are several differences from the centralized case. First, the systems may have incomplete information about each other. A DB has no easy way to infer what other workloads are running on Ursa Minor and how those workloads interact with Ursa Minor's resources. Ursa Minor does not know how the DB manages its resources, such as buffer cache. Control theory assumes measurements from both systems are visible to a third entity (the controller). That may not be the case, in practice. Second, both systems may make local optimizations and have separate decision making processes. For example, index tuning agents may decide online what indices should be created [Chaudhuri and Narasayya, 1998; Valentin et al., 2000]. Ursa Minor may attempt to maximize storage-node utilization with its decisions on data placement and encoding.

In the long-term, the role of humans in the face of self-predicting system will have to be evaluated. Currently, administrators we talk to want to ask *What...if* questions to gain the trust of the system. How smart do humans have to be to ask the right *What...if* questions? Will self-managing systems be more difficult to debug by humans, when some optimization goes wrong? CMU's Self-* Storage project is a long-term research endeavor meant to explore storage automation, including such questions (Ursa Minor is the first prototype of this project).

A Lessons from a legacy system

A common question I have had when presenting this work to colleagues in industry has been "how difficult is it to make an existing legacy system self-predicting?" Ursa Minor was a system we designed from scratch, and hence we did not have to deal with legacy concerns. In addition, we did not feel the same pressure that product teams have to meet project timelines. This appendix informs this question with one concrete experience with taking a commercial, legacy database (Microsoft's SQL Server) and enabling it to answer *What*...*if* questions. One concrete important *What*...*if* question in SQL Server is "*What* happens to each transaction type's performance, *if* we change the total amount of buffer cache?"

This appendix will not evaluate the accuracy of the results, which are very similar to those obtained in Ursa Minor (the interested reader is advised to consult [Narayanan et al., 2005, 2006]). Instead, we focus on what it took to add an end-to-end measurement framework and mathematical models to an existing system and how it can be done incrementally.

A.1 Initial state of system

Our experience was with a single instance of the SQL Server 2005 prototype running on Windows Server 2003. Thanks to the developers of SQL Server, we were given access to its source code.

SQL Server, at the time, had highly-configurable performance counters, but no end-to-end tracing. The Windows operating system, however, had a basic framework called ETW [Microsoft, 2005] that allowed traces to be collected efficiently (and stored them to a flat file)[1]. ETW exposed interfaces for posting activity records and storing them. However, at the time, no request IDs/breadcrumbs were maintained. SQL Server itself did not make use of ETW and maintained its own measurement framework.

SQL Server did not have mathematical models to address the types of *What...if* questions we were interested in.

A.2 Adding measurements with context

The first changes involved adding a measurement framework that could keep track of requests as they passed through the DB layers. First, we identified the points in the system where ETW traces should be collected. Just like in Ursa Minor, these points involved entrance and exit points from the common service centers in the system (e.g., task scheduler, buffer cache, disk driver). At each of these points, we inserted activity record posting calls to the ETW library. This first step did not involve propagating contextual information and made use of an already-existing tracing infrastructure in the Windows operating system. Some effort was required to find the locations to instrument, however, just like in Ursa Minor, these were usually well-defined in the documentation of the system and were easy to spot by looking at the code.

Second, we added contextual information by using the Magpie stitching framework (some differences between Magpie and Stardust are described in Section 4.3.3). The stitching did not require any changes to SQL Server's API, but did require a human (me) to define the structural behavior of the system and describe how requests are supposed to flow through it.

A.3 Adding mathematical models

The low-level mathematical models that we built into SQL Server were similar to the ones in Ursa Minor, except that the CPU model was only observational-based. The SQL language allows a lot of flexibility in defining

[1]A similar framework for Linux is the Linux Tracing Toolkit [GNU, 2002].

transactions; in practice, that makes it very difficult to build expectation-based models for query CPU behavior. Ursa Minor, on the other hand, exports a relatively small set of interfaces (approximately 19 NFS calls), for which is possible to construct expectation-based models. The backbone of the analysis engine in SQL Server used queuing analysis, just like in Ursa Minor.

The models in SQL Server were built as a separate library and ran offline. In Ursa Minor, many of the models can now run online. In practice, for performance predictions, offline predictions are sufficient. For anomaly detections, however, online analysis might be crucial.

A.4 What worked and didn't

Adding the instrumentation and building offline models took two people about 2 months. Refining and testing took one person an additional 2 months. So in total, 6 man-months were spent.

We ran several experiments on SQL Server, with an OLTP benchmark that emulated multiple clients. Answers to the question "*What* happens to each transaction type's performance, *if* we change the total amount of buffer cache?" were accurate for predicting averages. However, both throughput and latency in SQL Server had a high variance, for many of the same reasons discussed in Section 5.3.4 (e.g., no performance insulation between transaction types). Because of the lack of performance insulation, the buffer cache and disk models were forced to make worst-case predictions. Although some of the mechanisms for ensuring performance insulation to SQL Server are similar to the mechanisms in Ursa Minor, we did not implement them.

Another observation we made while experimenting with SQL Server is that, in addition to hardware resources, there are software resources that need to be modeled as well. For example, there were cases when the lock service would become a bottleneck, before any of the hardware resources were saturated. Coming up with a taxonomy of such non-hardware resources and modeling them is an area of future work for us.

Bibliography

Abd-El-Malek, M., Courtright II, W. V., Cranor, C., Ganger, G. R., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., Wachs, M., and Wylie, J. J. 2005. Ursa Minor: versatile cluster-based storage. In *Conference on File and Storage Technologies*. USENIX Association, 59–72. 10, 11, 62, 95

Abd-El-Malek, M., Courtright II, W. V., Cranor, C., Ganger, G. R., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., Wachs, M., and Wylie, J. J. 2006. Early experiences on the journey towards self-* storage. *IEEE Data Engineering Bulletin 29,* 3, 55–62. 11

Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. 2003. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating System Principles*. ACM Press, 74–89. 18, 52, 133

Akcelik, V., Bielak, J., Biros, G., Epanomeritakis, I., Fernandez, A., Ghattas, O., Kim, E. J., Lopez, J., O'Hallaron, D., Tu, T., and Urbanic, J. 2003. High Resolution Forward and Inverse Earthquake Modeling on Terasacale Computers. In *ACM International Conference on Supercomputing*. 83

ALLEN, N. March 2001. Don't waste your storage dollars: what you need to know. Research note, Gartner Group. Research note, Gartner Group. 13

ANDERSON, E. AND PATTERSON, D. 1997. Extensible, scalable monitoring for clusters of computers. In *Systems Administration Conference.* USENIX Association, 9–16. 50

ANDERSON, E., SPENCE, S., SWAMINATHAN, R., KALLAHALLA, M., AND WANG, Q. 2005. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems 23,* 4, 337–374. 22

ARLITT, M. AND JIN, T. 2000. A workload characterization study of the 1998 World Cup web site. *IEEE Network 14,* 3, 30–37. 31

BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. 1998. Resource containers: a new facility for resource management in server systems. In *Symposium on Operating Systems Design and Implementation.* ACM, 45–58. 24, 119

BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. 2004. Using Magpie for request extraction and workload modelling. In *Symposium on Operating Systems Design and Implementation.* USENIX Association, 259–272. 18, 53, 75

BORGELT, C. June 2007. DTreeGUI - Decision and Regression Tree GUI and Viewer. `http://www.borgelt.net//dtgui.html`. 130

BOROWSKY, E., GOLDING, R., JACOBSON, P., MERCHANT, A., SCHREIER, L., SPASOJEVIC, M., AND WILKES, J. 1998. Capacity planning with phased workloads. In *WOSP.* 31

BOUHANA, J. P. 1996. UNIX Workload Characterization Using Process Accounting. In *22nd International Computer Measurement Group Conference.* 379–390. 51

CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. 2004. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference.* USENIX Association, 15–28. 50

Carey, M. J., DeWitt, D. J., Franklin, M. J., Hall, N. E., McAuliffe, M. L., Naughton, J. F., Schuh, D. T., Solomon, M. H., Tan, C. K., Tsatalos, O. G., White, S. J., and Zwilling, M. J. 1994. Shoring up persistent applications. In *ACM SIGMOD International Conference on Management of Data*. ACM Press, 383–394. 82

Chambliss, D. D., Alvarez, G. A., Pandey, P., Jadav, D., Xu, J., Menon, R., and Lee, T. P. 2003. Performance virtualization for large-scale storage systems. In *Symposium on Reliable Distributed Systems*. IEEE, 109–118. 24

Chanda, A., Cox, A., and Zwaenepoel, W. 2007. Whodunit: Transactional profiling for multi-tier applications. In *EUROSYS*. 92

Chaudhuri, S. and Narasayya, V. 1998. AutoAdmin what–if index analysis utility. In *ACM SIGMOD International Conference on Management of Data*. ACM Press, 367–378. 23, 151

Chaudhuri, S. and Weikum, G. 2000. Rethinking database system architecture: towards a self-tuning RISC-style database System. In *International Conference on Very Large Databases*. Morgan Kaufmann Publishers Inc, 1–10. 1, 23, 24, 25

Chaudhuri, S. and Weikum, G. September 2006. Foundations of Automated Database Tuning. VLDB 2006 Proceedings. Tutorial. 23

Chen, M. Y., Accardi, A., Kiciman, E., Patterson, D., Fox, A., and Brewer, E. 2004. Path-based failure and evolution management. In *Symposium on Networked Systems Design and Implementation*. USENIX Association, 309–322. 18

Chen, M. Y., Kiciman, E., and Brewer, E. 2002. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *International Conference on Dependable Systems and Networks*. IEEE Computer Society, 595–604. 53

Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. 1994. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys 26,* 2, 145–185. 97

Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., and Symons, J. 2004. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, 231–244. 18, 53, 133

Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. 2005. Capturing, indexing, clustering, and retrieving system history. In *ACM Symposium on Operating System Principles*. ACM, 105–118. 127

Denning, P. J. and Buzen, J. P. 1978. The operational analysis of queueing network models. *ACM Computing Surveys 10,* 3, 225–261. 21

Dinda, P. A. 2006. Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. *IEEE Transactions on Parallel and Distributed Systems, 17,* 2, 160–173. 18

Ellard, D., Ledlie, J., Malkani, P., and Seltzer, M. 2003. Passive NFS tracing of email and research workloads. In *Conference on File and Storage Technologies*. USENIX Association, 203–216. 31, 70

Ganger, G. R., Strunk, J. D., and Klosterman, A. J. 2003. Self-* Storage: brick-based storage with automated administration. Tech. Rep. CMU–CS–03–178, Carnegie Mellon University. 1, 62

GNU. 2002. Linux Tracing Toolkit. `http://www.opersys.com/LTT/`. 154

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. 2004. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks*. 63

GraphViz. February 2006. Graphviz - Graph Visualization Software. `http://www.graphviz.org/`. 90, 135

GRAY, J. 2003. A conversation with Jim Gray. *ACM Queue 1,* 4, 8–17. 13

GUNTHER, N. J. 2005. *Analyzing Computer System Performance with Perl-PDQ*. Springer-Verlag GmbH. 40, 106

HARDWICK, J., PAPAEFSTATHIOU, E., AND GUIMBELLOT, D. 2001. Modeling the Performance of E-Commerce Sites. In *27th International Conference of the Computer Measurement Group*. 22

HE, Q., DOVROLIS, C., AND AMMAR, M. 2005. On the predictability of large transfer TCP throughput. In *ACM SIGCOMM Conference*. 145–156. 17, 18, 20, 103

HELLERSTEIN, J. L., MACCABEE, M. M., MILLSII, W. N., AND TUREK, J. J. 1999. ETE: a customizable approach to measuring end-to-end response times and their components in distributed systems. In *International Conference on Distributed Computing Systems*. IEEE Computer Society, 152–162. 53

HEWLETT-PACKARD. May 2007. Tools and traces. HP Labs web site. `http://www.hpl.hp.com/research/ssp/software/`. 31, 70

HOLLINGSWORTH, J. K., MILLER, B. P., AND CARGILLE, J. 1994. Dynamic Program Instrumentation for Scalable Performance Tools. In *Scalable High Performance Computing Conference*. IEEE Computer Society, 841–850. 92, 147

HRISCHUK, C., ROLIA, J., AND WOODSIDE, C. M. 1995. Automatic generation of a software performance model using an object-oriented prototype. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. 399–409. 53

IBM CORPORATION. 2004. DB2 Performance Expert. `http://www-306.ibm.com/software`. 47, 50

ICAC May 2004. International conference on autonomic computing. http://www.autonomic-conference.org/. 1

INTERNATIONAL BUSINESS MACHINES CORP. October 2001. Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM web site. http://www.research.ibm.com/autonomic/manifesto/. 1

Iperf May 2007. Iperf. http://sourceforge.net/projects/iperf. 103

ISAACS, R., BARHAM, P., BULPIN, J., MORTIER, R., AND NARAYANAN, D. 2004. Request extraction in Magpie: events, schemas and temporal joins. In *11th ACM SIGOPS European Workshop*. 92–97. 53, 74

KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. 2004. Triage: Performance Isolation and Differentiation for Storage Systems. In *International Workshop on Quality of Service*. IEEE, 67–74. 24

KATCHER, J. 1997. PostMark: a new file system benchmark. Tech. Rep. TR3022, Network Appliance. 82

KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. 2004. Designing for disasters. In *Conference on File and Storage Technologies*. USENIX Association, 59–72. 96

KREMENEK, T., TWOHEY, P., BACK, G., NG, A., AND ENGLER, D. 2006. From uncertainty to belief: inferring the specification within. In *Symposium on Operating Systems Design and Implementation*. 161–176. 134

LAZOWSKA, E., ZAHORJAN, J., GRAHAM, S., AND SEVCIK, K. 1984. *Quantitative system performance: computer system analysis using queuing network models*. Prentice Hall. 21, 22, 35, 39, 40, 48

LOHMAN, G. March 2007. Personal communications. DB2 Architect. 1

MAGOUTIS, K., BRUSTOLONI, J. C., GABBER, E., NG, W. T., AND SILBERSCHATZ, A. 2000. Building appliances out of components using Pebble. In *ACM SIGOPS European Workshop*. 211–216. 25

MASSIE, M. L., CHUN, B. N., AND CULLER, D. E. 2004. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing 30,* 7, 817–840. 50

MENASCE, D. AND ALMEIDA, V. 1998. *Capacity planning for web performance: metrics, models and methods.* Prentice Hall. 21, 23, 48

MESNIER, M. P., WACHS, M., SAMBASIVAN, R. R., ZHENG, A., AND GANGER, G. R. 2007. Modeling the relative fitness of storage. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* ACM, 37–48. 18, 133

MICROSOFT. 2005. Event tracing. `http://msdn.microsoft.com/`. 50, 51, 65, 153

MICROSOFT. 2005. Microsoft .NET. `http://www.microsoft.com/net/default.mspx`. 53

MICROSOFT. 2005. Windows Server 2003 Performance Counters Reference. `http://www.microsoft.com/technet/`. 47

MITCHELL, T. M. 1997. *Machine learning.* McGraw-Hill. 20, 131

MOORE, F. 2005. *Storage New Horizons.* Horison Information Strategies. 13, 127

NAGLE, D., SERENYI, D., AND MATTHEWS, A. 2004. The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage. In *SC.* 134, 143

NARAYANAN, D., THERESKA, E., AND AILAMAKI, A. 2005. Continuous resource monitoring for self-predicting DBMS. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).* 11, 23, 153

NARAYANAN, D., THERESKA, E., AND AILAMAKI, A. 2006. Challenges in building a DBMS Resource Advisor. *IEEE Data Engineering Bulletin 29,* 3, 40–46. 11, 153

NORCOTT, W. AND CAPPS, D. 2002. IOzone filesystem benchmark program. `http://www.iozone.org`. 82

Oracle Corporation. 2004. Oracle Database Manageability. `http://www.oracle.com/technology/`. 47, 50

OSDI December 2004. Symposium on operating system design and implementation. http://www.usenix.org/events/osdi04/. 1

Padala, P., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Salem, K., and Shin, K. 2007. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*. 151

Perl, S. E. and Weihl, W. E. 1993. Performance assertion checking. In *ACM Symposium on Operating System Principles*. 134–145. 17

Quest Software. October 2005. Application Assurance. Quest Software web site. `http://www.quest.com`. 53

Rabin, M. O. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36,* 2, 335–348. 97

Reumann, J., Mehra, A., Shin, K. G., and Kandlur, D. 2000. Virtual services: a new abstraction for server consolidation. In *USENIX Annual Technical Conference*. USENIX Association, 117–130. 24, 119

Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. 2006. Pip: Detecting the unexpected in distributed systems. In *Symposium on Networked Systems Design and Implementation*. Usenix Association, 115–128. 17

SAACS May 2005. International workshop on self-adaptive and autonomic computing systems. http://cms1.gre.ac.uk/conferences/saacs-2005/. 1

Salmon, B., Thereska, E., Soules, C. A. N., and Ganger, G. R. 2003. A two-tiered software architecture for automated tuning of disk layouts. In *Algorithms and Architectures for Self-Managing Systems*. ACM, 13–18. 15

SAMBASIVAN, R. R., ZHENG, A. X., THERESKA, E., AND GANGER, G. R. 2007. Categorizing and differencing system behaviours. In *Workshop on hot topics in autonomic computing (HotAC).* 93, 151

SAMPLES, A. D. 1989. Mache: no-loss trace compaction. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* 89–97. 70, 93

SAPUNTZAKIS, C. AND LAM, M. S. 2003. Virtual appliances in the collective: a road to hassle free computing. In *Hot Topics in Operating Systems.* USENIX Association, 85–90. 25

SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. 2006. Closed versus open system models and their impact on performance. In *Symposium on Networked Systems Design and Implementation.* 30

SHEN, K., ZHONG, M., AND LI, C. 2005. I/O system performance debugging using model-driven anomaly characterization. In *Conference on File and Storage Technologies.* USENIX Association, 309–322. 17

SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. 2003. Metadata efficiency in versioning file systems. In *Conference on File and Storage Technologies.* USENIX Association, 43–58. 105

SOURCEFORGE.NET. February 2006. State Threads Library for Internet Applications. `http://state-threads.sourceforge.net/`. 64

SQLITE. 2005. SQLite. `http://www.sqlite.org`. 67

STANDARD PERFORMANCE EVALUATION CORPORATION. 1997. SPEC SFS97_R1 V3.0. http://www.spec.org/sfs97r1/. 83

STEWART, C. AND SHEN, K. 2005. Performance modeling and system management for multi-component online services. In *Symposium on Networked Systems Design and Implementation.* 17

Sun, J., Qu, H., Chakrabarti, D., and Faloutsos, C. 2005. Neighborhood formation and anomaly detection in bipartite graphs. In *European Symposium on Microprocessing and Microprogramming*. 418–425. 127

Sun Microsystems. 1989. *NFS: network file system protocol specification.* 63, 66

Sun Microsystems. 2005. Java 2 Platform, Enterprise Edition (J2EE). `http://java.sun.com/j2ee`. 53

Tesauro, G., Das, R., Jong, N., and Bennani, M. 2006. A hybrid reinforcement learning approach to autonomic resource allocation. In *International Conference on Autonomic Computing*. 65–73. 20

Thereska, E., Abd-El-Malek, M., Wylie, J. J., Narayanan, D., and Ganger, G. R. 2006. Informed data distribution selection in a self-predicting storage system. In *International conference on autonomic computing*. 187–198. 11

Thereska, E., Narayanan, D., Ailamaki, A., and Ganger, G. R. 2007. Observer: keeping system models from becoming obsolete. In *Workshop on hot topics in autonomic computing (HotAC)*. 11

Thereska, E., Narayanan, D., and Ganger, G. R. 2005. Towards self-predicting systems: What if you could ask "what-if"? In *3rd International workshop on self-adaptive and autonomic computing systems*. 10

Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. 2006. Stardust: Tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM Press, 3–14. 10

Transaction Processing Performance Council. 2002. TPC Benchmark C. `http://www.tpc.org/tpcc/Revision5.1.0`. 82

Urgaonkar, B., Shenoy, P., and Roscoe, T. 2002. Resource overbooking and application profiling in shared hosting platforms. In *Symposium* *on Operating Systems Design and Implementation.* ACM Press, 239–254. 31

Usenix May 2006. Usenix annual technical conference. http://www.usenix.org/events/usenix06/tech/tech.html. 1

Utgoff, P. E., Berkman, N. C., and Clouse, J. A. 1997. Decision tree induction based on efficient tree restructuring. *Machine Learning 29,* 1, 5–44. 129, 136

Uysal, M., Alvarez, G. A., and Merchant, A. 2001. A modular, analytical throughput model for modern disk arrays. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems.* IEEE, 183–192. 17

Valentin, G., Zuliani, M., Zilio, D. C., Lohman, G. M., and Skelley, A. 2000. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *International Conference on Data Engineering.* 101–110. 23, 152

Verbowski, C., Kiciman, E., Daniels, B., Kumar, A., Wang, Y.-M., Roussev, R., Lu, S., and Lee, J. 2006. Flight Data Recorder: Always-on tracing and scalable analysis of persistent state interactions to improve systems and security management. In *Symposium on Operating Systems Design and Implementation.* 117–130. 70, 93

Wachs, M., Abd-El-Malek, M., Thereska, E., and Ganger, G. R. 2007. Argon: performance insulation for shared storage servers. In *Conference on File and Storage Technologies.* 24, 43, 105, 106, 119, 120

Wang, M., Au, K., Ailamaki, A., Brockwell, A., Faloutsos, C., and Ganger, G. R. 2004. Storage device performance prediction with CART models. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems.* IEEE/ACM, 588–595. 18, 133

WEATHERSPOON, H. AND KUBIATOWICZ, J. D. 2002. Erasure coding vs. replication: a quantitative approach. In *International Workshop on Peer-to-Peer Systems*. Springer-Verlag. 98

WEIKUM, G., MÖNKEBERG, A., HASSE, C., AND ZABBACK, P. 2002. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *International Conference on Very Large Databases*. 20–31. 2, 15, 23

WYLIE, J. J. 2005. A read/write protocol family for versatile storage infrastructures. Ph.D. thesis, Carnegie Mellon University. 63, 98

XAFFIRE INC. October 2005. Web session recording and analysis. Xaffire Inc. web site. `http://www.xaffire.com`. 53