SelfTalk for Dena: Query Language and Runtime Support for Evaluating System Behavior

Saeed Ghanbari, Gokul Soundararajan and Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto

ABSTRACT

We introduce *SelfTalk*, a novel declarative language that allows users to query and understand the status of a large scale system. *SelfTalk* is sufficiently expressive to encode an administrator's high level hypotheses/expectations about normal system behavior, such as, *"I expect that the throughputs across all system components are linearly correlated"*. *SelfTalk* works in conjunction with *Dena*, a runtime support system designed to help system administrators detect the root cause of system misbehavior quickly and accurately. Given a user hypothesis, *Dena* instantiates and validates it using actual monitored data within specific system contexts. We evaluate *Dena* by posing several hypotheses about system behavior and querying *Dena* to diagnose anomalies in a virtual storage system. We find that *Dena* can automatically validate the system performance based on the user hypotheses and also accurately diagnose system misbehavior.

1. INTRODUCTION

As storage systems become exceedingly large and complex, and their applications increasingly sophisticated, system administrators find themselves incapable to diagnose and manage these systems anymore. Many industry and academic initiatives e.g., Intel's Platform Computing, HP's OpenView set of tools, IBM's Autonomic Computing [10], etc., have been launched to address this, by now, acute problem. The general consensus is that solutions for enabling storage systems to self-configure, self-heal, self-tune and therefore self-manage, are crucial for the very survival of IT over the long haul.

While we concur with this long-term view, we observe that current approaches have centered around the fallacy that a self-managing system can and should replace most or all system administrator skills. Existing approaches [3, 7, 9] following this principle did produce systems that are able to self-configure, self-heal, and self-tune to some degree. On the down side, in most of these existing systems, when something goes wrong, the system can typically produce nothing more intelligible to the administrator than generic alarms.

Indeed, whether the self-managed system is fed a costly full-fledged analytical model of its own structure and behavior, or a set of event-condition-action rules from the get-go, or left to dynamically evolve its own statistical model of functional correlations between self-monitored metrics [3, 7, 9], system self-* abilities never include the crucial capability to *self-express*, i.e., for the system to talk about its own status in a *human* intelligible way. Even worse, with very few exceptions [6], self-managed systems cannot tell the difference between unknown faults and unknown workloads or configurations, hence may alarm frequently, and unnecessarily. This is potentially very annoying, as the administrator can only start scratching their head, poring over logs, or at best a list of ranked metric correlation violations [17] upon each system alarm. This pitfall is intuitively inherent to the approach. If human children were given a set of rigid event-condition-action assertions upon birth, or left to learn entirely from their environment, without communicating with their parent, they would also find it hard to distinguish or verbalize dangerous from benign situations.

With this paper, we take the first step in evolving *self-expression* for self-managing systems beyond the hard to understand *baby cry*. Towards establishing the necessary *dialogue* between the system administrator and the self-managed system, as a cornerstone of their symbiotic relationship, we make the following contributions. We introduce (i) a new high-level declarative language, called *SelfTalk* and (ii) new runtime support, called *Dena*, for an adaptive storage system, capable of learning, self-monitoring its metrics, and evolving dynamic models of metric correlations, as well as interacting with its administrator in *SelfTalk*.

With *SelfTalk*, the system administrator can, for the first time, *easily* express her beliefs and expectations about what constitutes *normal* system behavior, ask the system to *validate* those beliefs within given contexts or periods of time, *query* the system status at any point in time, and obtain meaningful responses. Our language is powerful and can encode known generic laws that govern system behavior, such as Little's law [8] that correlates throughput and latency values, or the monotonically decreasing property of the *miss-ratio curve* (MRC) [18] in any system cache, known relationships between any metric *classes*, such as the expectation of an exponential correlation between latency and the resource quotas allocated to an application, as well as more specific administrator insights, experience with the system, or a given application.

For example, *SelfTalk* allows the human administrator to express their beliefs in close to the following high-level format: *"I expect that the average query latency measured at the database system is greater than the average data access latency measured at its back-end storage server"*. The *beliefs* do not need to be always correct, and should be viewed more as expressed *hypotheses* to the system, rather than rigid *assertions*.

An automatic parser, included with *SelfTalk*, parses each admin-

Listing 1: Invariant Hypothesis

```
HYPOTHESIS
LESS_EQ['name=cache_miss','name=cache_get']
CONTEXT []
```

Listing 2: Hypothesis with a Context

```
HYPOTHESIS
LINEAR['unit=1/s','unit=1/s']
CONTEXT ['cache_size<=512']
```

istrator hypothesis, constructs a concrete mathematical expression to describe the relationship between metric classes, and evaluates compliance by fitting the monitored data within any given context to the mathematical expression. The system thus specializes the high level *hypotheses* into concrete *internal assertions* that match each hypothesis within a particular context, and enters the *hypothesis*, matching *assertions*, contexts, and a confidence score, reevaluated periodically, into a knowledge base.

In this way, whenever a benign change in the environment, such as a workload change occurs, the system can automatically retrain any of its own internal assertions to verify whether the high level hypotheses still hold. The system alerts the administrator only in the case that one or more previously validated hypotheses do not hold anymore, and in such anomaly cases, can provide specific instances of violations of any hypotheses in the system.

As in humans, development of *self-expression* is incremental, as new guidance is received from the administrator, and new situations occur, where the guidance received is applicable and valid, or not. The status of *Dena* relative to any and all hypotheses can be queried at any point in time; *Dena* outputs to the administrator its confidence degree that the system conforms to any given hypothesis, by checking whether its monitored data fits the hypothesis well.

We perform an evaluation of our approach by posing several hypotheses to understand normal behavior, and to diagnose misbehavior in an experimental virtual storage system called *Akash* [15]. We find that *Dena* can quickly validate system performance to users' hypotheses and can help in diagnosing faults, or other system misbehavior.

2. DESIGN

We introduce novel language and runtime support for interactive diagnosis of a virtual storage system. Specifically, we design a high-level query language, called *SelfTalk*, which allows the system administrator to express generic *hypotheses* about normal system behavior, including known system laws, and relationships between metric classes. The system administrator submits *hypotheses* in *SelfTalk* to a runtime system called *Dena*, which is in charge of instantiating and *validating* them, based on automatic metric monitoring, statistics collection, and correlation at various points in the storage system. In the following, we describe our query language, the design of *Dena*, our tool, and how the administrator and the system interact to check compliance to expectations, and to alarm on anomalies, respectively.

2.1 The SelfTalk Query Language

A *hypothesis* consists of a relationship on a set of metric classes and the associated validity *context* for that relationship. The context can be a set of configurations, or workload properties that could potentially affect the given relationship. If the relationship is believed to be an *invariant*, then its corresponding context is empty. For example, a simple *invariant* that can be checked is that the number of cache misses (`cache_miss`) must be less than or equal to the number of cache accesses (`cache_gets`), as shown in Listing 1. This is an *invariant* of the cache – that is, it must hold true for all configurations and workloads. Thus, the administrator can submit the hypothesis without a context and *Dena* will check if this relationship is indeed valid for all configurations.

However, some hypotheses are valid only for particular configurations. For example, in a database system, as the rate of queries processed increases so does the rate of operations within the operating system, i.e., more I/Os per second (assuming not all data is cached). The database administrator of this system can then hypothesize *"I expect that the throughput of all components are linearly correlated"* – that is, throughput related metrics, i.e., those with units `1/seconds` are correlated. In Listing 2, we show how the above hypothesis is specified in *Dena*. It states that the throughput metrics, i.e., those with units `1/s` are expected to be linearly correlated in configurations where the `cache_size` is less than or equal to `512MB`.

2.2 The Dena Runtime System

In the following, we provide the steps taken by *Dena* when the administrator submits a *hypothesis* to the system.

1. The system automatically instantiates the hypothesis and generates a (much larger) set of *expectations*, by enumerating all possible metrics within the metric classes and configurations that match the hypothesis.

2. The system validates each expectation with experimental data, computes a confidence score per expectation and stores the expectations in a database. The system is now ready for statistics collection. A wide variety of queries are possible including querying validity of expectations over components in a sub-part of the system, confidence intervals, number of expectations generated, standard deviations, etc.

3. The administrator asks the system to create *assertions* corresponding to the generated expectations with confidence above given confidence thresholds. The system will periodically check the validity of these assertions and will trigger alarms only when specific assertions fail, signaling the part of the system where they do.

Details of Query Execution: Given a hypothesis, *Dena* creates a list of *expectations* by iteratively applying the hypothesis for each metric matching the qualifiers, $\vec{Q}$. Next, it selects a function (from a dictionary of functions) that best describes the relationship between the metrics, $R(\vec{Q})$. Then, it evaluates the validity of each *expectation* using the monitored data. We describe each step in detail next.

First, *Dena* creates a list of *expectations* by applying the hypothesis for each set of metrics matching the qualifiers. For a set of metrics, $\vec{M}$, *Dena* extracts a subset of metrics $m_i \in \vec{M}$ such that m_i matches all conditions specified in qualifier set $\vec{Q}$. For example, for the query described in Listing 2, *Dena* applies the hypothesis to all throughput metrics creating a list of expectations. In this list, one *expectation* would be `EXPECT LINEAR ('queries_per_sec','io_per_sec')`.

To support flexible matching, we annotate each metric with its metadata, specifically its unit of measurement and how the unit of measurement relates to the base unit. For example, the operating system records throughput as sectors per second (where a sector equals 512 bytes), while we would like to measure throughput in kilobytes per second. In this case, we record that the metric `io_per_sec` is measured in units `sectors/s`, and that it can be converted to our base unit by multiplying the metric value by `0.5`.

Next, *Dena* selects a function that matches the relationship described in the hypothesis. We maintain a set of pre-defined functions in a dictionary along with its description to find a match with the relation. For example, if the relationship is `LINEAR('queries_per_sec','io_per_sec')` then we match it with a function $\text{Linear}(u, v) := \{u = \alpha v + \beta\}$ and instantiate the expectation. We use simple string matching to find a function that matches the relationship. We leave the study of other approaches as future work.

Then, *Dena* takes each *expectation* and fits the function to the monitoring data. The curve is fit using an optimization algorithm, i.e., gradient descent, by varying the *free* parameters in the function. In particular, for the linear correlation between the database and storage system throughput, the curve fitting algorithm searches for values of α and β that minimize the squared error from the measured values. The curve fitting algorithm outputs a confidence score, γ, between $0 \leq \gamma \leq 1$ representing a goodness of fit where $\gamma = 1$ is good fit and $\gamma = 0$ is a poor fit. *Dena* provides the aggregate confidence score for the *hypothesis* and it allows the user to zoom-in to get per-context confidence scores as well. An *expectation* with high confidence is called an *assertion*. We keep the *assertions* in database for anomaly detection.

Handling Contexts: The relationship between metrics is influenced by the workload and other system settings. Therefore, simply fitting the expectations to all measured data would lead to false fits. Consider the expectation `EXPECT LINEAR ('queries_per_sec','io_per_sec')` and assume that we get a 50% hit ratio with a 512MB cache and a 90% hit ratio with a 1GB cache. With different cache sizes, the exact relationship between the metrics `('queries_per_sec','io_per_sec')` will be different. In fact, the factor α would be 0.5 for a 512MB cache and 0.9 for a 1GB cache. Specifically, the administrator must provide his/her belief about the contexts that the hypothesis is sensitive to. A context is simply a list of conditions on a set of performance metrics, workload metrics, or configuration parameters. In Listing 2, the context is specified as `cache_size<=512`, which states that the administrator expects the hypothesis to hold true only when the cache size is less than 512MB. We also support a wildcard operator, e.g., `cache_size=*`, that indicates that `cache_size` is a configuration parameter that may affect the fit. In this case, *Dena* will evaluate the *expectation* for each setting of the configuration separately. In particular, each *expectation* is associated with a context. A hypothesis and a context form an *expectation*, and an *expectation* with high confidence score is an *assertion*. An *assertion* must hold as long as its context is unchanged.

2.3 Runtime Monitoring with Alarms

In addition to evaluating an hypothesis, *Dena* can use the hypothesis as a condition to check system health in the future. To enable this, we save the *assertions*, the particular values of the free parameters we determine during curve fitting, and the confidence in an *assertion* database. Then, we periodically fetch assertions that match the current context of the system and re-evaluate the quality of the fit (confidence score) of each of them.

If the new monitoring data (for the current context) does not match the trained assertion then *Dena* treats the situation as a potential anomaly in the system. *Dena* first tries to re-fit the hypothesis for the current context. If the anomaly is minor, i.e., perturbations in the disk performance or a workload change, then the *hypothesis* is determined to still be valid. In this case, no alarm is raised and *Dena* simply continues running. However, if there is a misbehavior, i.e., a true fault then the re-fit of the hypothesis would fail; the re-trained assertion has a low confidence score. In this case, *Dena* raises an alarm stating that the hypothesis failed to match system behavior, indicating an anomaly.

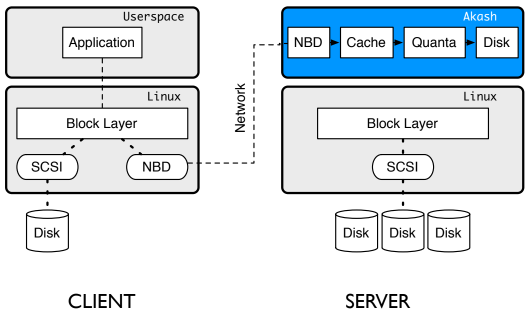


Figure 1: Testbed: We show our experimental platform. It consists of a storage server (*Akash*) and a storage client (DBMS) connected using NBD.

3. EVALUATION

Our evaluation infrastructure consists of two machines: (1) a database server running OLTP workloads and (2) a storage server running *Akash* [15] to provide virtual disks. *Akash* is a virtual storage system prototype designed to run on commodity hardware. It uses the Network Block Device (NBD) driver packaged with Linux to read and write logical blocks from the virtual storage system, as shown in Figure 1. The storage server is built using different modules:

Disk module: The disk module sits at the lowest level of the module hierarchy. It provides the interface with the underlying physical disk by translating application I/O requests into `pread()`/`pwrite()` system calls, reading/writing the underlying physical data.

Quanta module: The quanta module partitions the disk bandwidth using a quanta-based I/O scheduler [15]. The scheduler provides a fraction of the disk time to each workload sharing the disk volume.

Cache module: The cache module allows data to be cached in memory for faster access times.

NBD module: The NBD module processes I/O requests, sent by the client's NBD kernel driver, to convert the NBD packets into calls to other *Akash* server modules.

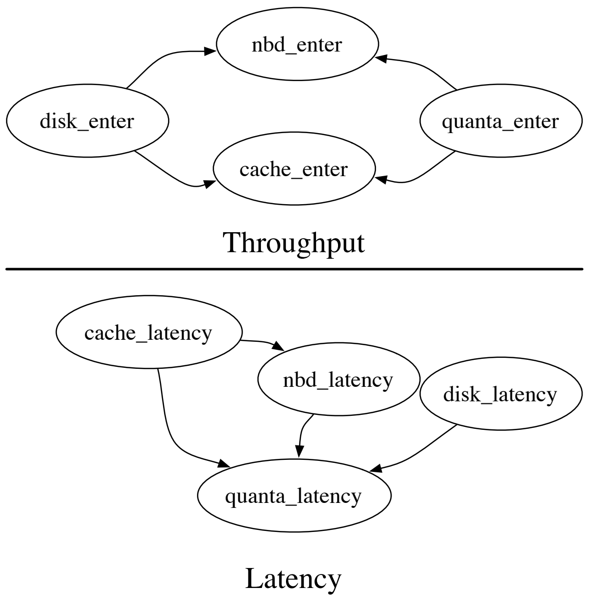
Due to space constraints, we only provide a brief description of each module. Additional details can be found in [15]. We generate a transaction processing (OLTP-like) workload using Oracle's ORION tool and by running the TPC-W benchmark using MySQL. We collect monitoring data from each module within *Akash*, which we periodically output, every 10s, to a file. We run each experiment for 1 hour, resulting in 360 samples for each configuration.

3.1 Preliminary Results

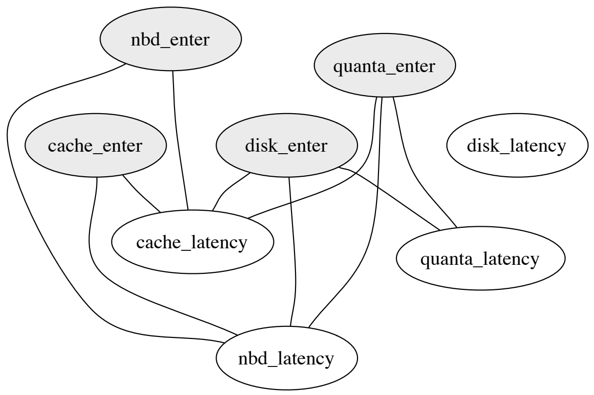
We present the correlations that *Dena* discovers for three simple hypotheses: (1) `LINEAR` – expects that metrics of the same type are linearly correlated, (2) `LESS/EQ` – states that round-trip latency is additive across layers and (3) `LITTLE` – states that throughput and latency adhere to Little's law. Table 1 shows the number of expectations generated for each hypothesis for all contexts. *Dena*

Hypothesis	Expectations	Avg. Confidence
LINEAR	3072	86%
LESS/EQ	3488	98%
LITTLE	3290	92%

Table 1: Expectations. We show the number of expectations generated for each high-level hypothesis.



(a) LESS/EQ



(b) LITTLE

Figure 2: Correlations. We show the pairwise correlations we discover for different administrator hypotheses in the above graph. The *nodes* represent different metrics and the *edges* show the correlation. The above results were gathered with a 1GB cache resulting in a miss-ratio of 50%, and the entire disk bandwidth was allocated to the application.

generates the expectations automatically for a given hypothesis.

Figure 2 shows the correlations discovered by *Dena* in a graph where the *nodes* represent metrics and the *edges* indicate a correlation. To simplify the presentation, we only show metrics related to the throughput and latency for each module. In addition, we only show results where we configure the cache to 1 GB resulting in a 50% miss-ratio and allocate the entire disk bandwidth to the application. We explain the correlations discovered for the LESS/EQ and LITTLE in detail next.

We develop the LESS/EQ hypothesis by using the information of the structure of *Akash* which allows us to hypothesize that latencies (throughput) measured in some modules are less than the latencies measured in other modules. Figure 2(a) shows our results using a directed graph where the arrowhead points from the smaller metric to the larger metric. For example, the cache module sits above the quanta module and forwards requests only on cache misses. Therefore, with a 50% miss-ratio, the latency at the cache module is less than the quanta module. This is shown as an arrow from `cache_latency` to `quanta_latency`. Conversely, the number of requests entering the quanta module is less than the number of requests entering the cache module, shown as an arrow from `quanta_enter` to `cache_enter`.

As *Akash* is closed-loop storage system, we hypothesize that performance adheres to Little's law [8] — that is, the throughput and latency metrics follow the *interactive response time law* and thus are inversely proportional. Figure 2(b) shows that indeed the system complies with Little's law as the throughput and latency metrics are indeed correlated. `disk_latency` is not correlated with Little's law as the quanta module self-adjusts its scheduling policy to varying disk service times [15]. leading to a weak correlation with the `disk_latency`.

Detecting Misbehavior: We show results showing how *Dena* can be used to detect a *misbehavior* in the system. First, we change the workload from a cache unfriendly to a cache friendly access pattern. This change is detected using our MRC hypothesis which states that *"I expect the cache misses to decrease monotonically with increasing cache size"*. Figure 3(a) shows that miss-ratio curve changes dramatically due to the workload change; the new data does not match the trained *assertion*. Rather than raise a *false alarm*, *Dena* re-evaluates the MRC expectation and finds that the cache model is still valid – that is, while the exact values of miss-ratio are different the relationship between the *miss-ratio* and cache size still follows a monotonically decreasing curve. Therefore, no alarm is raised.

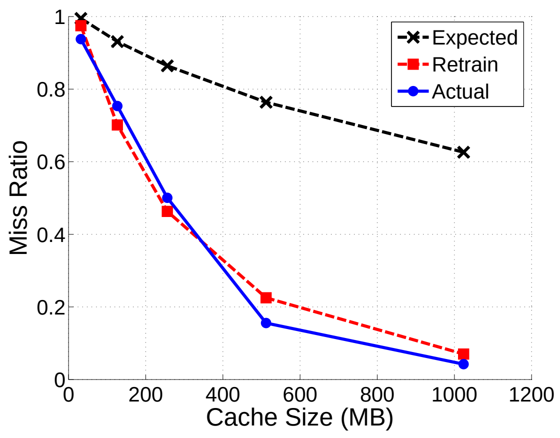
On the other hand, when we induce a fault in the cache replacement algorithm that reduces caching benefit, i.e., has more cache misses for some cache sizes, then our curve-fitting algorithm is not able to fit to the monitoring data leading to a very low confidence of $\gamma = 0.24$, shown in Figure 3(b). In this case, *Dena* raises an alarm signalling that the MRC hypothesis is violated.

4. RELATED WORK

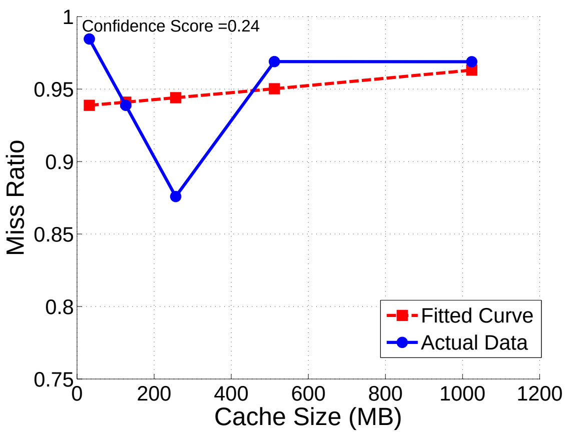
Related work in the area of fault diagnosis has focused on three approaches: (1) using statistical correlations, [1, 2, 4, 5, 9, 17], (2) using models [14, 16], and (3) using specialized languages [11, 12, 13].

The statistics based approaches assume that the system is *mostly* correct and detect anomalies as changes from the norm [1, 2]. Another approach is to use *invariants* – those metric correlations that hold in a variety of conditions as the correctness measure [9]. Cohen *et al.* [4, 5] correlate system metrics to high-level states to find the root cause of faults. PeerPressure [17] extends the analysis by comparing configuration across machines. Unlike our work, these only study simple correlations and statistical deviations, whereas we begin with a high-level hypothesis and analyze how the system's behavior matches with this hypothesis.

Model-based approaches leverage analytical models provided by the user to contrast system-behavior and localize mismatches [14,



(a) Workload Change



(b) Misbehavior

Figure 3: Detecting Misbehavior. *Dena* adapts to workload changes and signals errors on misbehaviors.

16]. The benefit of this approach is the clear relationship between the metrics and high-level system design. However, developing detailed models is difficult. While our hypotheses require an understanding of the system, we do not require the relationships described by the *hypothesis* to be always correct, and can inform the user of its validity. Language based approaches include MACE [11], TLA+ [12], and Pip [13]. They allow programmers to express their expectations about the system's communication structure, timing, and resource consumption. In contrast, we target our work towards system administrators who have a general insight into the system's behavior, but lack the knowledge of the details and have no access to the system's source code.

5. CONCLUSIONS

We introduce i) *SelfTalk*, a declarative high-level language, and ii) *Dena*, a novel runtime tool, that work in concert to allow users to interact with a running system, by hypothesizing about expected system behavior, and posing queries about the system status. Using the given hypothesis and monitoring data, *Dena* applies statistical models to evaluate whether the system complies with the user's expectations. The degree of fit is reported to the user as confidence scores. *SelfTalk* and *Dena* thus provide the basis for evolving system *self-expression* in a self-managed system towards the human-like ability to agree or disagree with the system administrator on facts and beliefs about the system in relation to given environments/contexts. We evaluate our approach on a virtual storage system called *Akash* and find that *Dena* can quickly validate user's hypotheses and accurately diagnose system misbehavior.

6. REFERENCES

[1] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, pages 259–272, 2004.

[2] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-Based Failure and Evolution Management. In *NSDI*, pages 309–322, 2004.

[3] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN*, pages 595–604, 2002.

[4] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, pages 231–244, 2004.

[5] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, pages 105–118, 2005.

[6] S. Ghanbari and C. Amza. Semantic-Driven Model Composition for Accurate Anomaly Diagnosis. In *ICAC*, pages 35–44, 2008.

[7] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking Probabilistic Correlation of Monitoring Data for Fault Detection in Complex Systems. In *DSN*, pages 259–268, 2006.

[8] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling* . John Wiley & Sons, New York, 1991.

[9] G. Jiang, H. Chen, and K. Yoshihira. Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management. *Cluster Computing*, 9(4):385–399, 2006.

[10] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[11] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, pages 179–188, 2007.

[12] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, pages 115–128, 2006.

[14] K. Shen, M. Zhong, and C. Li. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *FAST*, pages 309–322, 2005.

[15] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In *FAST*, pages 71–84, 2009.

[16] E. Thereska and G. R. Ganger. Ironmodel: Robust Performance Models in the Wild. In *SIGMETRICS*, pages 253–264, 2008.

[17] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI*, pages 245–258, 2004.

[18] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *ASPLOS*, pages 177–188, 2004.