A Query Language and Runtime Tool for Evaluating Behavior of Multi-tier Servers

Saeed Ghanbari, Gokul Soundararajan and Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto
saeed@eecg.toronto.edu, gokul@eecg.toronto.edu, amza@eecg.toronto.edu

ABSTRACT

As modern multi-tier systems are becoming increasingly large and complex, it becomes more difficult for system analysts to understand the overall behavior of the system, and diagnose performance problems. To assist analysts inspect performance behavior, we introduce *SelfTalk*, a novel declarative language that allows analysts to query and understand the status of a large scale system. *SelfTalk* is sufficiently expressive to encode an analyst's high-level hypotheses about system invariants, normal correlations between system metrics, or other a priori derived performance models, such as, *"I expect that the throughputs of interconnected system components are linearly correlated"*. Given a hypothesis, *Dena*, our runtime support system, instantiates and validates it using actual monitoring data within specific system configurations. We evaluate *SelfTalk/Dena* by posing several hypotheses about system behavior and querying *Dena* to validate system behavior in a multi-tier dynamic content server. We find that *Dena* automatically validates the system performance based on the pre-existing hypotheses and helps to diagnose system misbehavior.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Measurements, Modeling and prediction

General Terms

Verification, Measurement, Performance, Reliability

Keywords

Hypothesis, Performance Models, Management, Expectation

1. INTRODUCTION

As modern multi-tier systems increase in scale and complexity, and their applications are more sophisticated, performance validation and diagnosis of these systems become more and more challenging. Many commercial tools for monitoring and control of large-scale systems exist; tools such as HP's Openview and IBM's Tivoli products collect and aggregate information from a variety of sources and present this information graphically to users. However, the complexity of the information available with such tools exceeds the human ability to use these diagnosis tools effectively [10].

In this paper, we study high-level paradigms and advanced tools for performance analysis, and interactive performance diagnosis of multi-tier systems. We argue that the system architect, the performance analyst, and/or the system administrator have a wealth of information and expertise about the system, which can be expressed as *performance hypotheses*. *Performance hypotheses* approximate system performance metrics, as a function of other system metrics, the system structure and/or system configurations. The system architect, or analyst can provide pre-built *performance hypotheses*, in the form of statistical correlations, models, invariants, other mathematical laws or properties describing normal system behavior, for distribution to site operators. For example, *performance hypotheses* can be derived based on a variety of existing automated or semi-automated performance modelling techniques, such as, automatic statistical correlations across self-monitored metrics [3, 6, 9], black-box performance model generation by interpolation from experimental samples [20, 21], analytical system modelling [22], or gray-box approaches, where high-level semantic knowledge of the system is used to guide experimental sampling [22]. Regardless of the method for building hypotheses, we assume that they are validated based on monitoring data, and stored in a knowledge base. System administrators can then check the accuracy of existing performance hypotheses in the field, or even design hypotheses specific to their local configurations. The system analyst, or administrator can also use stored hypotheses to diagnose the behavior of the system, periodically, with newer monitoring data. For the remainder of the paper, for simplicity, we refer to the various persons that provide, monitor, and check hypotheses as analysts.

In order to assist analysts with the tasks of expressing, validating, and refining their *hypotheses* about the system, as well as diagnosing system problems interactively, we introduce a novel validation infrastructure: (i) a new high-level declarative language, called *SelfTalk*, for expressing and refining hypotheses about the system and (ii) new runtime support, called *Dena*, capable of self-monitoring system metrics, evolving, and adapting dynamic models of metric correlations, as well as interacting with its administrator in *SelfTalk*.

The relationship between metric classes within a *hypothesis* can be expressed in *SelfTalk* as a high-level, human-readable expression, such as, "linear", "less than", "equal to", or "monotonically decreasing", e.g., *"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."* Our language can also leverage existing models, or encode 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) [26] in a system cache. Each *performance hypothesis* expressed in *SelfTalk* contains as qualifiers the contexts where the hypothesis is expected to apply, i.e., the set of configurations, or workloads where the pre-computed models, or metric correlations expressed in the hypothesis hold.

By submitting a *SelfTalk* query, the analyst can validate, or refine, the existing hypotheses, or add new ones interactively, as the sys-

Listing 1: Invariant Hypothesis

```
HYPOTHESIS HYP-LESS-EQ
RELATION LESS-EQ(x,y) {
   "x.name='num_cache_miss' and
    y.name='num_cache_get'  and
    x.component=y.component"
}
CONTEXT {}
```

Listing 2: Hypothesis with a Context

```
HYPOTHESIS HYP-LINEAR
RELATION LINEAR(x,y) {
   "x.unit='1/s' and y.unit='1/s'"
}
CONTEXT (a) {
   "a.name='cache_size' and a.value<='512'"
}
```

tem configurations and the application set evolve. For this purpose, *Dena* parses each hypothesis, selects the appropriate mathematical expression that corresponds to the given keywords, and instantiates the *hypothesis* into a set of concrete *internal expectations*, one for each matching system configuration. Then, *Dena* evaluates compliance for the set of expectations by fitting the monitoring data within any given context to the mathematical expression. *Dena* enters the *hypothesis*, matching *expectations*, contexts, and a confidence score into the knowledge base. In addition, *Dena* can be configured to validate hypotheses periodically and generate alarms if previously derived correlations do not fit current monitoring data. In case of alarms, *Dena* pinpoints the hypotheses that became invalid, and the components affected, thus giving valuable feedback to the analyst.

We perform an evaluation of our approach by posing several hypotheses to understand normal behavior of the overall system, the behavior of individual components, and to diagnose misbehavior in a multi-tier dynamic content server consisting of a Apache/PHP web server, a MySQL database using virtual volumes hosted on a virtual storage system called *Akash* [22] running industry-standard benchmarks, TPC-W and TPC-C. We find that *Dena* can quickly validate system performance to analysts' hypotheses and can help in diagnosing faults, or other system misbehavior. In addition, we use *Dena* to validate different components, such as the MySQL cache, the storage system I/O scheduler, and the behavior of multi-level caches.

The rest of the paper is organized as follows. Section 2 presents the *SelfTalk* language and an overview of the overall system. We expand on the language in Section 3 and the design of *Dena* in Section 4. Section 5 describes our prototype and our experimental setup consisting of a multi-tier system consisting of MySQL databases using virtual storage on the *Akash* storage server. Section 6 presents results of our experiments showing how *Dena* allows the analysts to probe and understand the overall behavior and the per-component behavior in a multi-tier system. Section 7 discusses related work and Section 8 concludes the paper.

2. ARCHITECTURAL OVERVIEW

We introduce a novel language and runtime support for understanding and validating the behavior of a multi-tier dynamic content server system. Specifically, we design a high-level declarative language, called *SelfTalk*, which allows an analyst to express generic *hypotheses* about normal system behavior, including operational laws, and relationships between metric classes. The analyst 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 multi-tier system. In the following, we describe our language, the design of *Dena*, our tool, and how the analyst and the system interact to check compliance to expectations.

2.1 The SelfTalk 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. We provide some examples of hypotheses written in *SelfTalk*; these highlight the simplicity of the language and its ease of use. A simple *invariant* that can be checked by the analyst is that the number of cache misses (`num_cache_miss`) must be less than or equal to the number of cache accesses (`num_cache_gets`), as shown in Listing 1. This is a simple hypothesis issued by the analyst trying to understand the behavior of a cache in a multi-tier system; she does not have to know the details of the cache such as its replacement policy and only needs to have high-level understanding. She simply states that for a given cache, she expects the number of cache misses to be less than the number of cache accesses. This is an *invariant* of the cache – that is, it must hold true for all configurations and workloads. Thus, the analyst 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 analyst 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/s` 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`.

The above two examples illustrate the simplicity of the *SelfTalk* language. We strive to lower the learning curve for analysts to express the behavior of a complex multi-tier system. To achieve this, we provide simple relations (such as the `LINEAR` and `LESS-EQ` shown above) along with the system and pre-built hypotheses for common three-tier components, e.g., Apache and MySQL. However, an experienced analyst may define new metrics to monitor, create new relationships to test, and explore new facets of large multi-tier systems. We shall explain the various features of the *SelfTalk* language in detail in Section 3.

2.2 The Dena Runtime System

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

1. *Dena* 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. *Dena* 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 subsequent analysis.
3. The analyst may submit a wide variety of queries to *Dena* including querying the validity of expectations over components in a sub-part of the system, confidence intervals, number of expectations generated, standard deviations, etc.

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 that describes the relationship between the metrics, $R(\vec{Q})$. Then, it evaluates the validity of each *expectation* using the monitoring 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 HYP-LINEAR (x,y) ('name =queries_per_sec','name =io_per_sec').` Second, *Dena* selects a function that matches the relationship described in the hypothesis. We provide a set of pre-defined functions, however, the analyst may also define new relations to use with a hypothesis. For example, if the relationship is `LINEAR( 'name= queries_per _sec', 'name= io_per_sec')` then we match it with a function

$$y_{\alpha,\beta}(x_t) \quad = \quad \alpha x_t + \beta \tag{1}$$

and instantiate the expectation. Third, *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 a good fit and $\gamma = 0$ is a poor fit. *Dena* provides the aggregate confidence score for the *hypothesis* and it allows the analyst to zoom-in to get per-context confidence scores as well. We provide the details on how hypotheses are validated in Section 4.

In the following sections, we provide a detailed description of the *SelfTalk* language and the *Dena* runtime system.

3. SELFTALK LANGUAGE

In this section, we describe how a hypothesis can be declared in *SelfTalk* language and how the generated expectations can be subsequently analyzed using our query language. The *SelfTalk* language has two types of statements: a hypothesis, and a query. The *hypothesis* states the analyst's belief about the behavior of the system; it is identified by a unique name, a relation that describes a relationship between metrics, and a context that indicates the configurations affecting the validity of the hypothesis. *Dena* processes the submitted hypothesis and provides results on whether or not the analyst's beliefs match the system's behavior.

To further analyze the results, *SelfTalk* also allows the analyst to query and check the validity of the expectations; specifically, the analyst can query about the confidence of the expectations (resulting from the expansion of a hypothesis), evaluate the fit under various contexts, and for different sub-components. In addition, the analyst can obtain averages, rank the expectations, and statistically analyze the results computed by *Dena*. We describe how a hypothesis can be expressed in *SelfTalk* next; we focus on the different parts: how to specify the metrics, how to define the relation, and how to specify the validity context.

3.1 Hypothesis

```
HYPOTHESIS <hypothesisName>
RELATION <relationName> {<metricSet>}
CONTEXT {<contextSet>}
```

The *hypothesis* expresses the analyst's belief about the behavior of the system. Each hypothesis is identified by a unique name; this allows the hypothesis to be saved in a database and later retrieved for future querying. The hypothesis describes a relationship (defined as the *relation*) between metrics (selected from a *metric set*) for some system configurations (defined as the *context*). The relation defines a mathematical function describing the relationship between metrics, a set of filters to process the monitoring data (e.g., remove noise), a method to find the best fit, and a mapping to calculate the confidence score from a relation specific *goodness of fit*. The relation is identified by a *relation name* and it may be used in several hypotheses. The relation is evaluated for each combination of metrics contained in a metric set. For example, the analyst may define that she expects the throughput-like metrics to be linearly related; in this case, the relation will be evaluated for each pair of throughput-like metrics from a set of throughput-like metrics. The hypothesis can also specify a validity range – a set of contexts over which the analyst expects the relationship to hold true; the context set is described using a set of metric qualifiers; the context set however also specifies values defining the validity range. In the following, we describe each component of the hypothesis in detail next. We leave the details of the processing to Section 4.

Metric: The hypothesis describes a relationship between tuples of metrics where each tuple is selected from a *metric set* (also referred to as the *metric class*). The metric set, in turn, is constructed by a *join* of the available metrics (denoted as $\mathcal{M}$); in more detail, a hypothesis may define a relationship between two metrics x and y then, the metric set contains tuples of the form $< x_i, y_j >$ chosen from $\mathcal{M}^2 = \mathcal{M} \times \mathcal{M}$ according to the join condition. In general, *Dena* supports metric sets of more than two metrics. The metric set is constructed from an expression evaluated on each metric's attributes; the metrics that match the expression are included in the metric set. Each *metric* is a primitive entity that can be a performance measurement, a configuration setting, or a composite of several base performance metrics. Each metric has several attributes such as its name (e.g., `queries_per_sec`), the component name (e.g., MySQL) from where it is measured, the location of the component (e.g., hostname of the MySQL instance), and its unit of measurement (e.g., `query/sec` for throughput). For example, the measure of query throughput, `queries_per_sec` metric is defined as

```
METRIC queries_per_sec AS (
  number id,
  text component  = 'MySQL',
  text location  = 'cluster101',
  text unit  = '1/sec',
  number value
)
```

where the MySQL database is running on hostname `cluster101`. Configuration parameters are represented as metrics as well (e.g., `mysql_cache_size`); the configuration metrics are used to establish a context for the hypothesis. In some cases, it is useful to define a composite metric built from a combination of several primitive metrics. The composite metric may be defined persistently within the *Dena* system or temporarily by inlining the definition with the hypothesis. For example, for the cache, it is useful to define the cache miss-ratio as a composite metric that is computed as the ratio of number of cache accesses (`num_cache_gets`) and the number of cache misses (`num_cache_misses`). The metric set is constructed from the description of metrics given with the hypothesis; *Dena* selects the metrics by matching the attributes to the conditions specified in the expression (similar to the SQL `JOIN` and a `WHERE` clause). The attributes of a metric are optional (except `id` and `name`) and the metric can be thought of a schema-less relation; we use only the specified metric attributes to check a metric for inclusion into the metric set. The expression allows us to specify very broad qualifiers to capture a large set of metrics, or be very specific and capture metrics of a specific component. For example, we can express a relation between a set of throughput metrics, we specify the qualifiers as `"x.unit ='1/sec' and y.unit ='1/sec'"`, or we can express the metrics of a specific cache by specifying `"x.name='cache_hits' and y.name='cache_gets' and x.location=y.location"`.

Relation: The correlation between a set of metrics is described by a relation. The relation includes functions to filter the data, a mathematical function describing the relationship, an error function (e.g., squared error), a method to compute a best-fit (e.g., gradient descent), and a method to compute the confidence score. Many of these functions (e.g., the gradient descent optimizer and the method to compute the confidence score) are independent of the specific relation and may be shared by several relations. We follow an object-oriented paradigm to implement relations; we explain the details of our implementation in Section 5.1. To illustrate, we show a *SelfTalk* code snippet of the *linear* relation that is provided with the *Dena* runtime system.

```
DEFINE RELATION linear {
  PARAMETER a,b : number,
  INPUT x:number-array, y:number-array,
  ...
  FUNCTION confidence
  {
    OUTPUT confidence:number
    LANGUAGE 'matlab'
    SCRIPT
    $
      y_hat = a.*x .+ b;
      confidence = R2(y,y_hat);
      //calculate residuals
      ...
    $
  }
  ...
}
```

It shows the relation containing two parameters and two input data arrays; the parameters refer to the slope and y-intercept of the linear line and the two input arrays correspond to the input and output data values obtained by monitoring the system. We focus on the function to compute the confidence of the relation; the confidence score is a number between 0.0 and 1.0 representing how well the hypothesis fits the monitoring data. In the example, we specify the confidence as the R^2 (implemented as a MATLAB script delimited by $) and we also check the residuals before returning the confidence score.

Context: The relationship between metrics is influenced by the workload and other system configuration settings – referred to as the *context* of the hypothesis. Therefore, simply fitting the expectations to all measured data would lead to false fits. Consider the expectation `EXPECT LINEAR ('name=queries_per_sec', 'name=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 analyst must provide 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 `name=cache_size and value<=512`, which states that the analyst expects the hypothesis to hold true only when the cache size is less than 512MB. We also support a wild-card operator, e.g., `name=cache_size and value=*`, to indicate 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.

3.2 Query

Dena expands the hypothesis submitted by the analyst into expectations, fits each expectation to the monitoring data, and stores the results in a database; these results can be further analyzed by submitting *queries* written in *SelfTalk*. The analyst can query about the confidence of the expectations that result from the expansion of the hypotheses, evaluate the fit under various contexts and for different sub-components. We categorize the queries into two types: i) queries that focus the analysis on particular components, configurations, or confidence values, and ii) queries that modify the presentation of the results by ordering them based on confidence score, or grouping them by particular metrics, or by grouping them by the configuration type.

The general syntax of a *SelfTalk* query is

```
1 QUERY <HYP-NAME>
2   [METRIC <METRICS-SET>]
3   [CONTEXT {<CTX-SET>}]
4   [CONFIDENCE {<|>|=|>=|<= <VALUE>>}
5                   |{<IN> <RANGE>}]
6   [ORDER BY CONFIDENCE [ASC|DSC]]
7   [RANK BY CONFIDENCE [ASC|DSC]]
8   [GROUP BY METRIC <METRIC>...<METRIC>]
9   [GROUP BY CONTEXT <CONTEXT>...<CONTEXT>]
```

It consists of three parts: i) the preamble – we need to specify the name of the hypothesis being queried, e.g., the hypothesis name (shown in line 1), ii) the query focus – we narrow the analysis by specifying conditions on the metric set, the context set, and the confidence score (lines 2-5) and iii) the presentation of results – the results may be displayed by controlling the ordering based on the confidence score, and by grouping using a certain metric attributes or contexts (lines 6-9). We present the details of how queries enable analysis of the results using two examples next.

All queries include a hypothesis name; the hypothesis name is used to find the results stored by *Dena* in the database. If no options are specified, the results of all expectations that are generated from the hypothesis are returned — that is, the results all possible expansions (*expectations*) of the metric set and context set declared in the hypothesis. This is equivalent to the `SELECT *` construct in SQL; *SelfTalk* allows the fine-grained analysis to be done with ease by restricting the analysis to certain sub-components and for certain contexts. For example, the analyst may issue

```
QUERY HYP-LINEAR
METRIC (x,y) {
  "x.component='MySQL' and x.unit='1/sec'
   and
   y.component='Akash' and y.unit='1/sec'"
}
CONTEXT (a) {
  "a.name='mysql_cache_size' and a.value=512"
}
CONFIDENCE > 0.9
```

that returns results from expectations of the linear hypothesis (named `HYP-LINEAR`) for throughput-like metrics measured at the *Akash* storage server and MySQL only for configurations where the size of the MySQL cache is configured to 512MB and those expectations with a confidence score greater than 0.9.

In addition to allowing focused analysis of the results, *SelfTalk* allows the analyst to control the presentation of the results of a query by grouping, ordering, and ranking. We can analyze the effect of changing the size of the MySQL cache on the throughput by stating

```
QUERY HYP-LINEAR
METRIC (x,y) {
  "x.component='MySQL' and x.unit='1/sec'
   and
   y.component='Akash' and y.unit='1/sec'"
}
ORDER BY CONFIDENCE DSC
GROUP BY CONTEXT (a) {
  "a.name='mysql_cache_size' and a.value=*"
}
```

to return expectations from the execution of the linear hypothesis (named `HYP-LINEAR`) for throughput-like metrics collected from MySQL and the *Akash* storage server, grouped by MySQL cache configurations (where the confidence scores are computed as the average for each cache configuration) and sorted by the confidence score in descending order.

4. VALIDATING EXPECTATIONS

Dena expands the hypothesis posed by the analyst to generate a larger set of *expectations* by enumerating all possible metrics and configurations that match the hypothesis. In this section, we describe the steps taken by *Dena* to validate each expectation with the monitoring data and compute the confidence score.

4.1 Overview

An *expectation* is validated by evaluating how well the relationship described by the hypothesis applies to the monitoring data. At its core, we apply statistical regression techniques to fit a function (describing the relationship between metrics) and evaluate the *goodness of fit*. While statistical regression techniques have been studied in great detail elsewhere [17], the three main challenges exist in the implementation of a generic engine; we need to (1) process monitoring data collected from many different sources, (2) evaluate various relationships on the monitoring data, and (3) compute a mapping from the relationship specific *goodness of fit* to a human-understandable confidence score.

The first challenge arises from the fact that monitoring data from a component contains noise and that monitored values from multiple components may not be aligned in time. Thus, we first filter the data to make it suitable for statistical regression; filtering removes the outliers in the collected data and aligns the time-series data. After filtering, we can evaluate if the relation matches the monitoring data. The second challenge is that the statistical regression techniques differ for different types of relations; while at the heart of all expectations is a mathematical function describing a relationship between a set of monitored metrics, the method of fitting the function differs from closed-form solutions (e.g., for linear regression) to iterative methods such as gradient descent. Finally, we need to compute a confidence score – a human understandable output between 0.0 (low confidence) and 1.0 (high confidence) from the relation-specific *goodness* metric. To aid in the design of a generic engine, we evaluate a set of commonly asked questions by analysts and build a taxonomy of relations. In the following, we describe the taxonomy of relations and describe each of the steps in more detail. Then, we provide a list of sample relations used to evaluate the behavior of a multi-tier system.

4.2 Taxonomy

A relation describes a mapping between several metrics. Each relation specifies a function $\hat{y} = \hat{f}(x)$ that describes how two metrics x and y are expected to be correlated. The relationship may be *comparisons* – where the mapping between x and y is a boolean operator e.g., $y < x$ or *regressions* – where the mapping between x and y is a mathematical function e.g., $y = ax + b$. In addition, each of the relationships may be time dependent e.g., $\hat{y}_t = \hat{f}(x_t)$ or time-independent.

We classify the relations into different categories using the above criteria as shown in Figure 1. The relations are first classified into two categories: *regressions* and *comparisons*. The relations classified into *regressions* are functions that describe a mathematical relationship between several metrics. An example of a *regression* relationship is a linear relationship between two metrics; the function mapping x to y is described by $\hat{y}_{\alpha,\beta}(x) = \alpha x + \beta$. The validity of these relations can be evaluated using statistical regression techniques. The second relation type is a *comparison* where the mapping between two metrics is a comparison operator $(<, >, =, \leq, \geq)$. In this case, directly applying statistical regression techniques is difficult. Thus, we evaluate the validity of these

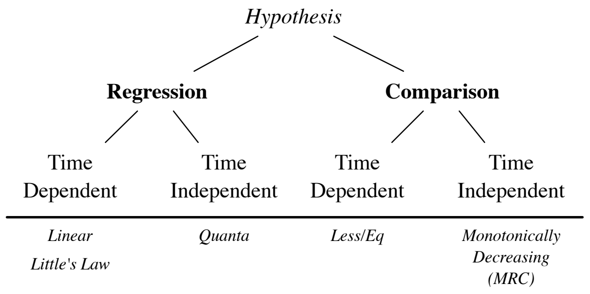


Figure 1: Relation Taxonomy: We classify the relations into different categories.

relations using simple counting ; we validate the relation by counting the fraction of points where the comparison holds true. Each of the above two categories (*regressions* and *comparisons*) can be applied to time-dependent or time-independent data. Time-dependent relations treat the input as time-series which relation between input metrics are considered through time; the input data to the relation is tuples of metric values with same time-stamps. On the other hand, time-independent relations treat the data as an unordered list. We explain the details next.

4.3 Evaluating Expectations

The evaluation of an expectation consists of three steps: (1) collect and filter monitoring data, (2) apply statistical regression and evaluate for monitoring data, and (3) compute the confidence score.

Step 1 – Filtering Monitoring Data: The monitoring data collected from components have two sources of error: (1) noise in the data collected from one component, and (2) mis-alignment of data collected from multiple components. We filter the data values before evaluating the relationship.

The noise in the monitoring data is seen as outliers in the data. The outliers occur when data is collected from components during their initialization phases either at start-up or after a configuration change, and due to interference from background tasks. One such example is the measurement of the `cache_hits` (the number of cache hits) and `cache_misses` (the number of cache misses) from a cache. During the initialization phase (i.e., cache warm-up), the cache misses are high as many of cache accesses experience *cold* misses since the cache is empty. However, as the cache warms up, the number of cache misses reduces steadily (conversely the number of cache hits increases steadily) until the values reach steady state. Similarly, infrequent background tasks from the operating system or transient network bottlenecks introduce noise in the measurements as well. We filter these outliers before applying statistical regression. The analyst can instruct *Dena* to apply any filtering technique. We choose to use *percentile filtering* due to its simplicity. Percentile filters are generic; they make no assumption about the distribution of data other than that the number of samples is large enough to cover most regions of the underlying distribution. We use percentile filtering to trim the top $t\%$ and the bottom $b\%$ of sampled data. By removing these samples, the percentile filter keeps the samples which form the majority in the distribution. Based on experience and insight about the process of collecting monitoring data, the analyst may specify filtering thresholds $t\%$ and $b\%$ thereby overriding the default values. The filtering process is different for time-independent relations; in these, we perform percentile filtering per configuration value rather than on the entire dataset.

Time-series data pose an additional challenge where the data measured at different components may be misaligned due to clock skew as well as due to causality between components. We evaluate time-series data by matching (i.e., *joining* in the database terminology) the sampled values using the timestamp. Causality between components can also account for some misalignment between the sampled metrics. For example, a change in the workload is reflected at the metrics collected at the higher layers (e.g., the database) before it is seen in the metrics collected in the lower layers (e.g., disk). While there are various sophisticated methods for aligning time-series data, we find that simple techniques of grouping values using a coarser-time granularity and using moving average filters work well; for example, we align the data values by grouping them into a coarse timestamp granularity (e.g., 10 seconds). We also use a moving-average filter. A moving average is used to analyze a set of data points by creating a series of averages of adjacent subsets of the full data set; this smooths out short-term fluctuations while maintaining the long-term trends. Aligning time-series data by estimating the clock skew and delay between components is an area for improvement; we leave this optimization as future work.

Step 2 – Performing Regression: After filtering the monitoring data, we perform statistical regression to evaluate how well the hypothesis fits the measured values. We find the best values for *free parameters* to reduce the squared error between the hypothesis and the measured values. For example, consider the linear relation,

$$\hat{y}_{\alpha,\beta}(x_t) \quad = \quad \alpha x_t + \beta \tag{2}$$

with two free parameters α – the slope of the line, and β – the y-intercept of the line. The best fit of the relation to the measured data is obtained when the squared-error between the predicted values and the measured values is minimized. We define the error (i.e., how the relation deviates from the measured values) as

$$\xi(\alpha, \beta) \quad = \quad \sum_{<x,y>} (y - \hat{y}_{\alpha,\beta}(x))^2 \tag{3}$$

and we find the best-fit of the relation by minimizing the squared error by mapping the problem of reducing the squared error as an optimization problem and use standard optimization techniques such as gradient descent (using the partial derivatives if given) to find the best parameter values. In some cases, the best parameter values can be obtained from closed-form solutions (such as for linear regression); we opt for the closed-form solution rather than iterative search in these cases.

Step 3 – Computing the Confidence Score: After applying statistical regression and optimizing the free parameters, we evaluate how well the relation describes the data and report the *confidence score*. The *confidence score* is a human-understandable number between 0.0 and 1.0, which indicates a poor and good fit respectively. The evaluation of the confidence score is dependent on the relation – whether the relation is a *comparison* or *regression*.

For the *comparison* relations, the confidence score is the fraction of data when it holds `true`; we count the number of times the comparison evaluates to `true` and divide by the total number of monitoring data points. For *regression* functions (i.e., those with a mathematical relationship), we use the coefficient of determination, R^2, to compute the confidence score. The R^2 is a fraction between 0.0 and 1.0. An R^2 value of 0.0 indicates that the function does not explain the relationship between the two metrics. Assuming a relation is defined as $\hat{y} = \hat{f}(x)$, the coefficient of determination is defined as

$$R^2 \quad = \quad 1 - \frac{SS_{err}}{SS_{tot}} \tag{4}$$

$$SS_{err} \quad = \quad \sum_i (y_i - \hat{f}(x_i))^2 \tag{5}$$

$$SS_{tot} \quad = \quad \sum_i (y_i - \bar{y})^2 \tag{6}$$

where SS_{err} is the residual sum of the squares, SS_{tot} is the total sum of squares, and $\bar{y} = mean(y)$. However, simply using R^2 to evaluate the fit may be incorrect. To better evaluate the fit, we perform a secondary test using the *residuals* of the regression; the residuals are the vertical distances from each point in the fitted line to the monitoring data. A good fit has the residuals equally above and below the fitted line. If the residuals are not randomly scattered – indicating a systematic deviation from the fitted line then, the R^2 value may be misleading; thus we report that the fit has a low confidence score.

4.4 Validating Performance of a Multi-tier Storage System

In this section, we provide a sample of hypotheses that we issue to understand and validate the behavior of a multi-tier storage system consisting of a MySQL database using a virtual volume hosted on a storage server. The details of the storage system are given in Section 5.2. We choose one or two hypotheses from each of the categories we describe in the relation taxonomy. For each hypothesis, we provide the high-level question the analyst is probing, the underlying regression/comparison function tested in the hypothesis, the filtering applied to the monitoring data, and the optimization algorithm used to find the best fit.

Time-dependent Regression – Linear/Little: The `LINEAR` hypothesis is one of the simplest hypothesis that an analyst can issue to *Dena*; we issue this hypothesis to diagnose traffic patterns along the storage path. Specifically, as an analyst, we ask the question – *"I expect the throughput measured at the storage system to be linearly correlated with the throughput measured at MySQL"* or more generally *"I expect the throughput metrics along the storage path to be linearly related"* with the belief that as we increase the load at the MySQL database, the load on the underlying storage server will increase correspondingly. The linear relation is defined as

$$\hat{y}_{\alpha,\beta}(x_t) \quad = \quad \alpha x_t + \beta \tag{7}$$

with two free parameters: α and β. We filter the time-series data by first removing the outliers using *percentile filtering* and then smoothing the values with a *moving average filter*. The line is fit to the monitoring data using linear regression and we use the *coefficient of determination* (R^2) as the confidence score. We further verify the fit using the residuals to determine if the data does not systematically deviate from the hypothesis. If the residuals are not valid, we report that the hypothesis is not a good fit.

Dena can incorporate results from models, such as those derived from operational laws, to verify the behavior of a multi-tier system; an example of this is the `LITTLE` hypothesis that defines a relationship between throughput and latency using Little's law [8]. Little's law states that if the system is stable then, the response time and throughput are inversely related; we issue this hypothesis to verify that the behavior of the system adheres to the behavior explained by operational laws; a stable system follows these laws. For example, the analyst can express her belief in operational laws by making a high-level hypothesis that *"I expect the throughput measured at the storage system is inversely correlated with the latency measured at MySQL"*. For an interactive system, such multi-tier storage systems, Little's law is expressed as

$$\hat{X}_{\mathcal{N},\mathcal{Z}}(R_t) \quad = \quad \frac{\mathcal{N}}{R_t + \mathcal{Z}} \tag{8}$$

with two free parameters: $\mathcal{N}$ and $\mathcal{Z}$, which are number of clients and average think time respectively, and X_t and R_t denoting throughput and response time. Similar to the processing of `LINEAR` relation, we filter the data by first removing the outliers using *percentile filtering* and then smoothing the values with a *moving average filter*. The curve is fit to the monitoring data using gradient descent optimization, and we use the *coefficient of determination* (R^2) as the confidence score. We further verify the fit using the residuals to determine if the data does not systematically deviate from the hypothesis; if the residuals are not valid, we report that the hypothesis is not a good fit.

Time-independent Regression – Quanta: Our storage system uses the quanta-based scheduler to divide the storage bandwidth among several virtual volumes. The quanta-based scheduler partitions the bandwidth by allocating a time quantum where one of the workloads obtains exclusive access to the underlying disk. For modeling the quanta latency, we observe that the typical server system is an *interactive*, closed-loop system. This means that, even if incoming load may vary over time, at any given point in time, the rate of serviced requests is roughly equal to the incoming request rate. Then, according to the *interactive response time law* [8]:

$$L_d = \frac{N}{X} - Z \tag{9}$$

where L_d is the response time of the storage server, including both I/O request scheduling and the disk access latency, N is the number of application threads, X is the throughput, and Z is the think time of each application thread issuing requests to the disk. We then use this formula to derive the average disk access latency for each application, when given a certain fraction of the disk bandwidth. We assume that think time per thread is negligible compared to request processing time, i.e., we assume that I/O requests are arriving relatively frequently, and disk access time is significant. Then, through a simple derivation, we arrive at the following formula

$$L_d(\rho_d) = \frac{L_d(1)}{\rho_d} \tag{10}$$

where $L_d(1)$ is the *baseline disk latency* for an application, when the entire disk bandwidth is allocated to that application. This formula is intuitive. For example, if the entire disk was given to the application, i.e., $\rho_d = 1$, then the storage access latency is equal to the underlying disk access latency. On the other hand, if the application is given a small fraction of the disk bandwidth, i.e., $\rho_d \approx 0$, then the storage access latency is very high (approaches ∞). The QUANTA hypothesis expresses the above belief from operational law model where we expect the storage access latency of the application to be inversely related to the allocation time fraction. The QUANTA hypothesis uses the *inverse* relationship that is described as

$$\hat{y}_{\alpha,\beta}(x) = \frac{\alpha}{x^\beta} \tag{11}$$

where the waiting time at the scheduler (y) is inversely related with the time fraction (x) given to the application. We filter the latency values using the percentile filter and average the samples (for each quanta setting) before performing regression. We find the best-fit for the free parameters using gradient descent and we use R^2 as the confidence score and use the residuals as a secondary check.

Time-dependent Comparison – Less/EQ: The LESS/EQ hypothesis is used to answer many storage questions. For example, the analyst can check on a configuration parameter — *"I expect the current size of the cache is less than or equal to the maximum size (as defined in the configuration)"* or check on a performance metric — *"I expect the latency (e.g., response time) measured at higher level components (MySQL) is higher than the latency measured at the lower level components (disk)"*. We remove the outliers using percentile filtering and use a moving average filter to synchronize the samples over time. There is no regression step and we report the confidence score as the fraction of samples where the comparison ($\leq$) holds true.

Time-independent Comparison – MRC/Constant: The miss-ratio curve (MRC) relation describes the behavior of a cache; it states that the cache miss-ratio (i.e., the ratio of cache misses to the cache accesses) is a *monotonically decreasing* curve with respect to the cache size. We capture this relationship in two ways: by comparing to a user-provided miss-ratio function or systematically checking that the curve is indeed monotonically decreasing. In the first case, the analyst may provide the expected miss-ratio curve from a model (i.e., using Mattson's stack algorithm [13]) or from a cache simulator; with either approach, we are given a list of tuples of the form $\langle c, m \rangle$ (where c is the cache size and m is the miss-ratio) and we evaluate the confidence using R^2. In the second method, for each cache size c, we obtain the values of the miss-ratio and apply the percentile filter; the filtering concentrates the miss-ratio samples into a cluster (for each cache size c). Then, we average the miss-ratios and use the resulting list $\langle c, \bar{m} \rangle$ of tuples (where $\bar{m}$ is the average of the miss-ratios for cache size c) to sort by cache sizes in ascending order and verify that the miss-ratio keeps decreasing (or remains flat) as the cache size is increased. We count the fraction of times the comparison holds true and report it as the confidence score.

The versatile CONST hypothesis checks if the values of a metric are constant; we use this relation to issue hypothesis of the form *"I expect that the size of the cache (i.e., the number of items stored in the cache) remains constant"*. We note that there is a small fraction of time (during start-up) when the size is not equal to the capacity which is filtered by the percentile filter. We filter the data using percentile filter to remove outliers and return high confidence if samples are almost constant – that is, the variation in the values is within a small ratio of its mean; we compute the ratio of the mean of x and divide by the standard deviation. If the ratio is less than a threshold, we report a confidence score of 1, else we report a confidence score of 0.

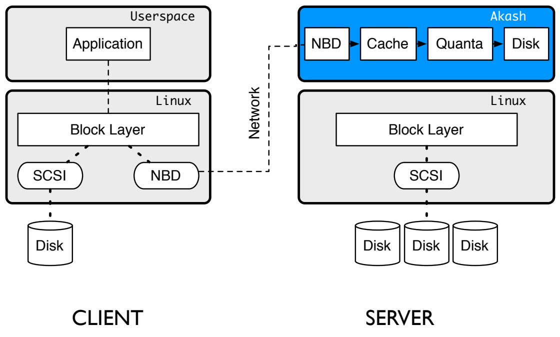


Figure 2: Testbed: We show our experimental platform. It consists of a storage server (*Akash*) and a storage client (DBMS) connected using NBD.

5. TESTBED

In this section, we describe the implementation of *Dena* and our experimental multi-tier infrastructure consisting of a MySQL database running on a virtual storage system, called *Akash*.

5.1 Prototype Implementation

The *Dena* runtime system is composed of multiple parts: a front-end consisting of the *SelfTalk* parser, a core regression engine, and a database backend storing the monitoring data. The monitoring data is collected from existing software; we use built-in instrumentation such as the MySQL/InnoDB monitor to get statistics from the database, vmstat and iostat to obtain statistics from the operating system, and built-in instrumentation from our storage server. We implement the core of the statistical regression algorithms using MATLAB utilizing JDBC to fetch data from the backend database. We provide simple relations that can be utilized by an analyst new to *Dena*; this includes all the relations we describe in Section 4.4 plus we provide relations describing exponential and polynomial curves, and all boolean comparisons.

The analyst can specify the hypothesis at the command-line or by referring *Dena* to a file; given a hypothesis *Dena* parses the details and expands the hypothesis to all possible expectations. *Dena* instantiates a new object for each expectation, obtains the data from the database, fits the relation to the monitoring data, and computes the confidence score. When the fitting is complete, the details of the hypothesis, the set of expectations, the final fitted values of the free parameters, and the descriptions of the contexts are stored into the database for future analysis.

5.2 Platform and Methodology

Our evaluation infrastructure consists of two machines: (1) a database server running OLTP workloads and (2) a storage server running *Akash* [22] 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 2. 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 [22]. 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 [22].

We use three workloads: a simple micro-benchmark, called UNIFORM, and two industry-standard benchmarks, TPC-W and TPC-C. We run our Web based applications (TPC-W) on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database engine. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the *Akash* server. We run the *Akash* storage server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. To maximize I/O bandwidth, we use RAID-0 on 15 10K RPM 250GB hard disks. Non-web applications (TPC-C) utilize the same MySQL and storage server instances; however, they do not use the machine running the Apache web server. The monitoring data is collected from the underlying operating system (using Linux utilities `vmstat` and `iostat`), the MySQL database, and the *Akash* storage server. The collected metrics are timestamped using `gettimeofday()`.

Specifically, we use the metrics that were collected over a period of 6 months [22]. The collected data includes storage-level metrics (from the *Akash* storage server), database-level metrics (gathered by instrumenting MySQL), and OS-level metrics (using `vmstat`). We collected the data for two physical machines, i.e., the database machine and the storage machine, and for four applications, i.e., four virtual disk volumes and database instances. The collected metrics, after pruning, result in over 10GB of data represented as flat files; we load these files into the database for analysis.

6. RESULTS

We evaluate the efficacy of *Dena* to validate overall system behavior and to understand per-component behavior. To achieve this, we issue broad high-level hypotheses describing the relationships in a multi-tier storage system and check the validity of these relationships. Next, we issue specific queries to provide insights into the behavior of a specific component and also one component's effect on other components within the multi-tier system. Then, we present additional results studying cases where there is a mismatch between the analyst's belief and the monitoring data. Finally, we present measurements calculating the cost and time breakdown of executing a hypothesis.

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.

6.1 Understanding the Behavior of the Overall System

We issue several broad high-level hypotheses to check the overall behavior of the system. 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 the Little's law. Table 1 shows the number of expectations generated for each hypothesis for all contexts. *Dena* generates the expectations automatically for a given hypothesis. Figure 3 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.

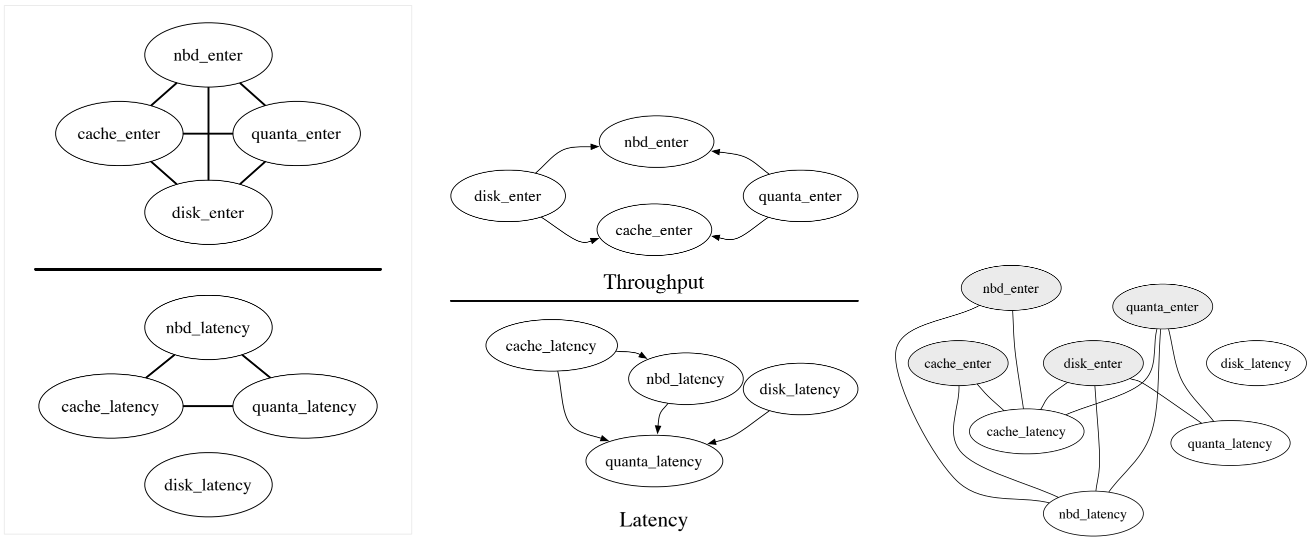
For the `LINEAR` hypothesis, shown in Figure 3(a), we find two clusters of metrics: a set of throughput related metrics and a set of latency related metrics. First, we see that the set of throughput metrics are linearly correlated. This is expected as the storage is configured as a single path from the NBD module to the disk module (see Figure 2). The cache and quanta modules do not affect the linear correlation between the throughput seen in the NBD module (`nbd_enter`) and the disk module (`disk_enter`) because while the cache causes less I/Os to be issued to disk, an increase in the rate of I/O requests entering the storage system still results in a corresponding increase in the rate of disk I/Os. Similarly, latency across components is linearly correlated as well except the quanta module; it controls the number of requests entering disk leading to an additional queuing delay between the `disk_latency` and the `quanta_latency` breaking the linear relationship across latencies [22].

We develop the `LESS/EQ` hypothesis by using the information of the structure of *Akash* which allows us to hypothesize that latencies (similarly throughput) measured in some modules are less than the latencies measured in other modules. Figure 3(b) 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 a 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 3(c) 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 [22] leading to a weak correlation with the `disk_latency`.

6.2 Understanding Per-Component Behavior

Next, we explore the behavior of different storage server components by studying the correlations found using different hypotheses.



(a) LINEAR (b) LESS/EQ (c) LITTLE

Figure 3: Correlations. We show the pairwise correlations we discover for different analyst 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.

We focus on the two major components: the cache and the quanta scheduler modules within *Akash*. Then, we present results showing how *Dena* can be used to study interactions between multiple components as well; to illustrate this, we focus on the effect of cache inclusiveness in multi-tier caches.

Understanding the Cache: We study the effect of caching on the performance of the storage system by issuing several hypotheses that provide an insight into its behavior: MRC – indicates the analyst's belief that the cache performance will improve (i.e., its miss-ratio will decrease) as the size of cache is increased, LESS/EQ – states that caching improves performance by reducing latency where the latency to access items from the cache is lower than the latency of accessing items from the underlying disk, and LINEAR – states the belief that since the cache size impacts performance, the linear relation between metrics must account for the size of the cache as a context. We evaluate these beliefs using the UNIFORM workload which has a miss-ratio of 75% with a small cache (256MB), 50% with a medium cache (512MB) and a 12% with a large cache (896MB). Figure 4 shows the results of the MRC and LINEAR hypotheses. The results from the TPC-W workload are similar and we omit these results due to space constraints.

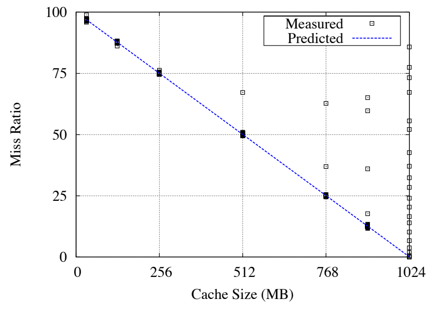
Figure 4(a), shows the miss-ratio for the UNIFORM workload. As expected, the miss-ratio is monotonically decreasing – a straight line from approximately 1.0 (many misses) with a small cache to near 0.0 (many hits). *Dena* computes a confidence score of 0.99 for the miss-ratio curve. Regardless of the cache size, caching provides a benefit in terms of performance. This improvement can be checked using the LESS/EQ hypothesis; *Dena* reports a confidence score of 1.0 for all cache sizes indicating that the throughput measured at the cache is higher than the throughput at the underlying disk and the latency at the cache is lower than the latency of fetching data from disk.

The detailed impact on the performance from different cache sizes can be obtained by issuing the LINEAR hypothesis as seen in Figures 4(b)- 4(c). Each plot shows three lines corresponding to three cache sizes: a small cache (shown in red with squares), a medium cache (shown in green with triangles), and a large cache (shown in blue with circles). The points are the samples (before percentile filtering) obtained through monitoring and the line is the best-fit of the relation described in the hypothesis. The plots show that performance can indeed be improved by increasing the size of the cache; the throughput ratio between the cache and the disk (i.e., the factor of improvement) is 1.25, 2, and 8 for small, medium, and large cache sizes respectively. Similar factors are seen in the reduction of the access latency at the cache and the underlying disk latency. These results also indicate the importance of filtering for accuracy of hypothesis verification.

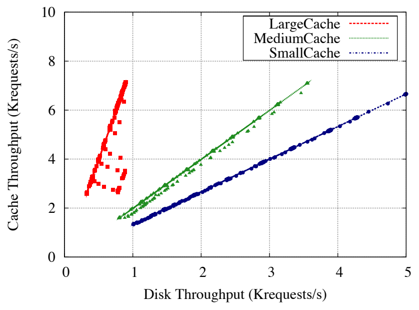
Understanding the Quanta Scheduler: The quanta scheduler is the mechanism *Akash* uses to proportionally allocate the disk bandwidth among multiple storage clients. As we describe in Section 4.4, the effect on performance can be modeled using operational laws. In this case, we observe that the *Akash* is a closed-loop system where the rate of serviced requests is roughly equal to the incoming request rate. Then, by using the *interactive response-time law*, we derive the relationship that the latency as seen from the quanta module varies inversely with fraction of the disk bandwidth allocated to the workload – that is, as the fraction of disk bandwidth is halved, the per-request latency doubles.

Figure 5 presents the results obtained from *Dena* for the UNIFORM workload. It shows three curves showing the results for the small, medium, and large cache sizes. In addition, we plot the measured values of the quanta latency for comparison. The results show that our belief that the latency varies inversely to the disk bandwidth fraction is correct; the fitted curve closely matches the observed values resulting in confidence scores of 0.94, 0.94, and 0.93 for the small/medium/large caches respectively. Using the QUANTA hypothesis allows us to understand the disk performance as well. Specifically, *Dena* shows that the confidence score for the large cache is slightly smaller than the small and medium cache sizes. The reason is that there is a higher variability of the average disk latency when (i) the underlying disk bandwidth isolation is less effective due to frequent switching between workloads and (ii) disk scheduling optimizations are less effective and reliable due to fewer requests in the scheduler queue. However, even with this variability, the underlying relationship is still inverse leading *Dena* to report a high confidence score.

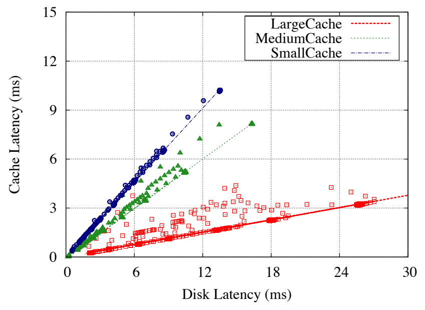
Understanding Two-tiers of Caches: In a multi-level cache hierarchy using the standard (uncoordinated) LRU replacement pol-



(a) MRC



(b) Throughput



(c) Latency

Figure 4: Understanding the Cache Behavior: We look at the impact of caching on the performance of the storage server by studying the miss-ratio curve and comparing the the throughput and latency across the cache module within *Akash*.

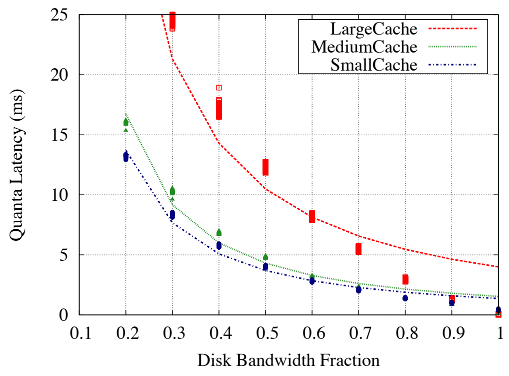


Figure 5: Understanding the Quanta Behavior: We see that the impact of the quanta scheduler is inverse where halving the disk bandwidth fraction leads to a doubling of the quanta latency.

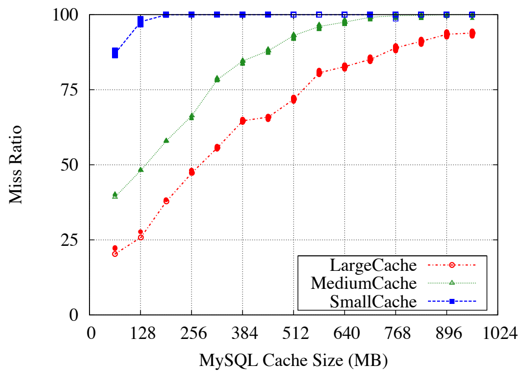


Figure 6: Understanding the Two-tier Cache Behavior: We see the effect of cache inclusiveness in the miss-ratio at the second-level cache. The miss-ratio increases steadily as the size of the first-level cache is increased.

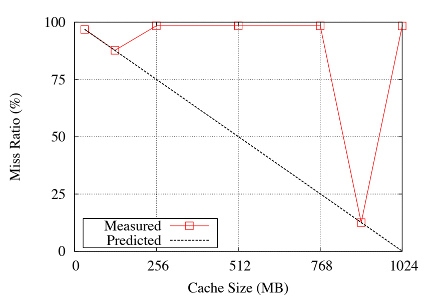
icy at all levels, any cache miss from cache level q_i will result in bringing the needed block into all lower levels of the cache hierarchy, before providing the requested block to cache i. It follows that the block is *redundantly* cached at all cache levels, which is called the *inclusiveness* property [25]. Therefore, if an application is given a certain cache quota ρ_i at a level of cache i, any cache quotas ρ_j given at any lower level of cache j, with $\rho_j < \rho_i$ will be mostly wasteful. We can verify this behavior using two hypothesis based on the MRC hypothesis. Due to cache inclusiveness, the analyst expects that by increasing the size of the first-level cache (i.e., the MySQL buffer pool) the performance of the second-level cache (i.e., the miss-ratio at the storage server cache) steadily decreases due to lower temporal locality.

We perform the analysis by stating that the relationship between the miss-ratio at the storage cache and the size of the MySQL buffer pool size is monotonically increasing; the context of the hypothesis is the storage cache size. Given this hypothesis, *Dena* presents these results grouped by each storage cache size. We present the results graphically for the TPC-W workload; the results from TPC-C are similar. Figure 6 shows this behavior for three different storage cache sizes: small (128MB), medium (512MB), and large (896MB) where the lines indicate the best-fit regression and the points are measured values. For the small storage cache (shown in blue with squares), we see that the miss-ratio is high at 80% for small MySQL buffer pool sizes but quickly increases to 100% for medium to large MySQL buffer pool sizes. For a large storage cache (shown in red with circles), the effect is more clear; the miss-ratio for a small MySQL cache is less than 25% but the miss-ratio worsens steadily as the MySQL cache is increased where it crosses 50% after 512MB of MySQL buffer pool and over 90% for very large MySQL cache sizes.

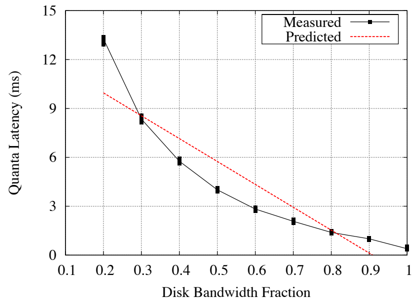
6.3 Understanding Mismatches between Analyst Expectations and the System

There can be a mismatch between the analyst's beliefs and the monitoring data; this can occur either due to a fault in the system or from a misunderstanding of the system by the analyst. In either case, *Dena* reports low confidence scores and the analyst may probe deeper by issuing different hypotheses to diagnose faults or to improve her understanding of the system. In the following, we present three cases of mismatch; we test for cases where (i) the system is faulty – we induce a fault in the cache resulting in errors in the cache replacement policy, (ii) the hypothesis is faulty – we hypothesize that the behavior of the quanta scheduler is *linear*, and (iii) the context is faulty – we hypothesize the metrics of the same type are linearly correlated but fail to provide the context information that the size of the storage cache may affect the relationship.

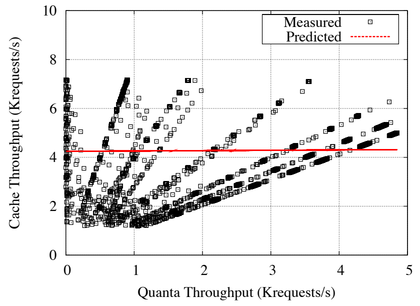
System is Faulty: In the first case, we show results showing how *Dena* can be used to detect a *fault* in the system; we detect a fault in the cache replacement policy using the MRC hypothesis which states that *"I expect the cache misses to decrease monotonically with increasing cache size"*. We run the UNIFORM workload for this experiment; in an earlier case, we have shown that the UNIFORM workload has a straight line as the miss-ratio curve, shown in Figure 4(a), and that with a fault-free cache replacement algorithm, the curve is indeed monotonically decreas-



(a) System is Faulty



(b) Hypothesis is Faulty



(c) Context is Faulty

Figure 7: Different Errors: *Dena* does not expect the analyst to issue correct hypotheses or the system to behave correctly. In both cases, there is a mismatch between the analyst and the system leading to low confidence scores. We show three such cases.

ing. Now we induce a fault in the cache replacement algorithm that reduces caching benefit; it has more cache misses than expected for some cache sizes as shown in Figure 7(a). Due to the fault, *Dena* is not able to validate the relationship using the monitoring data; this leads *Dena* to report a very low confidence score of 0.24. This scenario highlights one use-case where the analyst is confident in her hypothesis and thus can conclude that the system is faulty.

Hypothesis is Faulty: Another case where there is a mismatch between the analyst and the system is if the analyst's belief is incorrect; we test a case by issuing the hypothesis that we expect the *"latency of the quanta module is linearly related with the disk bandwidth fraction"*. During the design phase of *Akash*, we made a similar assumption; we noticed that the throughput of the storage system varies linearly with the disk bandwidth fraction (by applying Little's law) and incorrectly concluded that the effect on latency is linear as well. We have shown that the relationship is indeed inverse earlier in Figure 5; the error is noticed by *Dena*, as shown in Figure 7(b), where the expected line does not match the monitoring data. In this case, *Dena* reports a confidence score of 0.8. This scenario describes the second use-case where the analyst initiates a dialogue to understand the behavior of the system by issuing hypotheses (correctly or incorrectly) and obtaining feedback on its validity.

Context is Faulty: In the last case, we re-issue the LINEAR hypothesis but fail to identify that the size of the cache may affect the validity of the hypothesis. With an incorrect context, the relation cannot be fit; as Figure 7(c) shows, the data values form several lines with different slopes and y-intercepts and no *single* line satisfies the monitoring data. With an incorrect context, the best-fit of a line is a *null* fit and the confidence score is 0.0.

6.4 Cost of Hypothesis Execution

We also evaluate the cost of executing a hypothesis by measuring the time taken to fetch the data from the database and the time needed to perform statistical regression.

Our knowledge base is stored in a relational DBMS (PostgreSQL) and we use JDBC to fetch the data to be used by MATLAB for data processing. Our analysis shows that a majority of time is spent fetching the data from the DBMS and not in data processing (MATLAB). However, we also performed further analysis by considering monitoring data over longer time intervals (6 months) thereby stressing MATLAB. In this longer time interval, with roughly 1.5M samples, the time spent inside MATLAB is under 3 seconds. Similarly, in this case, the majority of the time is spent fetching the data from the DBMS/Disk.

In more detail, Figure 8 presents our results for queries accessing upto a week of monitoring data. It shows that a large fraction of the time is spent fetching the data from the database and a small fraction spent doing statistical analysis. Specifically, our results show

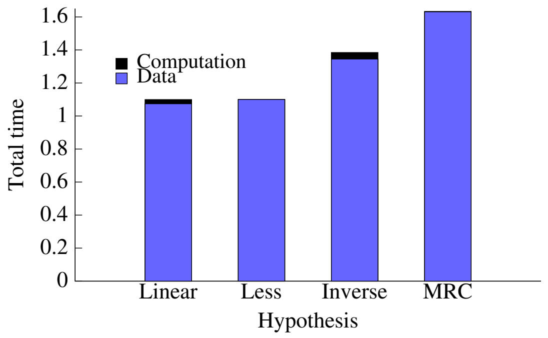


Figure 8: Timing of Hypothesis Execution: We measure the time to execute an expectation and notice that the bulk of the cost is fetching the data from the database while the time needed to perform statistical regression is small.

that it takes roughly 1 to 1.5 seconds (average) to fetch the data for an expectation and less than 40ms to find the best-fit. The computation cost is the least for comparison relations; these perform simple counting thus require less than 5ms to report the confidence score. The regression cost is higher as we need to fit the line to the monitoring data; the time needed to find the closed-form solution for LINEAR is 25ms and the time needed for QUANTA (inverse) is 39ms on average.

7. RELATED WORK

Related work in the area of fault diagnosis has focused on four approaches: (i) using statistical correlations [1, 2, 4, 9, 24], (ii) using models [18, 23], (iii) using specialized languages [11, 12, 16], and (iv) using fine-grained profiling [5, 7].

The statistics based approaches assume that the system is *mostly* correct and detect anomalies as changes from the norm. PeerPressure [24] extends the analysis by comparing configuration across machines. Pinpoint [2] and Magpie [1] are statistical tools for fault detection in component-based Internet service. 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] correlate system metrics to high-level states to find the root cause of faults. In contrast to 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 analyst to contrast system-behavior and localize mismatches

[18, 23]. 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 analyst of its validity.

Language based approaches include MACE [11], TLA+ [12], PCL [14], PSpec [15] and Pip [16]. They allow programmers to express their expectations about the system's communication structure, timing, and resource consumption. PSpec [15] is a performance checking assertion language that allows system designers to verify their expectations about a wide range of performance properties. The type of assertions of PSpec are similar to *SelfTalk* comparison relations. Also, similar to *SelfTalk*, PSpec uses a relational approach to represent and query monitoring data [19]. However, PSpec lacks the ability to use mathematical functions as the basis of checking the behavior of the system. Similar to our work, Pip [16] is an infrastructure for comparing actual behavior and expected behavior of a distributed system, expressed through a declarative language. However, unlike our work, Pip requires the source to be modified by adding some special annotations. In general, in contrast to the existing language based approaches, our work mainly targets users such as performance analysts 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.

Performance analysis tools [5, 7] allow programmers to analyze the performance of a system to find sources of inefficiencies. These approaches are useful for a programmer who has detailed insight about the system and has access to fine-grained profiling. While our approach can be used for the same purposes, it also provides, through hypotheses, a common knowledge shared by the parties involved in different life cycles of a system, mainly post-deployment where effective maintenance is the priority.

8. CONCLUSIONS

We introduce *SelfTalk* – a declarative high-level language, and *Dena* – a novel runtime tool, that work in concert to allow analysts 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 analyst's expectations. The degree of fit is reported to the analyst as confidence scores. We evaluate our approach on a multi-tier dynamic content web server consisting of a Apache/PHP web server, a MySQL database using storage hosted by a virtual storage system called *Akash* and find that *Dena* can quickly validate analyst's hypotheses and helps to accurately diagnose system misbehavior.

Acknowledgements

We are grateful to our shepherd, Erich Nahum, and the anonymous reviewers for their valuable feedback and suggestions. We also thank Kim Keeton for her feedback and for improving the quality of our earlier *HotStorage'09* publication of this work. Finally, we extend our thanks to the Systems Group at University of Toronto for their excellent suggestions and comments.

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

[5] M. E. Crovella and T. J. LeBlanc. Performance debugging using parallel performance predicates. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 140–150, New York, NY, USA, 1993. ACM.

[6] 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.

[7] K. A. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. D. Malony, L. C. McInnes, and B. Norris. Capturing performance knowledge for automated analysis. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–10, Piscataway, NJ, USA, 2008. IEEE Press.

[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] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. In *IBM System Journal*, pages 78–117, 1970.

[14] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[15] S. E. Perl and W. E. Weihl. Performance assertion checking. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 134–145, New York, NY, USA, 1993. ACM.

[16] 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.

[17] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

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

[19] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Trans. Comput. Syst.*, 6(2):157–195, 1988.

[20] G. Soundararajan and C. Amza. Towards End-to-End Quality of Service: Controlling I/O Interference in Shared Storage Servers. In *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference (Middleware'08)*, pages 287–305, 2008.

[21] G. Soundararajan, J. Chen, M. A. Sharaf, and C. Amza. Dynamic Partitioning of the Cache Hierarchy in Shared Data Centers. *Proceedings of the VLDB Endowment*, 1(1):635–646, 2008.

[22] 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.

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

[24] 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.

[25] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *USENIX*, pages 161–175, 2002.

[26] 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.