CHAMELEON: a self-evolving, fully-adaptive resource arbitrator for storage systems

Sandeep Uttamchandani[†] Li Yin[‡] Guillermo A. Alvarez[†] John Palmer[†] Gul Agha[*]
[†] IBM Almaden Research Center, 650 Harry Rd., San José, California 95120, USA
[‡] Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA
[*] Department of Computer Science, University of Illinois at Urbana-Champaign.

Abstract

Enterprise applications typically depend on guaranteed performance from the storage subsystem, lest they fail. However, unregulated competition is unlikely to result in a fair, predictable apportioning of resources. Given that widespread access protocols and scheduling policies are largely best-effort, the problem of providing performance guarantees on a shared system is a very difficult one. Clients typically lack accurate information on the storage system's capabilities and on the access patterns of the workloads using it, thereby compounding the problem. CHAMELEON *is an adaptive arbitrator for shared storage resources; it relies on a combination of self-refining models and constrained optimization to provide performance guarantees to clients. This process depends on minimal information from clients, and is fully adaptive; decisions are based on device and workload models automatically inferred, and continuously refined, at run-time. Corrective actions taken by* CHAMELEON *are only as radical as warranted by the current degree of knowledge about the system's behavior. In our experiments on a real storage system* CHAMELEON *identified, analyzed, and corrected performance violations in 3-14 minutes—which compares very favorably with the time a human administrator would have needed. Our learning-based paradigm is a most promising way of deploying large-scale storage systems that service variable workloads on an ever-changing mix of device types.*

1 Introduction

A typical consolidated storage system, in which multiple clients store and access petabytes' worth of data [21], serves the needs of various, independent, paying customers (e.g., a storage service provider) or divisions within the same organization (e.g., a corporate data center). Consolidation has proven to be an effective remedy for the low utilizations that plague storage systems [12], for the expense of employing scarce system administrators, and for the dispersion of related data into unconnected islands of storage. In the *utility* model, each client is guaranteed a portion of the shared resources regardless of whether other clients over- or under-utilize their allocations. Purchasing costs play a dwindling role relative to *managing* costs in current enterprise systems [12].

This paper addresses the problem of allocating resources in a fully automated, cost-efficient way so that most clients experience predictable performance in their accesses to a shared, large-scale storage utility. Although performance is just one of the dimensions of Quality of Service (*QoS*), it is the most critical and less understood. Static provisioning approaches to providing performance isolation and guaranteed performance are far less than optimal, given the high variability (e.g., burstiness) of I/O workloads and the incomplete characterizations of storage device capabilities[8]. Furthermore, static resource allocations do not contemplate hardware failures, load surges, and workload variations; system administrators must currently deal with those by hand, as part of a slow and error-prone observe-act-analyze loop. Prevalent access protocols (e.g., SCSI and FibreChannel) and resource scheduling policies are largely best-effort; unregulated competition is unlikely to result in a fair, predictable resource allocation.

Previous work on this problem includes management policies encoded as sets of rules [14, 28], heuristic-based scheduling of individual I/Os [8, 16, 19, 13], decisions based purely on feedback loops[18, 9] and on the predictions of models for system components[3, 2, 4]. The resulting solutions are either not adaptive at all (as in the case of rules), or dependent on hard-to-obtain models, or ignorant of the system's performance characteristics as observed during its lifetime.

This paper's main contribution is to demonstrate a novel technique for making automatic throttling decisions, based on a combination of performance models, constrained optimization, incremental feedback, and policies. CHAMELEONis a framework in which clients whose Service Level Agreement (*SLAs*) are not being met get access to additional resources freed up by *throttling* (i.e., rate-limiting) [8, 18] competing clients. Our goal is to take more accurate corrective actions as we learn more about the characteristics of the running system, and of the workloads being presented to it. As shown in Figure 1, CHAMELEON operates at any point in a continuum between decisions made based on relatively uninformed, deployment-independent

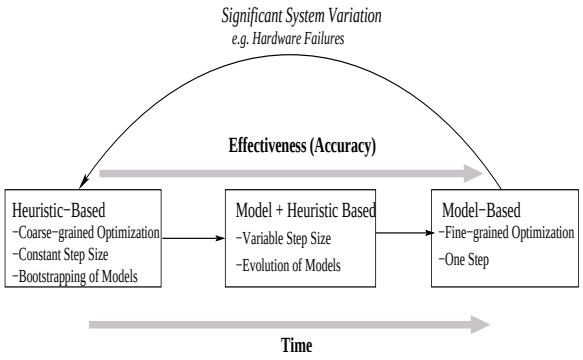


Figure 1. CHAMELEON **moves along the line according to the quality of the predictions generated by the internally-built models at each point in time.**

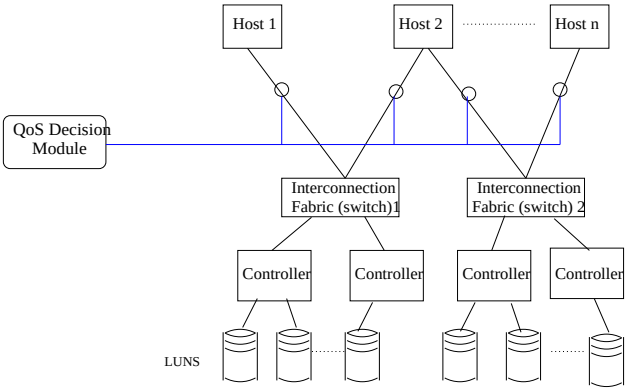


Figure 2. System model: CHAMELEON **has access to performance data captured between the hosts and the storage back-end, and can effect throttling decisions at the same points. Such instrumentation points can be co-located with logical volume managers or block-level virtualization appliances [11].**

heuristics, and on blind obeyance to models of the particular system being managed (Figure 1). This is done by having the throttling step size be a function of the statistical confidence of the models built automatically by CHAMELEON; when that confidence falls below a threshold, we revert to generic base heuristics.

By striking a balance between sub-optimal policies, uninformed feedback control, and model-based predictions, CHAMELEON can react to workload changes in a nimble manner, resulting in a marginal number of QoS violations. In our experiments on a real storage system using real-world workload traces, CHAMELEON managed to find the set of throttling decisions that yielded the maximum value of the optimization function, while minimizing the amount of throttling required to meet the targets and while satisfying the QoS requirements of most clients. Since it does not depend on prior knowledge about devices or workloads, our approach can be easily deployed on large-scale storage systems that service variable workloads on an ever-changing mix of device types. Our ultimate vision, of which CHAMELEON is just a part, is to combine automatic workload, device, and action characterization with machine learning, in order to solve a variety of systems management problems while operating on incomplete information [26].

The structure of this paper is as follows. Section 2 describes the architecture of CHAMELEON. We then proceed to describe the main components: the models (Section 3), the reasoning engine (Section 4), the base heuristics (Section 5), and the feedback-based throttling executor (Section 6). Section 7 describes our prototype and presents experimental results. Section 8 reviews previous research in the field, and Section 9 presents our conclusions and future directions.

2 Overview of CHAMELEON

CHAMELEON is a framework for providing predictable performance to multiple clients accessing a common storage infrastructure, as shown in Figure 2. Multiple hosts connect to storage devices in the *back end* in such a way that CHAMELEON can monitor every I/O processed by the system, thus gathering information on the access patterns, throughput and latency. Each workload has a known SLA associated with it, and uses a fixed set of physical components (such as controllers, disks, switches), and logical components (such as logical volumes) that are together referred to as its *invocation path*. CHAMELEON detects workloads whose SLAs are not being met, and solves the violations by identifying and throttling workloads whose resource consumption should be curtailed. It also periodically checks for unused bandwidth, and selectively unthrottles some workloads.

Our SLAs are *conditional*: they specify maximum average I/O latencies over short sampling periods, as long as workloads request up to a maximum number of bytes and I/Os (throughput) during said periods. If workloads inject load into the system at more than the rate prescribed in their SLAs, the system is under no obligation of guaranteeing any bound on latency. Obviously, such rogue workloads are prime choices for resource restriction; but in some extreme cases, well-behaved workloads may also need to be restricted. Throttling is effected at the client hosts by the leaky bucket protocol [25] where each workload is given tokens every 10 ms., and I/O rates are averaged over a sliding window of 1200 s. for comparison with the SLA.

The core of CHAMELEON consists of four parts, as shown in Figure 3:

- **Knowledge base**: by taking periodic performance samples on the running system, CHAMELEON builds internal representations of system behavior without any human supervision; these we encapsulate using *black-box models*. Our black-box models are mathematical functions that quantify the capabilities of each component in the system, the demands placed by each

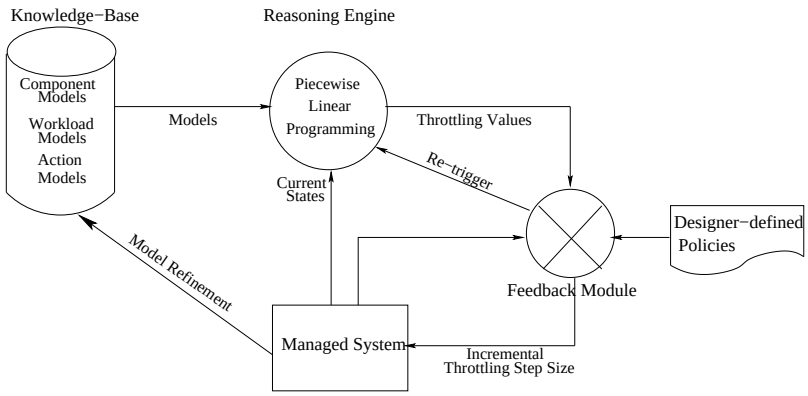


Figure 3. Architecture of CHAMELEON**.**

workload on each component, and the reaction of each workload to different levels of throttling. Models get better as time goes by, for CHAMELEON refines them automatically; they may bootstrap from a *tabula rasa*, or from convenient oversimplifications (e.g., an M/M/1 queueing system) for faster convergence.

- **Model-based optimization**: CHAMELEON decides the workload throttle values using constrained optimization techniques such as piecewise linear programming. Our *constraints* compare load and performance as predicted by the models against the SLAs. Many possible administrator-defined *objective functions* can be used to reflect the business goals of the storage utility, e.g., "minimize the number of SLA violations", or "ensure that highest priority workloads always meet their guarantees". Based on the errors associated with the models, the output of the constraint solver is assigned a *confidence value*.

- **System-designer policies**: As a fallback mechanism, we maintain a set of fixed heuristics specified by the system designer for system-independent, coarse-grain resource arbitration. Examples include "throttle all workloads sharing any component with the workload in trouble", or "greedily throttle lower-priority workloads as long as high-priority SLA violations exist".

- **Informed feedback loop**: The general guiding principle is to take radical corrective action as long as that is warranted by the available knowledge about the system. If the confidence value from the solver is below a certain threshold (e.g., during bootstrapping of the models), CHAMELEON falls back on the fixed policies to make decisions. Otherwise, the feedback module applies throttling decisions incrementally: the step size is a function of the confidence value. After each iteration, the feedback module has the option of continuing to apply prior decisions incrementally, or querying the reasoning engine to re-evaluate throttling decisions (e.g., when it observes abrupt changes in the system's behavior).

3 Knowledge base

CHAMELEON builds models in an automatic, unsupervised way. It uses them to characterize the workload being presented to the storage system, and the expected response of system components to different load types and intensities.

Models based on simulation or emulation require a fairly detailed knowledge of the system's internals; analytical models require less, but device-specific optimizations must still be taken into account to obtain accurate predictions [27]. *Black-box* models are built by recording and correlating inputs and outputs to the system in diverse states, without regarding its internal structure. We chose them because of properties not provided by the other modeling approaches: black-box models can evolve with changes in the component behavior, workload characteristics, and action effects, and they make very few assumptions about the phenomena being modeled. Because of this, black-box models are an ideal building block for an adaptive, deployment-independent management framework that doesn't depend on preexisting model libraries.

At the same time, the black-box models used in CHAMELEON are less accurate than their analytical counterparts; our adaptive feedback loop compensates for that. The focus of this paper is to demonstrate how several building blocks can work together in a hybrid management paradigm; we do not intend to construct good models, but to show that simple modeling techniques are adequate for the problem. CHAMELEON's models are constructed using Support Vector Machines (*SVM*) [17], a machine-learning technique for regression. This is similar to the CART [30] techniques for modeling storage device performance, where the response of the system is measured in different system states and represented as a best-fit curve function. Table-based models [3], where system states are exhaustively recorded in a table and used for interpolation, are not a viable solution as they represent the model as a very large lookup table instead of the analytic expressions that our constraint solver takes as input.

Black-box models depend on collecting extensive

amounts of performance samples. Some of those metrics can be monitored from client hosts, while others are tallied by each component—and collected via proprietary interfaces for data collection, or via standard protocols such as SMI-S [22].

A key challenge is bootstrapping, i.e., how to make decisions when models have not yet been refined. There are several solutions for this: run a battery of tests in non-production mode to generate baseline models, or run in a monitor-only mode until models are sufficiently refined, or use a pre-packaged library. We follow different approaches for different model types; but in all cases models are incrementally refined from performance observations, while the level of confidence in their predictions increases. We proceed to discuss how models are represented internally, bootstrapped and refined from performance observations.

3.1 Component models

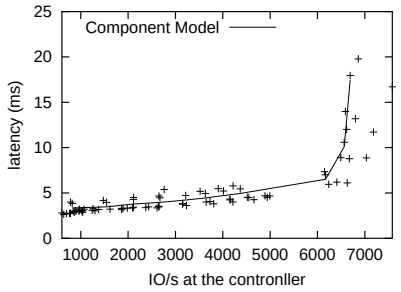


Figure 4. Component model.

A component model predicts values of a delivery metric as a function of workload characteristics. CHAMELEON can in principle accommodate models for any system component. In particular, the model for a storage device takes the form:
Response_time = $c(req\_size, req\_rate, rw\_ratio, random/sequential, cache\_hit\_rate)$
Function c is inherently non-linear, but can be approximated as piecewise linear with a few regions. We obtain this representation using SVM, as shown in Figure 4. Another source of error is the effect of multiple workloads sending interleaved requests to the same component. We approximate this nontrivial computation by estimating the wait time for each individual stream as per a multi-class queueing model [15]; more precise solutions [6] incorporate different workload characteristics. The effects of caching at multiple levels (e.g., hosts, virtualization engines, disk array controllers, disks) also introduce additional errors.

We took the liberty of bootstrapping component models by running off-line calibration tests against the component in question: a single, unchanging, synthetic I/O stream at a time, as part of a coarse traversal of c's parameter space.

3.2 Workload models

Representation and creation of workload models has been an active area of research [7]. In CHAMELEON, workload models predict the load on each component as a function of the request rate that each workload injects into the system. For example, to predict the rate of requests at component i originated by workload j:
Component_load$_{i,j}$ = $w_{i,j}(workload\_request\_rate_j)$
In real scenarios, function $w_{i,j}$ changes continuously as workload j changes or other workloads change their access patterns (e.g., a workload with good temporal locality will push other workloads off the cache). To account for these effects, we represent function $w_{i,j}$ as a *moving average* [24] that gets recomputed by SVM every n sampling periods.

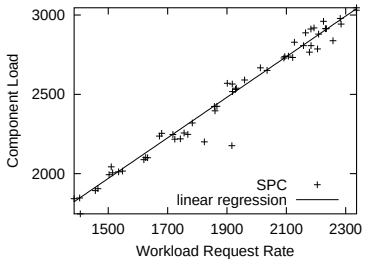


Figure 5. Workload model for SPC.

Figure 5 shows the workload models for the SPC web-search trace [10] running on a 24-drive RAID 1 LUN defined on an IBM FAStT 900 storage controller. From the graph, a workload request rate of 1500 iops in SPC translates to 2000 iops at the controller.

In practical systems, reliable workload data can only be gathered from production runs. We therefore bootstrap workload models by collecting performance observations; CHAMELEON resorts to throttling heuristics in the meantime, until workload models become accurate enough.

3.3 Action models

In general, action models predict the effect of corrective actions on workload requirements. The throttling action model computes each workload's average request rate as a function of the token issue rate, i.e.
Workload_request_rate = $a(token\_issue\_rate)$
Real workloads exhibit significant variations in their I/O request rates due to burstiness and to ON/OFF behaviors [6]. We model a as a linear function: $a(token\_issue\_rate) = \theta \times token\_issue\_rate$ where $\theta = 1$ initially for bootstrapping. This simple model assumes that the components in the workload's invocation path are not saturated.

To handle bursty workloads more realistically, we could have θ be a function of the request rates observed in the latest n sampling periods. This would maintain the Probability Distribution Function (PDF) of the request rate for each workload, and compute θ as a moving average of a given percentile.

Function a will, in general, also deviate from our linear model because of performance-aware applications (that modify their access patterns depending on the I/O performance they experience) and of higher-level dependencies between applications that magnify the impact of throttling.

4 Reasoning engine

The reasoning engine computes the rate at which each workload stream should be allowed to issue I/Os to the storage system. It is implemented as a constraint solver (using piecewise-linear programming [1]) that analyzes all possible combinations of workload token rates and selects the one that optimizes an administrator-defined objective function, e.g., "minimize the number of workloads violating their SLA".

It should be noted that the reasoning engine is not just invoked upon an SLA violation to decide throttle values, but also periodically to unthrottle the workloads if the load on the system is reduced.

4.1 Intuition

The reasoning engine relies on the component, workload, and action models as oracles on which to base its decision-making. Figure 6 illustrates a simplified version of how the constraint solver builds a candidate solution: 1) for each component used by the *underperforming* workload (i.e., the one not meeting its SLA), use the component's model to determine the change in request rate at the component required to achieve the needed decrease in component latency; 2) query the model for each workload using that components, to determine which change in the workload's I/O injection rate is needed to relieve the component's load; 3) using the action model, determine the change in the token issue rate needed for the sought change in injection rate; 4) record the value of the objective function for the candidate solution. Then repeat recursively for all combinations of component, victim workload, and token issue rates. The reasoning engine is actually more general: it considers *all* solutions, including the ones in which the desired effect is achieved by the combined results of throttling more than one workload.

4.2 Formalization in Chameleon

To formalize the problem for constraint solving, we need to formulate the task of deciding throttle values in terms of variables, objective function, and constraints.

Variables

One per workload, representing its token issue rate:$t_1, t_2, \ldots$

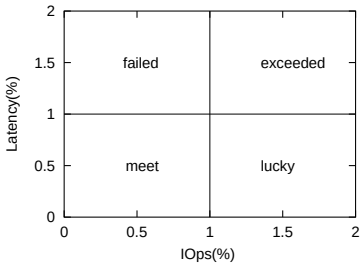


Figure 7. Workload classification. Region limits correspond to the 100% of the SLA values.

Objective function

Workloads are pigeonholed into one of the four regions (Figure 7) according to their current request rate, latency, and SLA goals: `meet`, `failed`, `lucky`, and `exceeded`. Region names are self-explanatory—`lucky` denotes workloads that are getting a higher throughput while meeting the latency goal, and `exceeded` denotes higher throughput while violating the latency goal.

Many objective functions can be accommodated by the current CHAMELEON prototype (e.g., all linear functions); moreover, it is possible to switch them on the fly. For our experiments, we used

$$Minimize \sum_{i \notin \texttt{failed}} \left| P_{quadrant_i} P_{Wi} \frac{SLA_{Wi} - a_i(t_i)}{SLA_{W_i}} \right| \quad (1)$$

where P_{Wi} are the *workload priorities*, $P_{quadrant_i}$ are the *quadrant priorities* (i.e., the probability that workloads in each region will be selected as throttling candidates), and $a_i(t_i)$ represents the action model for Wi. Table 1 provides some insight into this particular choice.

Constraints

Constraints are represented as inequalities: the latency of a workload should be less than or equal to the value specified in the SLA. More precisely, we are only interested in solutions that satisfy $latency_{Wi} \leq SLA_{Wi}$ for all workloads W_i running in the system.

The value of $latency_{Wi}$ is estimated using the following chain of parameters: t $\Rightarrow$ application_request_rate $\Rightarrow$ component_request_rate $\Rightarrow$ component_latency. Equivalently, $latency_{Wi} = c(w(a(t)))$.

For example, with only a single workload W_1 running in the system with its I/O requests are being served by a storage controller followed by physical disks is represented as the following constraint:

$$c_{controller}(w_{1,controller}(a_1(t))) + c_{disks}(w_{1,disks}(a_1(t))) \leq SLA_1 \quad (2)$$

In general, multiple workloads will share the components. As such, a more realistic example is:

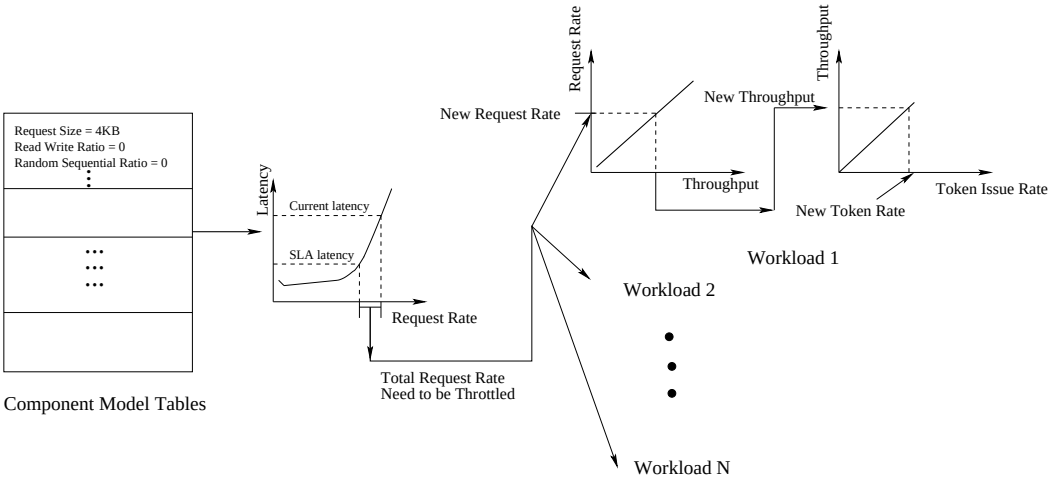


Figure 6. Overview of constrained optimization.

Intuition	How it is captured
The lower a workload's priority, the higher its probability of being throttled	The solver minimizes the objective function; violating the SLA of a higher priority workload will reflect as a higher value for $P_{Wi}\frac{SLA_{Wi}-a_i(t_i)}{SLA_{W_i}}$
Workloads in the `lucky` or `exceeded` region have a higher probability of being throttled	This is ensured by the $P_{quadrant_i}$ variable in the objective function; the `lucky` and `exceeded` have a higher value compared to the other regions (for example $P_{meet} = 1, P_{exceed} = 8, P_{lucky} = 32$). It is also possible to define $P_{quadrant_i}$ as a function.
Operating workloads closer to the SLA boundary	This is ensured by the difference of the current throughtput and SLA-value in the objective function; it is possible to assign a value function for workloads operating beyond their SLA using a bimodal objective function (currently that value is zero).

Table 1. Internals of the objective function.

$$c_{controller}(total\_req_{controller}) + c_{disks}(total\_req_{disks}) \leq SLA_{W1} \quad (3)$$

$$total\_req_{controller} = w_{1,controller}(a_1(t_1)) + w_{5,controller}(a_5(t_5)) \quad (4)$$

where w_1, w_5 are the workloads sharing the storage controller.

4.3 Workload unthrottling

CHAMELEON invokes the reasoning engine periodically, to re-assess token issue rates; if the load on the system has decreased since the last invocation, some workloads will be unthrottled.

In CHAMELEON, the goal of unthrottling is to re-distribute the unused bandwidth of the storage system based on the priority values and the average I/O rates. If a workload is consistently wasting tokens issued for it (because it has less significant needs), the additional tokens will be considered for re-distribution; on the other hand, if the workload is using all its tokens, they won't be taken away from it, no matter how low its priority is. Unthrottling decisions are constructed using the same objective function, but with additional "lower-bound" constraints such as not allowing each I/O rate to become lower than its current average value.

4.4 Confidence value of the reasoning engine

The confidence value of the reasoning engine is based on the accuracy of the models. Inaccuracies can stem from errors due to curve-fitting, and also from trying to use the models outside of the region(s) where they were trained (residuals).

There are multiple statistical formulas to represent the confidence values [15] . CHAMELEON uses the following formula to capture both the errors due to regression and the residuals.

$$S_p = S\sqrt{1 + \frac{1}{n} + \frac{(x_p - \overline{x})^2}{\sum x^2 - n\overline{x}^2}} \quad (5)$$

where S is the standard error, n is the number of points used for regression, and $\overline{x}$ is the mean values of the predictor variables used for regression. (S_p) represents the standard deviation of the predicted value in predicting the value of y_p using input variable x_p. In CHAMELEON, the confidence value (CV) of a model is represented as the inverse of its S_p.

The reasoning engine uses the component, workload, and action models to make throttling decisions; the output values are derived using a composition of models such that the output of one is an input for the other i.e. $c(w(a(t)))$. As such, the confidence value of the reasoning engine is approximated as $CV_{component} \times CV_{workload} \times CV_{action}$.

5 Designer-defined Policies

The system designer defines heuristics as a coarse-grained control mechanism, for deciding the workloads to be throttled; this is required in scenarios where the predictions of the models cannot be relied upon (either during bootstrapping or after significant system changes such as hardware failures). For example, "a component is saturated if its utilization is greater than 85%", or "start throttling workloads in the `lucky` region", or "if the workload-priority variance is less than 10%, uniformly throttle all workloads sharing the component". These heuristics can be expressed in a variety of ways such as Event-Condition-Action (ECA) rules or hard-wired code.

Coming up with useful throttling heuristics is a highly complex problem [26], especially if they are to consider a useful fraction of the solution space and to accommodate priorities; this is an error-prone process that is sure to result in a brittle set of policies (especially with respect the threshold values changes with the underlying physical configuration).

In CHAMELEON, the designer-defined heuristics are implemented as simple hard-wired code which is a modified version of the throttling algorithm used in Sleds [8]:

1 Determine the components being used by the underperforming workload and generate a $compList$.

2 For each component in the $compList$, determine the non-underperforming workloads using the component and add them to the $candidateList$.

3 Within the candidateList, order the workloads into groups based on their current operating quadrant: `lucky`, then `exceed`, then `meet`. Within each group, order the workloads based on their priority values.

4 Traverse the $candidateList$ and throttle each workload, either uniformly or proportionally to its priority (the higher the priority, the less significant the throttling).

6 Informed Feedback

Figure 8 shows the working of the feedback module. The feedback module incrementally throttles workloads based on the decisions of either the reasoning engine or the system-designer heuristics (when the confidence value of the reasoning engine is below a certain threshold value, say 30%). The step size for the incremental throttling is proportional to the confidence value of the constraint solver or is a small constant value while using designer heuristics.

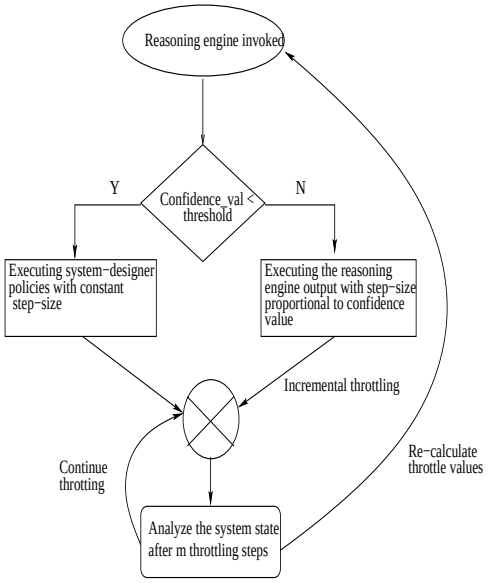


Figure 8. Working of the feedback module

After every m incremental throttling steps, the feedback module analyzes the state of the system. If any of the following conditions is true, it re-invokes the reasoning engine (otherwise it continues applying the same throttling decisions in incremental steps):

- Latency increases for the underperforming workload (i.e., it moves away from the meet region).
- A non-underperforming workload moves from the meet or exceed region to the lucky region.
- Any workload undergoes a 2X or greater variation in the request-rate or any other access characteristic (compared to the values at the beginning of throttling).
- There is a 2X or greater difference between predicted and observed response times for a component.

After the reasoning engine is invoked consecutively for l times and has a lower confidence value in each invocation, the feedback module discards the throttle values and switches to the designer heuristics. There can be multiple reasons for the above-mentioned conditions to be true: component failures, application-level correlations between workloads, unpredictable variations in the component behavior.

7 Experimental Evaluation

The experimental setup consists of a host machine generating multiple workload streams that are being served by a storage infrastructure. The host is an IBM x-series 440 server (2.4GHz 4-way with 4GB memory running Redhat Server 2.1 kernel); the back-end storage is a 24 drive RAID

1 LUN created on a IBM FAStT 900 storage controller (dual HBA) with 512MB of on-board NVRAM. The host and the storage controller are connected using a 2Gbps FibreChannel (FC) link.

The IOs generation is controlled by a token-based leaky bucket protocol i.e. a token is required to issue an IO request to the storage system. The number of tokens issued to each workload stream is controlled by CHAMELEON that is running on the host machine as a separate process. The RAID 1 logical volume is mapped at the host as a `raw` device; as such there is no IO caching at the host-level.

The key capability of CHAMELEON is to regulate resource load so that SLAs are achieved. The experimental results use numerous combinations of synthetic and real-world request streams to evaluate the effectiveness of CHAMELEON; synthetic workloads are easier to handle compared to their real-world counterparts that exhibit a bursty and highly variable access characteristics. In addition to the effectiveness, we evaluate the computation complexity of the constraint solver as a function of the number of workloads.

7.1 Using synthetic workloads

The synthetic workload specifications used in this section were derived from Minerva's performance study [2]. Since the workloads are relatively static and controlled, the models for action and workload have a small error rate. In this experiment, the component model has $r = 0.72$, the workload and action models with $r > 0.9$. (r is the correlation coefficient [15] and represents the accuracy of the models; the closer r is to 1, the more accurate the models are.)

These tests serve two objectives. First, they evaluate the correctness of the decisions made by the constraint solver i.e. the throttling decisions should take into account the workload priorities, current operating point compared to the SLA, and the percentage of load on the components generated by the workload. Second, the tests depict the effect of model errors on the output values of the constraint solver and how incremental feedback helps the system converge to an optimal state.

Workload	Request Size (KB)	Read Write Ratio	Sequential Random Ratio	Footprint Size (GB)
W_1	27.6	0.98	0.577	30
W_2	2	0.66	0.01	60
W_3	14.8	0.641	0.021	50
W_4	20	0.642	0.026	60

Table 2. Synthetic workload streams

Test 1: Workloads with equal priorities

Figure 9 shows the latency and throughput values for the four workloads W_1, W_2, W_3, and W_4 running on the system. Initially the system runs in uncontrolled phase (till t=60 sec); in this phase workload W_1 is violating its SLA while other workloads such as W_3 and W_4 are above their SLA. CHAMELEON calculates the new token issue rate for each workload and executes them incrementally with a step-size proportional to the confidence value (step-size in this case is 12%). The settling time between each incremental step is 60 sec. At time t = 300 sec, the system converges to a state where no workload is violating its SLA.

The final state of the system can be represented in the SLA quadrant(figure 10), where the arrows represent the new operating point after throttling. Figure 10 compares the system state that would have been achieved if the output throttle values [1] of the reasoning engine were directly executed; the throttle values cause over-throttling with workload W_1 operating much further in the meet region and no workload in the exceed region. Over-throttling is caused by model errors (in this case component model) where the predicted latency used by the reasoning engine was higher compared to the actual observed value.

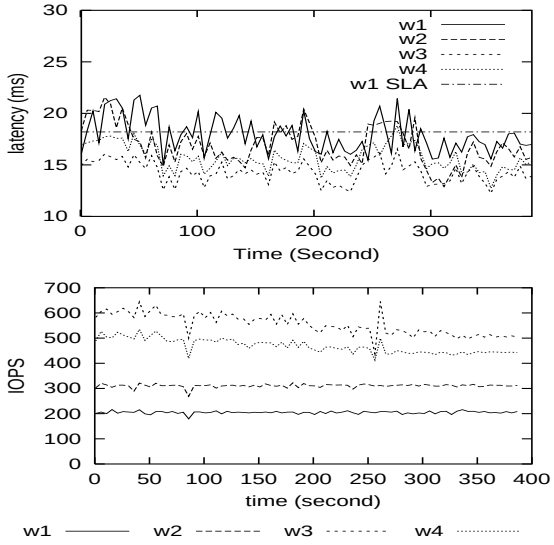
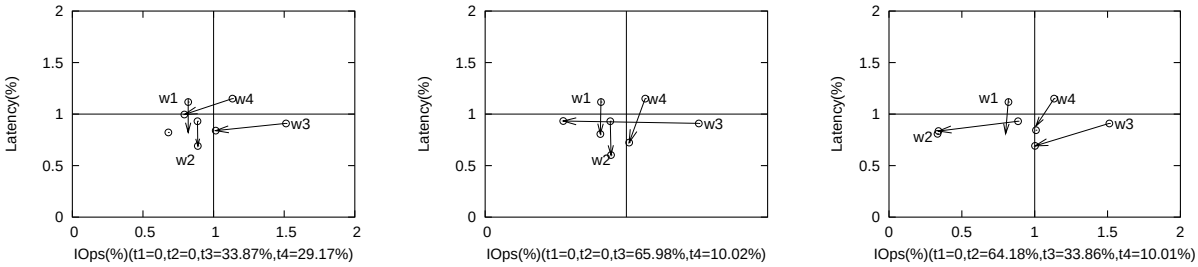


Figure 9. Throughput and latency values for synthetic workloads with equal priorities. Throttling of workloads starts at t=60 sec.

Test2: Effect of workload and quadrant priorities

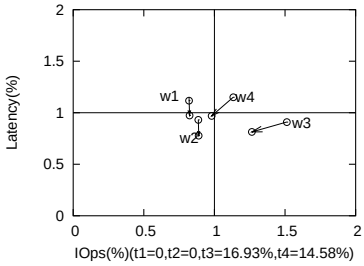
Figure 11 compares the direct output of the constraint solver with priority values for the workloads ($W_1 = 8, W_2 = 16, W_3 = 2, W_4 = 8$) and the SLA quadrants. Compared to the no priority case, workload W_3 is throttled more aggressively. This is because the constraint solver internally uses a greedy algorithm, throttling the lowest priority workload before moving to the higher ones.

[1] The throttle values shown at the bottom of SLA quadrant figures represents the percentage decrease in the token issue rate

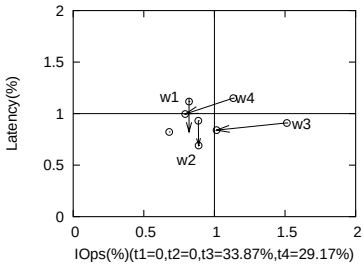


Equal priority *Workload priorities* *Quadrant priorities*

Figure 11. Effect of priority values on the output of the constraint solver.



Final system-state with incremental throttling



Final system-state if the output of the reasoning engine is directly executed

Figure 10. Effect of model errors on output of the constraint solver.

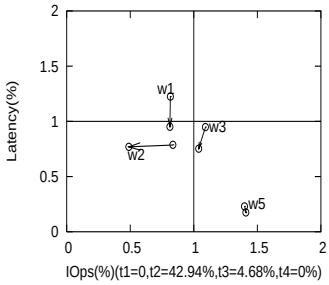


Figure 12. Sanity test for the reasoning engine (workload W_5 operating from controller cache

Test 3: Usage of the component by the workload

This test is a sanity check with workload W_5 operating primarily from the controller cache (not using the disk bandwidth). To solve the SLA violation for workload W_1, the reasoning engine shouldn't select W_5 for throttling even if W_5 has the lowest priority. The throttling decision made by CHAMELEON is as shown in figure 12 which indicates the reasoning engine selects W_2 and W_3.

7.2 Using real-world workload trace replay

In these experiments, we replay the real-world workload for SPC web-search [10] and HP's Cello96 [2] traces. Cello96 was collected from a departmental fileserver over the period from 9 September to 29 November 1996. Both these traces are block-level with timestamps recorded for each IO request. The trace duration for SPC is around 6 hours while that for Cello96 is 1 day long. To generate a reasonable IO load for the storage infrastructure, the SPC traces were replayed 40 times faster and the cello traces were replayed 10 times faster.

For these tests, in addition to the SPC and Cello96 traces, a phased synthetic workload was used; this workload is assigned the highest priority. In an uncontrolled case i.e. without throttling, with three workloads running on the system, one or more of them violate their SLA. Figure 13 shows the throughput and latency values for uncontrolled case. The horizontal line in the delay figures represents the SLAs for each workload. As we can see from the figure, when the synthetic workload is turned on, the SLA on latency were violated.

The aim of the tests is to evaluate the following:

- The throttling decisions made by CHAMELEON for converging the workloads towards their SLA.
- The reactiveness of the system with throttling and periodic unthrottling of workloads (under reduced system load).
- The handling of unpredictable variations in the system that cause errors in the model predictions, forcing CHAMELEON to use the sub-optimal but conservative designer-defined policies.

For these experiments, the models were reasonably accurate (component $r = 0.68$, workload $r = 0.7$, and ac-

[2] http://tesla.hpl.hp.com/public software

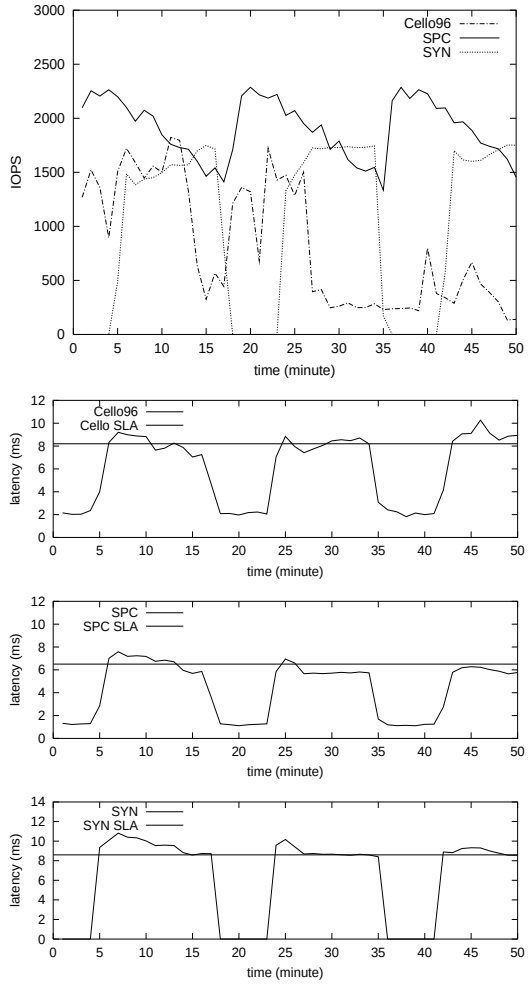


Figure 13. Uncontrolled throughput and latency values for real-world workload traces

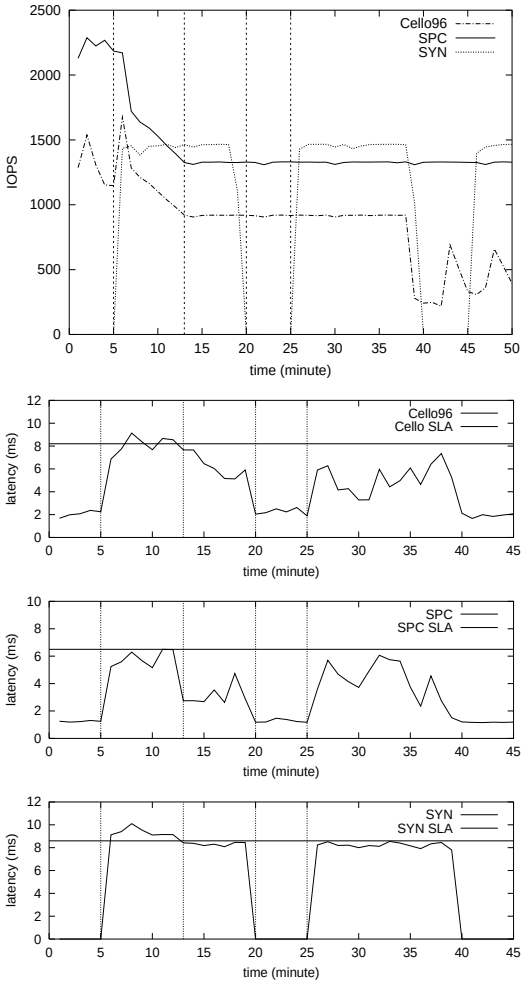


Figure 14. Throughput and latency values for real-world workload traces with throttling (without periodic unthrottling)

tion $r = 0.6$).In addition, the SLAs for each workload are: Cello96 1000 IOPS with 8.2ms latency, SPC 1500 IOPS with 6.5 ms latency and 1600 IOPS with 8.6ms latency for the synthetic workload unless otherwise specified.

Case 1: Solving SLA violation using throttling

The behavior of the system is shown in figure 14. To explain the working of CHAMELEON, we divide the time-series into phases described as follows:

Phase 0 (t=0 to t=5 min): Only the SPC and Cello96 traces are running on the system; the latency values of both these workloads is significantly below the SLA.

Phase 1 (t= 5 min to t= 13 min): The phased synthetic workload is introduced in the system. This causes an SLA violation for the Cello96 and synthetic traces. CHAMELEON triggers the throttling of the SPC and Cello96 workloads (Cello96 is also throttled because it is operating in the exceeded region, means it is sending more than it should. Therefore, it is throttled even its SLA latency goal is not met). The system uses a feedback approach to move along the direction of the output of the constraint solver. In this experiment, the feedback system starts from 30% of the throttling value and uses step size is 8%. (30% and 8% are decided according to the confidence value of the models). It took the system 6 minutes to meet the SLA goal and the feedback stops.

Phase 2 (t=13 min to t= 20 min): The system stabilizes after the throttling and all workloads can meet their SLAs.

Phase 3 (t=20 min to t= 25 min): The synthetic workload enters the OFF phase. During this time, the load on the system is reduced, but the throughput of Cello96 and

SPC remains the same.

Phase 4 (beyond t= 25 min): The system is stable, with all the workload meeting their SLA. A side-point is that at around t=39 min, the throughput of Cello96 decreases further; this is because of the inherent trace characteristics.

Figure 14 shows the effectiveness of the throttling: all workloads can meet their SLA after throttling. However, because the lack of unthrottling scheme, throttled workloads have no means to increase their throughput even when tokens are released by other workloads. Therefore, the system is underutilized.

Case 2: Side-by-side Throttling and Unthrottling of workloads

Figure 15 shows throttling of workloads with periodic unthrottling (every 60 sec) during reduced system loads. In comparison to Figure 14, there are four interesting observations:

- First, the behavior of system during the off phase of the synthetic workload (t=17 min to t=27 min). In this duration, the system load is reduced that triggers unthrottling of the SPC and Cello96 workloads. Unthrottling is based on workload priorities and the average IO demand of the individual workload streams. The SPC and Cello96 grab more tokens and are sending out accumulated requests due to limited tokens when the synthetic workload is on.

- Second, the settling time required to throttle the SPC and Cello96 workloads, whenever the synthetic workload gets into the on phase (t= 27 min to t=37 min, t=47 min to 50 min). Compared to the throttling-only scenario, the throughput and latency variations of the system are higher, taking a longer time to stabilize.

- Third, the constraint solver made a different throttling decision from Case 1: Cello96 was not throttled. This is because when the reasoning engine is triggered, the Cello96 was sending less than its SLA IOPS and was not meeting its SLA latency goals (t=9 min). As a result, both Cello96 and the synthetic workload were operating in the `failed` region, therefore, the reasoning engine will not throttle Cello96 as in Case 1 and only SPC was throttled.

- Fourth, between t=10 min to t=13 minutes, reasoning engine was triggered twice. This is because all workloads met their SLA goals after the first throttling (t=11 min) and the feedback stops. However, at t=12 min, SLAs for Cello96 and the synthetic workloads were violated again, a second call on reasoning engine was triggered and SPC was throttled again. After t=13 min, the system was stabilized.

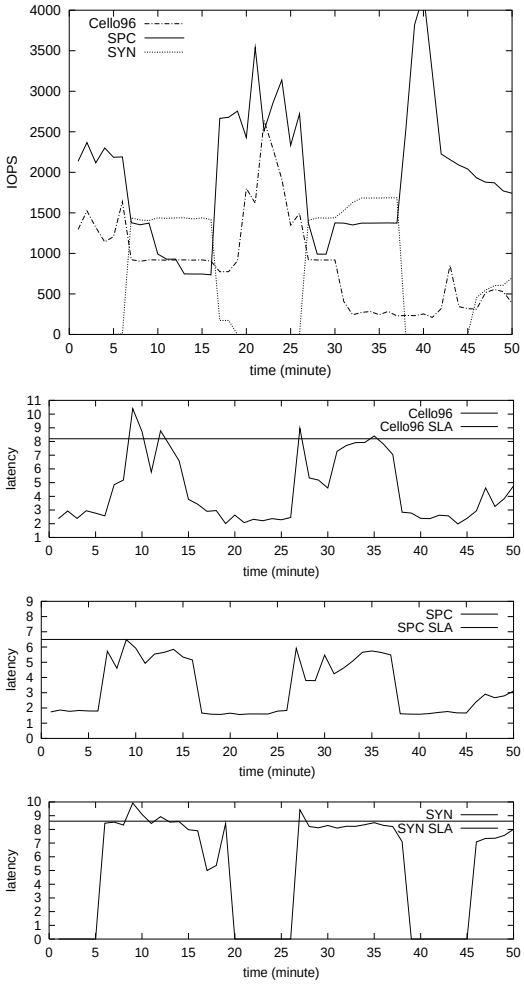


Figure 15. Throughput and latency values for real-world workload traces with throttling and periodic unthrottling

Case 3: Handling changes in the confidence value of the models at run-time

This test demonstrates how CHAMELEON caters to change in confidence value of the models at run-time; this change can be due to unpredictable system variations (hardware failures) or un-modeled properties of the system (such as changes in the workload access characteristics that change the workload models). It should be noted that refining the models to reflect the changes will not be instantaneous; in the meantime, CHAMELEON should have the ability to detect a fall in the confidence value and switch to a conservative management mode (using designer-defined policies or generate a log message for a human administrator).

Figure 16 show the reaction of the system when the access characteristics of the SPC and Cello96 workloads are synthetically changed such that the cache hit rate of Cello96 increases significantly (in reality, a similar scenario

arise due to changes in the cache allocation to individual workload streams sharing the controller) and the SPC is doing more random access (sequential random ratio increases from 0.11 to 0.5). In the future, we plan to run experiments with hardware failures induced on the RAID 1 logical volume.

The SLAs used for this test are: Cello96 has a SLA with 1000 IOPS with 7ms latency, SPC is 2000 IOPS with 8.8ms latency and the synthetic workloads has a SLA with 1500 IOPS and 9ms latency.

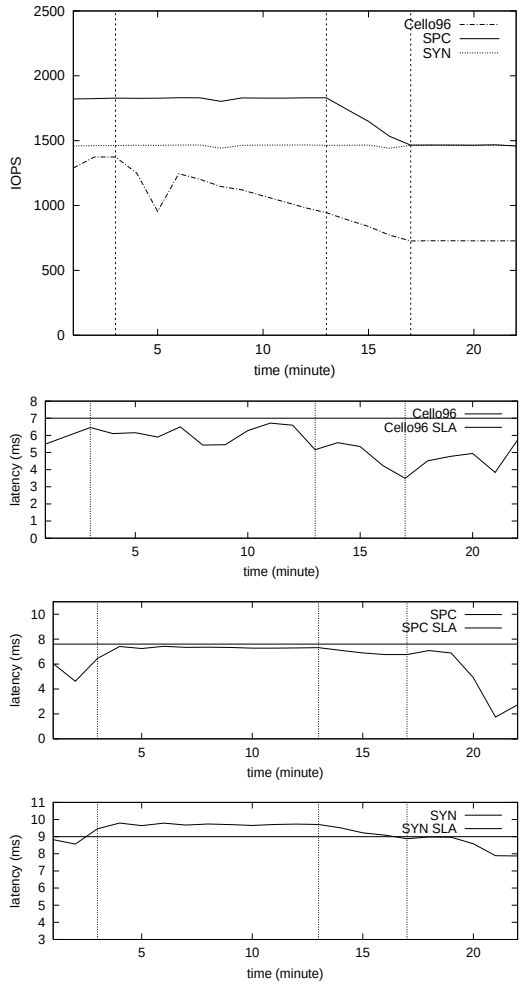


Figure 16. Handling a change in the confidence value of the models at run-time

Phase 0 (at t= 3 mins): The synthetic workload violates its latency SLA. In response, CHAMELEON decides to throttle the Cello96 workload (using the original workload model). The output of the reasoning engine as a confidence value of 65%

Phase 1 (t= 3 min to t = 13 min): The feedback module continues to throttle for 3 consecutive increments; since the latency of the synthetic workload does not change, it re-invokes the reasoning engine. The output of the reasoning engine is similar to the previous invocation (since the models haven't changed), but its confidence value is lower (because of the higher differences between predicted and observed model values). This repeats for consecutive invocations of the reasoning engine after which the feedback module switches to use the designer-defined policies.

Phase 2 (t= 13 min to t=17 min): A simple designer policy the CHAMELEON uses is to throttle all the non-violating workloads uniformly (*uniform pruning*). Both SPC and Cello96 are throttled in small steps (5% of their SLA IOPS) till the latency SLA of the synthetic workload is satisfied.

Phase 3 (beyond t= 17min): All workloads are meeting their SLA goals and the system is stabilized.

7.3 Computational complexity of the reasoning engine

The current implementation of CHAMELEON uses a piece-wise linear programming approach for constraint solving. The computational complexity of the constraint solver is a function of the number of variables involved. Figure 17 shows the amount of time CHAMELEON takes to generate the answer. This experiment was run on a P4 2.8 Ghz machine with 512MB memory.

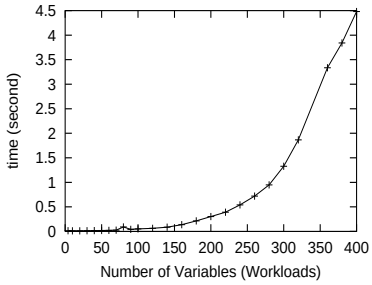


Figure 17. Computational complexity of the reasoning engine

7.4 Discussion of the experimental results

We were pleased that in our experiments, CHAMELEON was able to automatically execute throttling procedures that moved the system to its optimal state as defined by the value function. In all cases the right workloads were throttled and the amount of the throttling was the minimum needed to meet the targets. With our current system and for the workload perturbations we imposed, we saw reaction times between 3 and 12 minutes. While this is not instantaneous, it is almost certainly quicker than a human could react (notice the problem, decide what to do, execute) and also almost certainly more precise in execution (got to exactly the

right place). Unthrottling was similarly successful, releasing constraints when load on the high priority workloads decreased; it introduces an additional delta to the settling-time which is dependent on how proactively the token are issued and recovered. Again, we believe our reaction times were better than human-capable at 14 minutes. Thus, we are achieving the goals of allowing full hardware utilization when all workloads are meeting responsiveness requirements and we were able to appropriately restrain the lower priority or greedy ones when system limits were reached.

8 Related work

In *rule-based systems*, system administrators encode policies as sets of event-condition-action rules [28, 14] that fire when some precondition (typically, one or more system metrics going beyond a predetermined threshold) is satisfied. Most current commercial tools for automatic resource allocation (e.g., BMC Patrol [5]) belong to this category. Rules are a clumsy, error-prone programming language; they front-load all the complexity into the work of creating them, in exchange for simplicity of execution at run time. Administrators are expected to create rules that account for all relevant system states, to know which corrective action to take in each case, to specify useful values for all the thresholds that determine when rules will fire, and to make sure that the intended rule will fire if preconditions overlap. Since all this work has to be done in advance and with minimum quantitative information about the system, and simple policy changes may translate into modifications to a large number of rules, this approach is unlikely to significantly improve manageability and accuracy of response. Rule-based systems can only provide a coarse-grained optimization, as good as the human who wrote the rules. In contrast, CHAMELEON relies on constraint-solving algorithms that explore the entire search space of throttle values for each workload. Instead of relying on hardwired thresholds, CHAMELEON uses its dynamic models to make optimization decisions that admit iterative refinement. A variation [29], based on case-based reasoning, relies on iterative refinement to derive rules from a *tabula rasa* initial knowledge base. This approach does not scale well to real systems, because of the exponential size of the search space that is explored in an unstructured way.

Feedback-based approaches use a narrow window of the most recent performance samples to make allocation decisions based on the difference between the current and desired system states. They are not well-suited for decision-making with multiple variables [23], and can keep thrashing between local optima. from Façade [19] controls the queue length at a single storage device. If there is a QoS violation, Façade decreases the target queue length, doing the equivalent of throttling the combination of all workloads down to the current request rate of the device; unlike CHAMELEON, it does not make complex decisions about which subset of the workloads should be left alone. Under overload conditions Façade will reduce its target queue length all the way down to 1, thus disallowing internal optimizations in the storage device and getting poor performance; CHAMELEON will provide differentiated service by throttling the low-priority workloads. Triage [18] keeps track of which performance band the system is operating in; it shares Façade's lack of selectivity, as a single QoS violation may bring the whole system down to a lower band (which is equivalent to throttling every workload). Sleds [8] can selectively throttle just the workloads supposedly responsible for the QoS violations, and has a decentralized architecture that scales better than Façade's. However, the policies for deciding which workload to throttle are hard-wired and will not adapt to changing conditions. Hippodrome [4] fine-tunes the initial data placement iteratively. Given the high cost of each data migration, it can take a long time to converge and may get stuck in local minima as it relies on a variation of hill-climbing.

Scheduling-based approaches establish relative priorities between workloads and individual I/Os. Jin et.al. [16] compare different scheduling algorithms for performance isolation and resource-usage efficiency; their experimental results show that scheduling is effective but cannot ensure tight bounds on the SLA constraints (which is especially required for high-priority workloads). Stonehenge [13] uses a learning-based bandwidth allocation mechanism to map SLAs to virtual device shares dynamically; although it allows more general SLAs than CHAMELEON, it can only arbitrate accesses to the storage device, not to any other bottleneck component in the system. In general, scheduling approaches are designed to *optimize for the common case*, and may not be effective in handling exception scenarios such as hardware failures.

Model-based approaches depend on accurate models of the storage system in order to make decisions. Minerva [2] assumes that models are given—but system administrators very rarely have that level of information about the devices they use. Polus [26] proposes to build those models on the fly; CHAMELEON is an intermediate step towards the full Polus vision. The main challenge in this category is to acquire robust, accurate models—far from trivial for practical systems.

9 Conclusions

An ideal solution for resource arbitration in shared storage systems would adapt to changing workloads, client requirements and system conditions. It would also relieve system administrators from the burden of having to specify when to step in and take corrective action, and what actions to take—thus allowing them to concentrate on specifying the global objectives that maximize the storage utility's business benefit, and having the system take care of the details. No existing solution satisfies these criteria; prior approaches are either inflexible, or require administrators to supply up-front knowledge that is not available to them.

Our approach to identifying which client workloads should be throttled is based on constrained optimization. Constraints are derived from the running system, by monitoring its delivered performance as a function

of the demands placed on it during normal operation. CHAMELEON's approach to model building results in a solution that requires no prior knowledge about the quantitative characteristics of workloads and devices—and that can make good decisions even in the presence of realistic scenarios, like those involving workloads with relative priorities. The objective function being optimized can be defined, and changed, by the administrator as a function of organizational goals. Given that the actions prescribed by our reasoning engine are only as good as the quality of the models used to compute them, CHAMELEON will switch to a conservative decision-making process if insufficient knowledge is available.

We replayed traces from production environments on a real storage system, and found that CHAMELEON makes very accurate decisions for the workloads examined. CHAMELEON always made the optimal throttling decisions, given the available knowledge. The times to react to and solve performance problems were in the 3-14 min. range, which is quite encouraging.

As areas for future work, first, we can improve the quality of model representations and the processes used to build them: component models could account for phased workloads and accurate interleaving, and workload models could incorporate additional workload characteristics such as temporal locality (and even incorporate some degree of prediction using techniques related to ARIMA [24]). Second, the reasoning engine could be based on a more general type of optimization, e.g., use non-linear programming for the constraint solver as supported by OPT++ [20]. Finally, we could account for a variety of additional real-world aspects: preventing over-fitting in models, avoiding oscillations or ping-pong effects, or even generating explanations for the administrator for the throttling decisions made by CHAMELEON.

References

[1] GLPK (GNU linear programming kit). http://www.gnu.org/software/glpk/glpk.html.

[2] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[3] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, HP Laboratories, July 2001.

[4] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around stroage administration. pages 175–188, January 2002.

[5] BMC Software. *Patrol for Storage Networking*, 2004.

[6] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *Proceedings of the fi rst international workshop on Software and performance*, pages 199–207. ACM Press, 1998.

[7] Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proc. IEEE*, 81(8):1136–1150, 1993.

[8] D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems*, October 2003.

[9] Guillermo A. Alvarez Chenyang Lu and John Wilkes. Aqueduct: online data migration with performance guarantees. pages 175–188, January 2002.

[10] Storage Performance Council. Spc i/o traces. http://www.storageperformance.org/.

[11] FalconStor Software. Ipstor: build an end-to-end IP-based network storage infrastructure. White paper. http://www.falconstor.com, 2001.

[12] Gartner Group. Total Cost of Storage Ownership—A User-oriented Approach. Research note, Gartner Group, 2000.

[13] Lan Huang, Gang Peng, and Tzi cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.

[14] IETF Policy Framework Working Group. IETF Policy Charter. http://www.ietf.org/html.charters/policy-charter.html.

[15] Raj Jain. *The Art of Computer System Performance Analysis*. Wiley, 1991. JAI r 91:1 1.Ex.

[16] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, 2004.

[17] T. Joachims. *Making large-scale SVM learning practical*. MIT Press, Cambridge, USA, 1998.

[18] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proc. of the 12th. Int'l Workshop on Quality of Service*, June 2004.

[19] C. Lumb, A. Merchant, and G.A. Alvarez. Faç ade: virtual storage devices with performance guarantees. In *Proc. 2nd Conf. on File and Storage Technologies (FAST)*, pages 131–144, April 2003.

[20] J.C. Meza. Opt++:an object-oriented class library for nonlinear optimization. Technical report.

[21] H. Newman, M. Ellisman, and J. Orcutt. Data-intensive e-science frontier research. *CACM*, 46(11):68–77, 2003.

[22] Storage Networking Industry Association. SMI Specifi cation version 1.0. http://www.snia.org, 2003.

[23] David Gerand Sullivan. Using probabilistic reasoning to automate software tuning. September 2003.

[24] Nancy Tran and Daniel A. Reed. ARIMA time series modeling and forecasting for adaptive i/o prefetching. In *Proceedings of the 15th international conference on Supercomputing*, pages 473–485. ACM Press, 2001.

[25] J. Turner. New directions in communications. *IEEE Communications*, 24(10):8–15, October 1986.

[26] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease. Polus: Growing storage QoS management beyond a 4-year old kid. In *FAST04*, March 2004.

[27] M. Uysal, G. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems*, pages 183–192, August 2001.

[28] D. Verma. Simplifying network administration using policy based management. (2), March 2002.

[29] D. Verma and S. Calo. Goal Oriented Policy Determination. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Sys.*, pages 1–6. ACM, June 2003.

[30] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. *SIGMETRICS Perform. Eval. Rev.*, 32(1):412–413, 2004.