Early experiences on the journey towards self-* storage

Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger,
James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad,
Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen,
John D. Strunk, Eno Thereska, Matthew Wachs, Jay J. Wylie*
Carnegie Mellon University, **HP Labs, Palo Alto, CA*

Abstract

Self- systems are self-organizing, self-configuring, self-healing, self-tuning and, in general, self-managing. Ursa Minor is a large-scale storage infrastructure being designed and deployed at Carnegie Mellon University, with the goal of taking steps towards the self-* ideal. This paper discusses our early experiences with one specific aspect of storage management: performance tuning and projection. Ursa Minor uses self-monitoring and rudimentary system modeling to support analysis of how system changes would affect performance, exposing simple What...if query interfaces to administrators and tuning agents. We find that most performance predictions are sufficiently accurate (within 10-20%) and that the associated performance overhead is less than 6%. Such embedded support for What...if queries simplifies tuning automation and reduces the administrator expertise needed to make acquisition decisions.*

1 Introduction

The administration expenses associated with storage systems are 4–8 times higher than the cost of the hardware and software [2, 6, 8]. Storage systems are key parts of important data-centric applications, such as DBMSes, hence their high administration cost directly translates to higher costs for the latter. Storage system administration involves a broad collection of tasks, including data protection (administrators decide where to create replicas, repair damaged components, etc.), problem diagnosis (administrators must figure out why a system is not behaving as expected and determine how to fix it), performance tuning (administrators try to meet performance goals with appropriate data distribution among nodes, appropriate parameter settings, etc.), planning and deployment (administrators determine how many and which types of components to purchase, install and configure new hardware and software, etc.), and so on.

Like many [3, 7, 15], our goal is to simplify administration by increasing automation [5]. Unlike some, our strategy has been to architect systems from the beginning with support for self-management; building automation tools atop today's unmanageable infrastructures is as unlikely to approach the self-* ideal as adding security to a finished system rather than integrating it into the system design. We have designed, implemented, and are starting to deploy a cluster-based storage infrastructure (called Ursa Minor) with many self-management features in a data center environment at Carnegie Mellon University.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

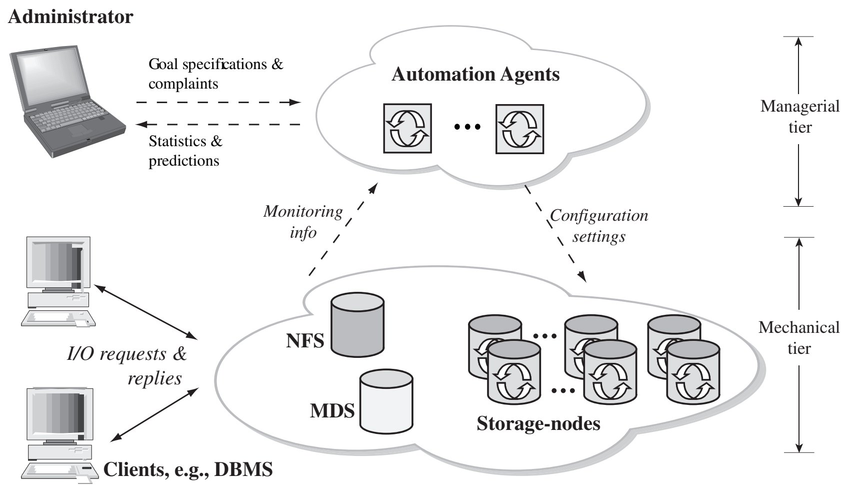


Figure 1: Architecture of Ursa Minor

Ursa Minor's high-level architecture is shown in Figure 1. The design separates functionality into two logical tiers: a mechanical tier that provides storage of and access to data and a managerial tier that automates many decision and diagnosis tasks. This separation of concerns, with clean interfaces between them, allows each tier to be specialized and evolved independently. Yet, the two tiers collaborate to simplify administration. The mechanical tier provides detailed instrumentation and status information to the managerial tier and implements decisions passed down from it.

Ursa Minor's mechanical tier consists of versatile cluster-based storage [1]. We focus on cluster-based storage, rather than traditional monolithic disk arrays, because it can simplify some aspects of administration by its nature. For example, unlike monolithic arrays, cluster-based storage naturally provides incremental scalability. This feature reduces the consequences of not over-provisioning on initial purchases and the effort involved in growth over time—one can simply add servers to the cluster as demand increases. Ursa Minor's data access protocols are versatile, allowing per-object data distribution choices, including data encoding (e.g., replication vs. erasure codes), fault model (i.e., numbers and types of faults tolerated), and data placement. This versatility maximizes the potential benefits of cluster-based storage by allowing one scalable infrastructure to serve the needs of many data types, rather than forcing administrators to select the right storage system for a particular usage at the time of purchase or migrate data from one to another as requirements change.

The managerial tier contains most of the functionality normally associated with self-* systems. It provides guidance to the mechanical tier and high-level interfaces for administrators to manage the storage infrastructure. The guidance comes in the form of configuration settings, including the data access versatility choices mentioned above. Various automation agents examine the instrumentation data exposed by the mechanical tier, as it serves client requests, to identify improvements and solutions to observed problems. These automation agents also condense instrumentation data to useful information for administrators and allow them to explore the potential consequences/benefits of adding resources or modifying a dataset's performance and reliability goals.

This paper focuses on our experiences with one specific aspect of storage administration: predicting the performance consequences of changes to system configuration. Such predictions represent a crucial building block for both tuning and acquisition decisions. Yet, such predictions are extremely difficult to produce in traditional systems, because the consequences of most configuration changes are determined by a complex interaction of workload characteristics and system internals. As such, it is a substantial source of headaches for administrators working with limited budgets.

Ursa Minor supports performance prediction with a combination of mechanical tier instrumentation and managerial tier modeling. The mechanical tier collects and exports various event logs and per-workload, per-resource activity traces [13]. The mechanical tier processes this information and uses operational laws and simple models to support *What*...*if* queries (e.g., "*What* would be the expected performance of client A's requests *if* I move its data to the set S of newly purchased storage-nodes?") [12].

Our experiences with this approach, to date, have been very positive. Instrumentation overheads are acceptable (less than 6%), and prediction accuracies are sufficiently high (usually within 10–20%) for effective decision making. This paper discusses these experiences, some lessons learned, and directions for continuing work.

2 Tuning knobs in Ursa Minor

Like any substantial system, Ursa Minor has a number of configuration options that have a significant impact on performance and reliability. In this paper, we focus on two sets on knobs: those that define the data's encoding and those that decide where to actually place the data once the encoding decision has been made. Both encoding and placement selection involve many trade-offs and are highly dependent upon the underlying system resources, utilization, and workload access patterns. Yet, significant benefits are realized when these data distribution choices are specialized correctly to access patterns and fault tolerance requirements [1]. Expecting an administrator to understand the trade-offs involved in tuning these and to make informed decisions, without significant time and system-specific expertise, is unreasonable. This section describes the encoding and placement options, and the next section explains how Ursa Minor supports choosing among them.

Data encoding: 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 [4, 10]. 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. Confidentiality requirements dictate whether or not encryption is employed. Encryption is performed prior to encoding (and decryption is performed after decoding). The basic form of *What*...*if* questions administrators would like answers to is "*What* would client A's performance be, *if* its data is encoded using scheme E?".

There is a large trade-off space in terms of the level of availability, confidentiality, and system resources (such as CPU, network, storage) consumed as a result of the encoding choice [12, 14, 16]. 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 CPU demand also increases (to encrypt the data). 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—3-way replication (i.e., a 1-of-3 encoding), for example, 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.

Data placement: In addition to selecting the data encoding, the storage-nodes on which encoded data fragments are 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, such as 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 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 A can get *if* its workload is moved to a set of storage-nodes *S*?") need answers to several sub-questions. For example, 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 workload A to/from this storage-node?"). The answer to such a question will depend on the particulars of the workload access patterns and the storage-node's buffer cache management algorithm. Then, the disk 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 workload A to/from this storage-node?"). The new network and CPU demands on each of the storage-nodes needs to be predicted as well.

3 Performance prediction support

With hundreds of resources and tens of workloads it is challenging for administrators to answer *What...if* questions such as the above. Doing so accurately requires detailed knowledge of system internals (e.g., buffer cache replacement policies) and each workload's characteristics/access patterns (e.g., locality). Traditionally, administrators use two tools when making decisions on data encoding and placement: their expertise and system over-provisioning. 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 storage system specifics are available to them, they apply these rules-of-thumb to planning challenges. Since human-utilized rules-of-thumb are rarely precise, over-provisioning is used to reduce the need for detailed decisions. Both tools are expensive, expertise because it requires specialization and over-provisioning because it wastes hardware and human resources — the additional hardware must be configured and maintained. Further, sufficient expertise becomes increasingly difficult to achieve as storage systems and applications grow in complexity.

Ursa Minor is designed to be self-predicting: it is able to provide quantitative answers to performance questions involved with administrator planning and automated tuning. Instrumentation throughout the system provides detailed monitoring information to automation agents, which use simple models to predict the performance consequences of specific changes. Such predictions can be used, internally, to drive self-tuning. They can also be exported to administrators via preconfigured *What...if* query interfaces The remainder of this section describes the two primary building blocks, monitoring and modeling, and illustrates the effectiveness with example data.

System self-monitoring: The monitoring is to be detailed so that per-workload, per-resource demands and latencies can be quantified. Aggregate performance counters typically exposed by systems are insufficient for this purpose. Ursa Minor uses end-to-end instrumentation in the form of traces of *activity records* that mark steps reached in the processing of any given request in the distributed environment. Those traces are stored in relational databases (Activity DBs) and post-processed to compute demands and latencies. The monitoring is scalable (hundreds of distributed nodes with several resources — CPU, network, buffer cache and disks) and easy to query per-workload (tens of workloads). The central idea in designing the monitoring is for it to capture the work done by each of the system's various resources, including the CPUs used for data encoding/decoding, the network, the buffer caches, and the disks. There are less than 200 instrumentation points in Ursa Minor. All those points of instrumentation are always enabled, and the overhead has been found to be less than 5-6%, as quantified by Thereska et al. [13]. As a general rule of thumb, we observe that approximately 5% of the available storage capacity is used for Activity DB storage. Different clients' access patterns generate different amounts of traces; the main insight we had from the work on the instrumentation of multiple systems [9, 13] is that it is inexpensive to monitor a distributed system that has storage at its core. This is because the rate of requests to such a system is relatively slow, since the system is usually I/O bound. We find the performance and statistics maintenance cost a reasonable performance price to pay for the added predictability.

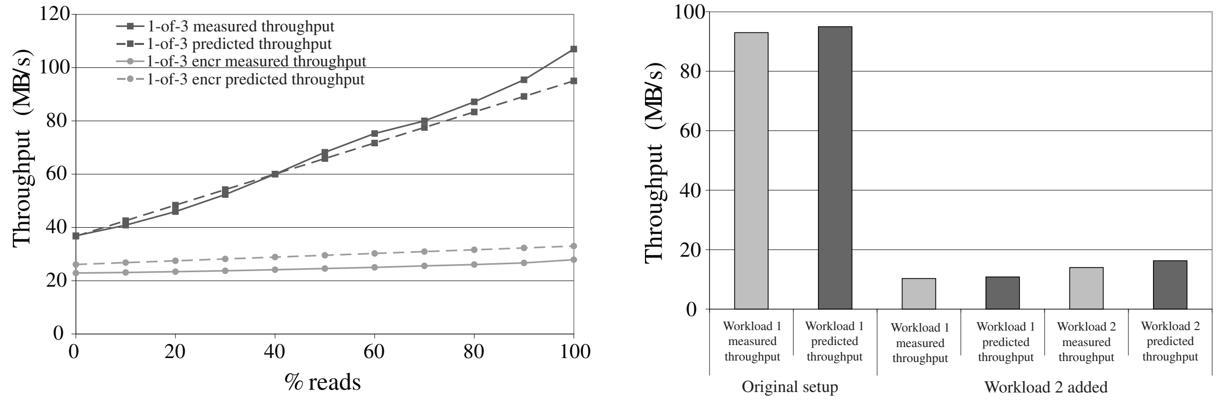
Performance modeling tools: Modules for answering *What...if* questions use modeling tools and observation data to produce answers. Tools used include experimental measurements (for encode/decode CPU costs), operational laws (for bottleneck analysis of CPU, network and disks), and simulation (for cache hit rate projections). *What*...*if* questions can be layered, with high-level *What*...*if* modules combining the answers of multiple lower-level *What*...*if* modules. 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. All *What*...*if* modules make use of the observation data collected through self-monitoring.

The basic strategy for making a high-level prediction involves consulting low-level *What*...*if* modules for four resources: CPU, network, buffer cache and disk. To predict client A's throughput, the automation agents consult these resource-specific *What*...*if* modules to determine which of the resources will be the bottleneck one. Client A's peak throughput will be limited by the throughput of that resource. In practice, other clients will share the resources too, effectively reducing the peak throughput those resources would provide if client A was the only one running. The automation agents adjusts the throughput predicted for client A to account for that.

The CPU *What*...*if* module answers questions of the form "*What* is the CPU request demand for requests from client i *if* the data is encoded using scheme E?". The CPU modules use direct measurements of encode/decode costs to answer these questions. Direct measurements of the CPU cost are acceptable, since each encode/decode operation is short in duration. Direct measurements sidestep the need for constructing analytical models for different CPU architectures. The network *What*...*if* module answers questions of the form "*What* is the network request demand for requests from client i *if* the data is encoded using scheme E?". To capture first-order effects, the network module uses a simple analytical function to predict network demand based on the number of bytes transmitted. Intuitively, schemes based on replication utilize little client CPU but place more demand on the network and storage resources (n storage nodes are updated on writes). Schemes based on erasure coding are more network and storage efficient (data is encoded in a "smart" way), but require more client CPU work to encode the data (math is needed for the "smart" way). All schemes require significant amounts of CPU work when using encryption.

The buffer cache module answers questions of the form "*What* is the average fraction of read requests $1 - p_i$ that miss in the buffer cache (and thus have to go to disk) *if* a workload from client i is added to a storage-node?". The buffer cache module can similarly answer questions on other workloads when one client's workload is removed from a storage-node. The buffer cache module uses simulation to make a prediction. The module uses buffer cache records of workloads that are to be migrated (collected through monitoring) and replays them using the buffer cache size and policies of the target storage-node. The output from this module is the fraction of hits and misses 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 too complex and system-dependent to be accurately captured using analytical formulas. The storage-node buffer cache policy in Ursa Minor is a variant of least-recently-used (LRU) with certain optimizations. The disk *What*...*if* module answers questions of the form "*What* is the average service time of a request from client i *if* that request is part of a random/sequential, read/write stream?" The average service time for a request is dependent on the access patterns of the workload and the policy of the underlying storage-node. Storage-nodes in Ursa Minor use NVRAM and a log-structured disk layout [11], which helps with making write performance more predictable (random-access writes appear sequential). When a disk is installed, a simple model is built for it, based on the disk's maximum random read and write bandwidth and maximum sequential read and write bandwidth. These four parameters are easy to extract empirically. The disk module is analytical. It receives the sequence of I/Os of the different workloads from the buffer cache *What*...*if* module, scans the combined trace to find sequential and random streams within it, and assigns an expected service time to each request.

Figure 2 illustrates the prediction accuracy for two high-level *What*...*if* questions the administrator may pose (for the exact setup of these experiments, please refer to Thereska et al. [12]). In general, we have observed predictions accuracies are within 10-20% of the measured performance [12].



(a) **Predicting peak throughput for CPU/NET-bound workloads.** "What is the throughput if I use encryption (or if I do not)?" The question is answered for different read:write ratios. CPU can be the bottleneck resource when using encryption. The network can be the bottleneck resource if the workload is write-mostly.

(b) **Predicting peak throughput for workload movements.** "What is the throughput client 2 can get if its workload is moved to a new set of storage-nodes?" The set of nodes contains a workload from client 1 that is sequential and hits in the buffer cache. When workload 2 is added, the hit rate for both drops and the interleaved disk access pattern looks like random access.

Figure 2: Prediction accuracies for two example *What...if* questions.

4 Lessons learned and the road ahead

We have had positive experiences with Ursa Minor's two-tiered architecture, particularly in the space of performance self-prediction and its application to self-tuning and provisioning decision support. With acceptable overheads, sufficient instrumentation can be continuously gathered to drive simple models that can effectively guide decisions. This sections expands on some key lessons learned from our experiences thus far and some challenges that we continue to work on going forward.

4.1 Lessons learned

Throw money at predictability: Administration, not hardware and software costs, dominate today's data center's costs. Hence, purchasing extra "iron" to allow self-prediction may be warranted. Ursa Minor utilizes "spare" resources to aid with both self-monitoring and modeling. Spare CPU is used to collect and parse trace records (we measure about 1-5% of the CPU goes towards this per machine). Spare network is needed to ship traces to collection points for processing. Spare storage is needed to store these traces and statistics (about 5% of the storage is dedicated to them). Spare CPU time is also used by automation agents to answer *What...if* questions.

Per-client, per-resource monitoring is a must: Exporting hundreds of performance counters to an administrator is counter-productive. Performance counters neither differentiate among workloads in a shared environment nor correlate across nodes in a distributed environment. The instrumentation in Ursa Minor tracks a request from the moment it enters the system until it leaves, from machine to machine. Such instrumentation is the only way to know 1) where requests spend their time, 2) what was the context during which a client experienced a performance degradation, and 3) what are the bottleneck resources for one specific workload in the distributed system.

Separate data collection from usage: We found that there is value in separating the system instrumentation from its use in specific tuning and control loops, rather than tightly coupling the two. This separation has allowed easy data access for new uses of the instrumentation, such as performance debugging. It has also allowed us to continuously refine our notions of what data are needed to make an informed tuning decision.

Rough system models work well: Resources in Ursa Minor (CPU, buffer pool, network, disks) have simple models associated with them. These models are based on direct measurements (CPU), analytical laws (network, disk) and simulation (buffer pool). These resources are complex, especially when shared by multiple workloads (e.g., the disk's performance may range over two orders of magnitude depending on the workload's and disk's characteristics). However, basic modeling works well, at least to pinpoint the bottleneck resource and give bounds on improvement if the bottleneck is removed. Furthermore, rough modeling is usually sufficient to pick one from among four or five possible configurations.

4.2 Research agenda

We are following several research directions toward making storage systems truly self-* [5], including automated data protection, problem diagnosis and repair, and of course tuning. This paper discusses our experiences with one building block: performance prediction support. Even in this one sub-area, several difficult and exciting research issues still remain:

Predicting values beyond the average: We need to develop a common terminology for how to measure predictability (and thus know when we have reached a satisfactory outcome). All our predictions so far concentrate on expected values, or averages. Making predictions about variance requires assumptions about workload patterns (e.g., Poisson arrival times) that may not hold. How can we ensure the variance is predicted within reasonable bounds as well? Can we get a notion of confidence associated with each prediction?

Co-operation with other self-* systems: How will Ursa Minor interact with other self-* systems, e.g., a DBMS that also has self-tuning at its core? The DBMS may decide to do an optimization (e.g., suggest to its administrator to double the amount of buffer cache). That change may alter the workload that Ursa Minor sees, triggering in turn an optimization from Ursa Minor (e.g., Ursa Minor could suggest to its administrator to switch the encoding from 3-way replication to 3-of-5). It is desirable for the combined DBMS+Ursa Minor system to be stable, settle on good global configurations and avoid repeating cycles of optimization. Should the DBMS micro-manage Ursa Minor's operations and optimizations, or should the DBMS convey high-level performance goals to Ursa Minor and let the latter take any necessary action to meet those goals?

Integration of legacy components: We built Ursa Minor from scratch and were thus able to insert enough detailed instrumentation inside it to answer the above *What...if* questions. However, it is convenient to be able to incorporate off-the-shelf components, such as databases, for various services within Ursa Minor (e.g., a metadata service, an asynchronous event notification service, etc). Is performance prediction possible when such legacy systems are introduced within Ursa Minor? In particular, how will we account for their resource utilization (they may use all four system resources just like clients)? What kinds of *What...if* questions can be answered for these legacy components and how fine-grained can they be?

Performance isolation for predictability: Without a basic level of performance isolation in a shared environment with competing workloads, predictions will not be meaningful. Whenever a prediction is made that workload W_n will get X MB/s of throughput (a QoS guarantee), that prediction should not be annulled when another workload W_{n+1} comes inside the system. Although performance isolation for the CPU and network resources is usually straightforward to do (utilizing well-known scheduling techniques), it still eludes researchers for the disk resource, which is traditionally non-work-conserving (the cost of a disk "context switch" is prohibitively high, on the order of milliseconds).

Acknowledgements

We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun and Symantec) for their interest, insights, feedback, and support. This work is supported in part by Army Research Office grant number DAAD19-02-1-0389, by NSF grant number CNS-0326453, and by the Air Force Research Laboratory via contract F49620-01-1-0433.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005.

[2] N. Allen. Don't waste your storage dollars: what you need to know, March, 2001. Research note, Gartner Group.

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

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

[5] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[6] J. Gray. A conversation with Jim Gray. *ACM Queue*, **1**(4). ACM, June 2003.

[7] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, **36**(1):41–50. IEEE, January 2003.

[8] E. Lamb. Hardware spending sputters. *Red Herring*, pages 32–33, June, 2001.

[9] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (Atlanta, GA, 27–29 September 2005), 2005.

[10] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, **36**(2):335–348. ACM, April 1989.

[11] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 43–58. USENIX Association, 2003.

[12] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. *International conference on autonomic computing* (Dublin, Ireland, 12–16 June 2006), 2006.

[13] E. Thereska, B. Salmon, J. Strunk, M. Wachs, Michael-Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Saint-Malo, France, 26–30 June 2006), 2006.

[14] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. *International Workshop on Peer-to-Peer Systems* (Cambridge, MA, 07–08 March 2002). Springer-Verlag, 2002.

[15] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. *VLDB*, pages 20-31, August, 2002.

[16] J. J. Wylie. *A read/write protocol family for versatile storage infrastructures*. PhD thesis, published as Technical Report CMU-PDL-05-108. Carnegie Mellon University, October 2005.