DecisionQoS: an adaptive, self-evolving QoS arbitration module for storage systems

Sandeep Uttamchandani Guillermo A. Alvarez Gul Agha*
{sandeepu,alvarezg}@us.ibm.com
IBM Almaden Research Center, 650 Harry Rd., San José, California 95120, USA

Abstract

As a consequence of the current trend towards consolidating computing, storage and networking infrastructures into large centralized data centers, applications compete for shared resources. Open enterprise systems are not designed to provide performance guarantees in the presence of sharing; unregulated competition is very likely to result in a free-for-all where some applications monopolize resources while others starve. Rule-based solutions to the resource arbitration problem suffer from excessive complexity, brittleness, and limitations in their expressive power. We present DECISIONQOS*, a novel approach for arbitrating resources among multiple competing clients while enforcing QoS guarantees.* DECISIONQOS *requires system administrators to provide a minimal, declarative amount of information about the system and the workloads running on it. That initial input is continuously refined and augmented at run time, by monitoring the system's performance and its reaction to resource allocation decisions. When faced with incomplete information, or with changes in the workload requirements or system capabilities,* DECISIONQOS *adapts to them by applying machine learning techniques; the resulting scheme is highly resilient to unforeseen events. Moreover, it overcomes significant shortcomings of pre-existing, rule-based policy management systems.*

1 Introduction

Data centers are becoming increasingly popular in enterprise environments, as resources are consolidated to reap the benefits of statistical sharing and lower management costs. Consolidated resources are mainly computing power in the form of CPU cycles, network bandwidth, and storage sub-systems. Within a data center, hundreds or thousands of hosts typically access a terabyte or more of data each, stored in large, shared storage servers interconnected by a storage area network (SAN) such as Fibre Channel. A variety of applications such as web servers, online transaction processing and decision support systems runs on enterprise data centers. Many of those applications depend on predictable performance from the storage system in order to accomplish their goals, e.g., acceptable interactive transactions may require average I/O latencies to be under 5 ms. In general, a Service Level Agreement (*SLA*) prescribes the minimum quality of service (*QoS*) that a client application will experience, provided that its demands on the system do not exceed given bounds. We concentrate on the performance that the storage system must guarantee to its clients; other QoS dimensions include reliability, performability, and manageability.

Guaranteeing SLAs is a difficult problem, as the overwhelming majority of off-the-shelf devices, operating systems, and protocols allocate resources on a best-effort basis. The problem becomes still more difficult in consolidated systems [8, 21]. Storage consolidation introduces additional coupling, when previously unrelated workloads compete for resources such as disk drive actuators, network links and endpoints, switch backplanes, controller processors, data caches, system buses, and SCSI interconnects. Due to largely unpredictable and platform-dependent scheduling policies, unregulated competition will result in some applications starving while others use more than their fair share of the system's resources. Solutions based on static provisioning typically result in low levels of system utilization, and the only way of guaranteeing a fair allocation of resources is to resort to physical or logical separation. Static approaches also cope poorly with unforeseen events such as workload variations, failures, and additions of capacity to the system.

For additional flexibility, resource consumers can declare their needs in advance to an arbitrator (a model closer to that of admission control plus support for resource compartmentalization), or specialized QoS Arbitration Modules (*QAMs*) can modify resource allocations on the fly

*Department of Computer Science, University of Illinois at Urbana-Champaign. E-mail: agha@cs.uiuc.edu

whenever they are found to be inadequate. A QAM is responsible for ensuring that, as long as enough aggregate resources are present in the system, all SLAs will be satisfied[1].

The existing approach for building a QAM is using the rule-based paradigm [23]. Rules are used to define how the resources are partitioned among client applications. In AI terminology, the rule-based approach for specifications are referred to as *pattern directed procedure invocation* [15, 16]. While the advantage of procedural rule-based schemes is that they allow the specification of direct interaction between facts and eliminate the need for wasteful run-time searching [14], the disadvantage is that writing procedures (i.e., rules) are like programs that are difficult to write, modify and error-prone [10]. In contrast, this paper introduces DECISIONQOS which is a declarative, i.e., non-procedural paradigm [13] for building QAMs, based on Polus [25]. In DECISIONQOS, the administrator is not required to specify how the resources should be partitioned in different system states. Instead, the administrator simply specifies facts and constraints as logical formulas which are easy to write and modify. These logical formulas are combined at run-time with prepackaged formalisms (referred to as *reasoning*), to decide resource partitioning among the client applications. Logic-based specifications were originally considered deficient in capturing heuristic knowledge, which led to MIT's "procedure-is-best" debate [29]. The debate ended in favor of logic-based approaches, after Kowalski's procedural interpretation of the behavior of a Horn-clause linear resolution proof finder [19].

In summary, DECISIONQOS's specifications are declarative; it does not require system administrators to encode policies as complex, brittle rule sequences. DECISIONQOS does not require accurate, detailed inputs [1] to make good decisions; it can take a potentially minimal amount of system and workload information, and then refine it at run time. The net effect is that of relieving users from the burden of making sensitive, error-prone decisions (e.g., setting decision thresholds), and achieving nimble responses to changes in the operating conditions. DECISIONQOS hides a significant amount of complexity that is not relevant to users; in so doing, it does not depend on human experts to tweak and maintain the rule sets.

We define our version of the QoS arbitration problem, and discuss the shortcomings of rule-based QAMs in Section 2. Section 3 introduces our main assumptions; Sections 4 and 5 present DECISIONQOS's architecture and internal operation, respectively. We put our work in context in Section 6 and draw some conclusions in Section 7.

[1]More stringent definitions are possible: QAMs could attempt to guarantee that resources will be fairly shared, or that they will be optimally utilized (e.g., load balancing). Such extensions are beyond the scope of this paper.

2 The resource arbitration problem

In our version of the problem, a QAM manages the assignment of available storage resources to host workloads. This mapping must ensure that no workload fails to meet its SLA (a *QoS violation*). The QAM is invoked each time a QoS violation is detected. Upon invocation, the QAM attempts to bring the system to a state where no SLAs are violated, by identifying workloads whose resource consumption should be *throttled*.

Choosing which workloads to throttle is a fairly complex task for many reasons. First, workload access patterns change constantly (e.g., as a result of burstiness). The amount of resources freed up by throttling a given workload to a given degree is a dynamic function. Second, a workload's behavior may be related, at the application level, to that of other workloads or even human users. For instance, throttling accesses to a database log will affect the transaction workloads. The QAM needs to consider these dependencies when they exist. Third, each workload uses a fixed set of physical components referred to as its *invocation path*. The QAM should make sure that the workloads being throttled share either invocation path elements or application-level dependencies with the workloads that are experiencing QoS violations—otherwise their performance would be independent of one another, so throttling them would not help remedy the problem. Fourth, failures occur at unpredictable times. Even if data remains accessible due to built-in redundancy (e.g., RAID) performance will typically suffer because of the decrease in the overall amount of available resources. QAMs need to adapt to these events within a fairly short time interval, reapportioning resources so that the system continues to satisfy the SLAs. Fifth, the QAM should be potentially able to throttle any subset of the workloads in the system (although doing so optimally is NP-hard); this results in an exponential number of possible decisions.

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 in order for the QoS violations to disappear. When faced with several choices for a given set of QoS violations, the QAM should minimize the side effects of its actions, i.e., have as little impact as possible on the workloads that are neither experiencing inadequate performance nor being throttled directly.

Many existing implementations of storage QAMs are

based on flavors of policy-based management [17, 23] where system behavior is described as a set of rules that are invoked when certain system conditions are met. Most rules are variations on the theme of Event-Condition-Action (*ECA*), with the semantics that the action will be executed if both a given type of event occurs (e.g., a violation of a given QoS metric) and a condition is satisfied. Let us consider writing a few example rules to define the behavior of a hypothetical QAM[2]. The set of rules can be divided into two categories:

Rules for selecting candidate workloads: Workloads are throttled in increments of $step\_size$.

Condition: If workload exceeds (1.6 SLA) $\wedge$ $inv\_path(w) \triangle inv\_path(w_{under\_provisioned})$
Action: Mark workload w as candidate and *step_size* = 15%

Condition: If workload is between (1.25–1.6 SLA) $\wedge$ $inv\_path(w) \triangle inv\_path(w_{under\_provisioned})$
Action: Mark workload w as candidate and *step_size* = 10%

Condition: If workload exceeds (1.15 SLA) $\wedge$ $inv\_path(w) \triangle inv\_path(w_{under\_provisioned})$
Action: Mark workload w as candidate and *step_size* = 3%

Rules for deciding which candidate workloads should be throttled: Relationships between workloads are represented as a set of *correlation probabilities* cp_w, that throttling workload w will indirectly throttle any other workload in the system. Let avg_cp, var_cp denote the average and variance of the correlation probability over all candidates.

Condition: If *num_candidates* $> 1 \wedge avg\_cp < 0.4 \wedge var\_cp < 0.1$
Action: Throttle all w with $cp_w \leq 0.8$ by *step_size*

Condition: If *num_candidates* $> 1 \wedge avg\_cp < 0.4 \wedge var\_cp > 0.3$
Action: For workloads with $cp < 0.2$, throttle 85% of excess demand; for workloads with cp between 0.2-0.6, throttle 45% of excess demand

Condition: If *num_candidates* $> 1 \wedge avg\_cp > 0.6 \wedge var\_cp > 0.3$
Action: Select w_min such that cp_{w_min} is minimum, throttle by *step_size*.

The example highlights the main limitations of rule-based approaches for building QAMs:

- *Complexity*: writing rules to express the QAM logic is non-trivial and requires a fair amount of expertise. Rules have built-in threshold values that are quite difficult to determine in a practical system—but the effectiveness of the rules depends on their accuracy to a significant extent. More importantly, while writing the rules, the system-builder has to (manually) account for all the possible states that the system can be in, and for all the possible steps that can be taken from those states. System administrators cannot cope with this level of complexity, resulting in error-prone policy setting. Since rules implicitly capture the reasoning details, it is difficult to understand and maintain the precise reasoning behind the creation of a set of rules.

- *Limitations on expressive power*: the rule-based paradigm is based on imperative specifications that trade off a relatively lightweight processing at run time by extensive reasoning required at rule creation time—when detailed information about system and workload may be hard to obtain. In addition, this largely static approach makes it difficult to express semantics such as throttling multiple workloads by varying amounts, or to reason about the best possible option in terms of minimizing domino effects due to correlations.

- *Brittleness*: this problem dates from the early days of expert systems, as a consequence of policies getting (unnaturally) encoded into sets of rules with little or no internal structure. In a QAM, making any changes to the workload selection policies is non-trivial. For example, assume that the host workloads now have an additional parameter for relative priorities. Rule specifications will have to be extensively changed to accommodate this change. A preferable solution would allow changes to simply add to the reasoning engine in an incremental way, which is one of the strengths of DECISIONQOS.

- *Order-dependence*: rules with overlapping conditions are common, and the typical way of ensuring that they right one will fire is to make sure they are evaluated in a known order. This fall-through mechanism is error-prone and not intuitive.

- *Lack of adaptivity*: specifications should be *self-evolving*, i.e., the QAM should be able to augment the information encoded in them by observing the behavior of the running system. This is especially required since the workload implications are continuously changing and statically defined rules may not always be effective. Frameworks based on ECA rules do not have this property. It is possible to define *ad hoc* variables within the rules for learning, but the rule-based model does not inherently support learning.

3 System model

Figure 1 depicts our system model. Multiple hosts $H_1, H_2 \ldots H_n$ connect to the storage devices in the *back end*

[2]Predicate $A \triangle B$ is true iff sets A and B have a nonempty intersection.

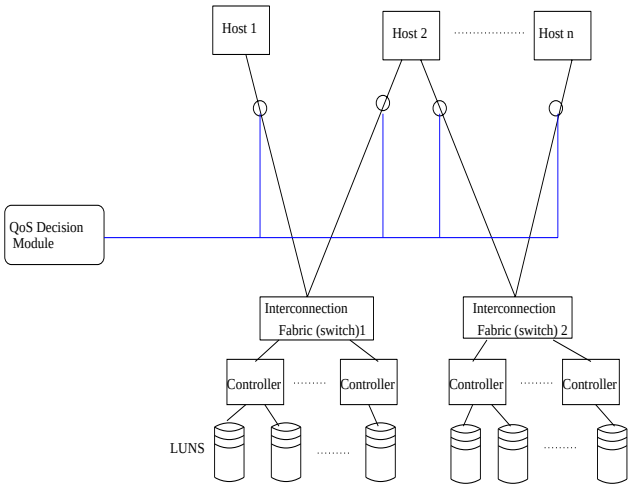

Figure 1. The system model

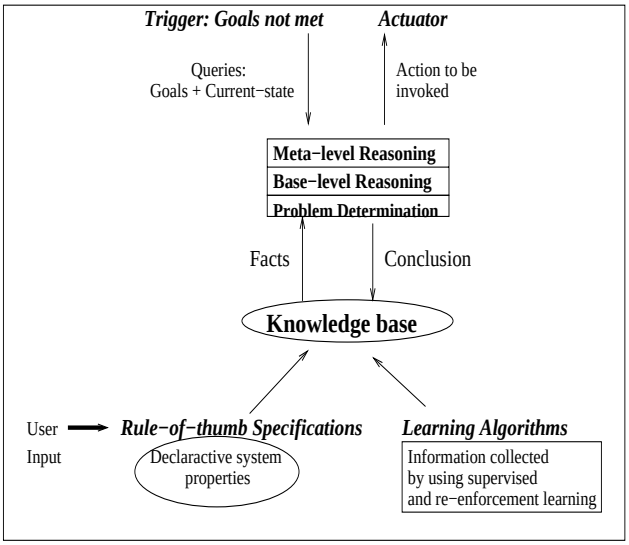

Figure 2. Bird's-eye view of DECISIONQOS

in such a way that the QAM can monitor every single I/O processed by the system. (One way of achieving this property [8] is by instrumenting the code running on a block-level virtualization appliance that is already present in the system to fulfill some other role.) The QAM can gather accurate information on the workload injected by each host, and on the latencies currently experienced by hosts. I/O requests originating from each host are grouped together into one or more workloads, e.g., $workloads(H_1) = \{W_1, W_8, ...W_\alpha\}$, according to which client applications issue them. All requests from a single client hosts could be grouped into a single workload without changing the semantics of the proposed framework, at the cost of making it more difficult to observe patterns at run-time—which are generally less identifiable in an aggregation of streams. Each workload has an SLA associated with it. At the physical level, the storage infrastructure is represented as a collection of elements (e.g., switches) that comprise the interconnection fabric IF; the system also contains the storage controllers SC and the logical disks (or *LUN*s) L. There is a one-to-many relationship between the IF and SC, e.g., $controllers(IF_1) = \{SC_3, SC_7, ...SC_\gamma\}$. Similarly, there is also a one-to-many relationship between SC and L such that a controller can have one or more LUNs that it manages, e.g., $luns(SC_1) = \{L_1, L_3, ...L_\beta\}$.

In this context, the invocation path I of a workload represents the storage components CO used to service the I/O requests of the workload, e.g., $inv\_path(w_1) = \{IF_2, SC_4, L_8\}$. Workloads differ in their access characteristics such as read/write ratio, block-size, sequential/random ratio. The exact amount of resources used by a given workload along its invocation path depend on the workload's access characteristics. In summary, workloads differ from each other in terms of the SLAs associated with them, the access characteristics, the invocation paths, the resources being used to execute the workload.

4 Bird's-eye view of DECISIONQOS

DECISIONQOS is a logic-based approach. Contrary to existing imperative approaches in which the rule-specifications encode the workloads to be invoked for different system states, DECISIONQOS uses a combination of a declarative knowledge-base and logic-based reasoning to derive the workloads at run-time. The declarative knowledge-base defines details of a workload(referred to as *workload-object*, and capabilities of components present within the system. The logic used in the reasoning engine encodes the thinking process that is implicit while writing the rules in imperative approaches. Making the logic explicit allows for more efficient throttling decisions as it is possible to consider choices that are difficult to specify statically e.g. considering combinations of workloads and optimizing the throttling decision based on a particular goal-function such as minimizing side-effects. Additionally, DECISIONQOS proposes an innovative approach for creating the knowledge base, i.e., it uses a combination of specifications and learning algorithms to evolve the workload-objects within the knowledge. Specifications serve to prune to learning-space, allowing for faster convergence.

Within the knowledge base, the workload-objects are represented as "first-class" entities with attributes defining details such as the resource implication of throttling the workload, the invocation path(s) associated with the workload, the SLA goals of the workload. The details of the workload-object are derived using a combination of *declar-*

ative specifications and *learning*. The declarative specifications fundamentally enumerate the *feature-set*, which is used by the learning engine for interpolation.

Reasoning is a three step process. The *problem determination* step derives a list of components whose behavior needs to change. The *base-level reasoning* is expressed in first-order logic and is responsible for searching the workload-objects. It derives the candidate set of workloads that can be possibly used to satisfy the requirements generated by problem determination step. The *meta-level* decides between the candidate workloads using optimization functions. We have developed a working prototype, based on the ABLE toolkit [6], of the learning algorithms and policy-manipulation functions of DECISIONQOS; we plan to implement the remaining parts during the next few months.

5 Design details

The design details of DECISIONQOS are divided into:

- Representation of the workload objects
- Incremental creation of the workload objects using declarative specifications and learning
- Using of two-level reasoning to decide the workloads to be throttled

To make the discussion more concrete, we consider the following example (the example is kept simple for ease of explanation) : There are 3 workloads $W_{e1},\ W_{e2},\ W_{e3}$ operating on the storage infrastructure. The infrastructure consists of a fibre-channel switch IF_α, a storage controller SC_β and two LUNS $L_{\gamma 1}, L_{\gamma 2}$.The workloads access a subset of the LUNs.

5.1 Representation of the workload objects

In DECISIONQOS, the workload objects are first-class entities with the following attributes:

- **Invocation path**: It represents the physical storage components being used by the workload. Additionally, for each component being used, the workload object has information about the percentage of the componentś requests that are generated by this workload. Per-component usage values are dynamic; they are constantly updated by monitoring the system. The invocation path is represented as:
  $I_{e1} = \{(IF_\alpha, V_1), (SC_\beta, V_2), (L_{\gamma 2}, V_3)\}$
  where V_1, V_2, V_3, represents the load as the percentage of the total number of requests handled by the component.
- **Implications**: This represents the resource impact of throttling the workload, i.e., the per-component resources that will be made available as a function of throttling the workload. The impact is dynamic as it depends on the access characteristics of the workload, which are constantly changing.
  $\zeta(W_\alpha, \%throttling) = \{CO_n, \%usage\ change \mid \forall\, CO_n \in I_\alpha\}$
  For example, the implication of W_{e1} is represented as: $\zeta(W_{e1}, t) = \{(IF_\alpha, f(t)), (SC_\beta, g(t)), (L_{\gamma 2}, m(t))\}$ where $f(t), g(t), m(t)$ are functions of the throttling percentage t.
- **Preconditions**: Based on the SLA for each workload (we use both terms interchangeably). The preconditions are generally of the form:
  $\{(x, T_1)\ , (y, T_2)\ \mid \forall x, y\ ((y < T_2) \Rightarrow (x < T_1)) \wedge ((y > T_2) \Rightarrow (Best\_effort))\}$

  For a precondition with performance goals, the preconditions are defined in terms of throughput and latency, where $x \in latency$ and $y \in throughput$.

5.2 Incremental creation of the workload objects using specifications and learning

In DECISIONQOS, the workload objects are generated using an innovative combination of declarative specifications and learning. An alternative would be to observe the system behavior (by monitoring the input and output values) and interpolate the attributes of the workload object using existing machine learning algorithms. For real-world systems, this pure black-box approach is not feasible because the number of observables present in the learning space is huge, making interpolation difficult.

5.2.1 Declarative Specifications

Specifications in DECISIONQOS are non-prescriptive, i.e., they simply enumerate properties of the workload. The entities used in the specifications are referred to as the *feature-set*. The feature-set is used by the learning algorithms for monitoring and interpolation. Specifications are *incomplete* in that they do not fully quantify the values associated with the feature-set. For example, the possible specification is *throttling workload x affects component y*. In this example, y is added to the feature-set of workload x. Further, the specification is not fully quantified in that it did not specify the percentage by which x affects y. The current version of DECISIONQOS takes as input a complete formulation of the invocation paths; this can be obtained from automatic configuration discovery tools.

Specifications in DECISIONQOS define the following:

- ***The precondition associated with the workload*** changes quite infrequently.
- ***The components in invocation path of the workload.*** This information is added to the Invocation path attribute of the workload object. It defines details such as the LUNs being used by the workload, the controllers used to access these LUNs, the port numbers on the switches that connect to these controllers. The interconnection details such as port numbers, etc are based on the physical interconnection and are relatively static. The invocation path specifications are used in conjunction with monitoring to derive information such as the per-component resource usage associated with the workload.

For example, specifications for workload W_{e1} are defined as:
Precondition: {(Latency, 5 ms), (Throughput, 1000 iops)}
Invocation: $\{IF_{\alpha}, SC_{\beta}, L_{\gamma 2}\}$
This information is added to the Preconditions and Invocation path attribute for the workload object W_{e1}.

5.2.2 Learning

Learning is used to derive information associated with the workloads and the components. In DECISIONQOS, learning is a combination of on- and off-line processing. After DECISIONQOS is initially deployed, it simply records the system activity without making any decisions (*training phase*). After initial training, DECISIONQOS then starts making decisions and uses reinforcement learning to refine the literals in the interpolation function. The information derived using learning is as follows:

- ***Implication attribute of the workload object, i.e.,*** ζ Each time a workload is throttled, its percentage change in the usage of each component in its invocation path is measured. A learning function such as a neural net (based on reinforcement learning [12, 18, 24]) is used to interpolate the ζ function. As mentioned earlier, ζ is a dynamic function whose value changes with the access characteristics of the workload namely read/write ratio, block-size, sequential/random ratio.
- ***Correlation between the workloads*** Workload W_a is correlated with W_b if throttling W_a also throttles W_b. Correlation arises due to the application-level dependencies of the workloads. For example, W_a may represent the log associated with the database application while W_b represents the query processing. Workload correlation is represented as a dependency graph with weights associated to the edges of the directed graph (Figure 3). The weights represent the probability, as shown in the figure, that throttling W_1 will affect W_3 is 20%.

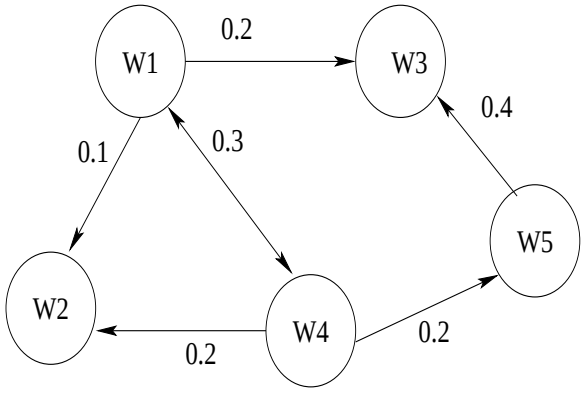

Figure 3. Correlation between the workloads

- ***Behavior model of the storage components*** The aim is to model the relationship between throughput and latency for each physical component such as switches, storage controllers, disks. The traditional, "hockey-stick" relationship between throughput and latency is represented by a curve where latency decreases for increasing throughput up to a point (the "knee" of the curve) beyond which increasing throughput drives latency up. The learning function interpolates the value of throughput, given the value for latency.

5.3 Reasoning

Reasoning is invoked when the SLA for any workload is violated. Reasoning is a three-step process:

- ***The Problem determination*** Determines the list of components whose behavior needs to change.
- ***The Base reasoning*** Based on the list of components whose behavior needs to be modified, this step analyzes the workload objects and derives a list of candidate workloads that can be throttled.
- ***The Meta reasoning*** Using the list of candidate workloads, this step determines the workloads to be actually throttled by optimizing based on the correlation between the workloads.

5.3.1 Problem determination

This step analyzes the components in the invocation path of the workload whose SLA is violated. The analysis uses the throughput-latency model of the component in conjunction with its current usage values. The algorithm for problem determination is as follows:
Input: Workloads $W_1, ..., W_\theta, ...$ whose SLA is violated
Output: A set η where each element is of the form: *(CO, Maximum change in throughput, Maximum change in*

latency)

Approach: For each workload W_θ whose SLA is violated:

- Determine the list of components in the invocation path $I_\theta = \{IF_\theta, SC_\theta, L_\theta\}$ of W_θ. Let the required change in latency for W_θ be χ.
- We want to determine the required change in the throughput of each component such that the overall latency of I_θ is reduced by χ.
- For each component, use the throughput-latency model. If the current operating point of the component is above the "knee point" (i.e., high throughput implies high latency), then continue. Else select the next component in the path.
- The maximum throughput change for a component is defined as moving the current operating point to the knee of the curve (i.e., beyond the knee, a change in throughput does not change latency). The corresponding change in latency is referred to as the maximum change in latency.

For example, assume W_{e3} is not meeting its preconditions (Goal latency = 6 ms, Current value = 10 ms, iops within specified threshold). The problem determination module analyzes the invocation path of W_{e3} which consists of $IF_\alpha, SC_\beta, L_{\gamma 2}$. The average latency of $IF_\alpha = 0.5ms$, $SC_\beta = 2ms$, $L_{\gamma 2} = 7.5ms$. Looking at throughput-latency for each of these components, the problem determination module determines that the number of requests serviced by SC_β can reduced by a maximum of 18% to reach the knee point, while that for $L_{\gamma 2}$ by 43%. Component IF_α is operating at the knee point and does not need any changes.

5.3.2 Base reasoning

This step searches the workload objects. The semantics for searching the specifications are expressed in first-order logic.
Input: A set η of components whose behavior needs to change.
Output: A set where the elements are of the form: (Workload, %*Maximum possible throttling*)
Approach: Base reasoning is similar to constraint-solving. The input from the problem determination module can be represented as a constraint Ω: *"Find all workloads that actively use the components in η"*. The constraint is solved by analyzing the attributes of the workload object that are internally represented as *sentences* in propositional logic.

In simple words, the semantics for constraint solving are expressed in first-order logic as: *" A workload that is exceeding its SLA limits $\wedge$ Has the specified component in its invocation path $\wedge$ Has a non-zero implication function ζ for the specified component."* Inference is carried out using first-order logic operations. In what follows, let w stand for a workload, c for a component, *curr_state* for the state of the system, and α for a requirement set such as η.

$\forall\ w, \alpha\ \ Satisfy(\alpha) \Rightarrow S_{Precondition}(w) \wedge S_{Invocation}(w, \alpha) \wedge S_{Implication}(w, \alpha)$

$\forall\ w\ \ S_{Precondition}(w) \Rightarrow \forall\ y\ Precondition(w, y) \wedge Greater(Throughput(y), Throughput(curr\_state))$

$\forall\ w, \alpha\ \ S_{Invocation}(w, \alpha) \Rightarrow \exists\ c\ Invocation(w, c) \wedge Equals(Component(\alpha), c)$

$\forall\ \ w, \alpha\ \ S_{Implication}(w, \alpha) \Rightarrow \exists! c\ \ Component(\alpha) \wedge Implication(w, c) \wedge Greater(Interpolate(w, Access\_Patt(curr\_state), c), 0)$

$$Equals = \{(x, y) \mid \forall\ x,\ y\ string(x) = string(y)\}$$

The function $Interpolate(x, y, z)$ approximates the impact of throttling workload x on component z, for its current access pattern y.
The base reasoning in addition to deciding the candidate workloads also prescribes the maximum allowable throttling, defined as follows:

$\forall\ w,\ t\ \ MaxThrottling(w, t) \Rightarrow Greater(t) \wedge \neg\ S_{Precondition}(w)$

In the example above, α is a workload that affects $(IF_\alpha \vee SC_\beta)$. Workload W_{e1} is selected as a candidate workload since the predicate $S_{Precondition}(W_{e1}) \wedge S_{Invocation}(W_{e1}, \alpha) \wedge S_{Implication}(W_{e1}, \alpha)$ is true.

5.3.3 Meta reasoning

This step selects the workloads to be throttled from the set of candidate workloads. This selection is based on an optimization goal such as minimizing the side-effects of throttling the workload, or minimize variance in resource utilization, etc. We describe details of meta reasoning with the goal function for minimizing the side-effects of throttling workloads.
Input: A set of candidate workloads W_C which is of the form (Workload, %*Max. possible throttling*, impact function for each component); plus a set of the form (Component, Max possible change in throughput); plus a set of constraints of the form (Set of components, Total required change in latency)
Output: The throttling decisions which is a set of the form: (Workload, $\%Throttling$)

Approach: As mentioned earlier, the correlation between the workloads is represented as a directed graph (as in Figure 3). The weight on the edges is expressed as a probability $P(A \Rightarrow B)$.

- Let the set of all the workloads be represented by W.
- Calculate the weight associated with each workload in the set of candidate workloads W_C
  $Weight_\Phi = \Sigma_{\Omega \in (W \cap W_C)'} P(\Phi \Rightarrow \Omega)$
- The elements in W_C are sorted in ascending-order based on their weight. The smaller the weight, the more preferable is the workload.
- Throttle first element in W_C. If (SLA met) then terminate. Else remove the first element,and repeat this step.

6 Related work

The related research is divided into two domains:

- Resource arbitration frameworks
- Policy-based management and AI-based frameworks

6.1 Resource arbitration frameworks

Resource arbitration frameworks such as Façade [21] provide a per-workload storage performance monitoring and QoS enforcement capabilities. Façade is built using a central scheduler that regulates the rates of I/O workloads accessing a common storage container such as a RAID logical disk. Façade does not account for competing workloads sharing resources in various degrees, e.g., two logical units in the same vs. in different disk arrays; and it throttles all workloads to similar degrees when QoS violations occur. Sleds [8] can selectively throttle only 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.

The problem of resource arbitration has also been addressed in domains other than storage systems. Many networking solutions [3] are based on selectively dropping packets [7]. They do not extend to widespread storage access protocols [2] for multiple reasons [28], including the severe consequences of packet loss. Proposals like DiffServ [7] are not rich enough to distinguish all service classes that may need to be treated differently.

6.2 Policy-based management and AI-based approaches

Policy based infrastructures have been used to automate the task of management [22, 26]. In these applications, the underlying policy specification model is based on ECA. There are multiple approaches (i.e., syntax) for specifying policies: Specified in terms of a special language that is processed and interpreted as a piece of software [4] or in terms of a formal specification language [9] or the simplest approach is to interpret policies as a sequence of rules. The IETF has chosen rule-based policy representation in its specifications [17, 23]. The problems of brittleness and complexity are one of the primary reasons limiting the wide-spread usages of policy-based management (which is precisely the problem a logic-based approach such as DECISIONQOS aims to solve).

A variation of policy-based management has been proposed in [27]. They use a Case-Based Reasoning approach, in which a system starts off with no specifications and uses the previously learnt cases to decide how a goal should be transformed, has been employed in the webserver configuration domain. The bootstrapping behavior of that approach is not attractive in real-world scenarios where the reasonable number of cases that need to be learned a priori are 0 (resource states, workload characteristics, goals, action set). Bearden *et al.* [5] propose an approach to separate the goal from the base rule specification. They create a mapping between each rule and user requirements, making it easy to validate a rule set. The DECISIONQOS approach is more sophisticated, in that it encodes the goal implications and uses them to automate the reasoning process.

An approach that uses genetic algorithms for self-tuning has also been proposed [11]. In this approach each system parameter is tuned by an individual algorithm and the genetic algorithm decides the best combination of algorithms. Unlike DECISIONQOS, this approach does not allow refinement of the decision-making based on learning. Zinky *et al.*, [30] present a general framework, called QuO, to implement QoS-enabled distributed object systems. The QoS adaptation is achieved by having multiple implementations. Each implementation is mapped to an environment and a QoS region. This approach is static, as it does not implement semantics for reasoning about the various possible configurations.

DECISIONQOS leverages concepts in AI and uses them as building blocks in its solution. Techniques for specification in expert systems are broadly classified as imperative (e.g., rule-based), declarative (e.g., logic programming) or mixed. Brittleness has been identified as the biggest drawback of imperative rule-based systems [10], whereas logic based systems overcome this problem by using a reasoning engine to combine facts/beliefs in the knowledge base

to draw conclusions. The DECISIONQOS specification of action attributes is similar to the declarative approach. Further, reasoning in DECISIONQOS is a combination of specification search algorithms and higher-order operations. DECISIONQOS uses forward chaining to search the specifications, but it is possible to use other approaches such as backward chaining or heuristic-based searching. Other popular approaches for reasoning are: Model-based, Constraint-based, and Case-based reasoning [20]. Finally, learning in DECISIONQOS systematically refines the specifications. It leverages research in the domain of machine learning algorithms such as neural networks and reinforcement learning [12, 18, 24].

7 Conclusions

Most applications running on an enterprise data center depend on getting minimum performance levels from the storage system; if that cannot be provided, they fail. The typical scenario where many applications compete for relatively few high-end resources such as network switches and disk arrays is not well suited for predictable sharing. Because of their workload characteristics and of scheduling idiosyncrasies, resources will not be distributed according to each application's needs in the absence of a regulating entity.

We present DECISIONQOS, a novel paradigm for building resource arbitration modules, and discuss its application to storage systems. DECISIONQOS relies on declarative specifications and on machine learning techniques to keep an up-to-date body of knowledge about the storage system and the workloads running on it. This body of knowledge captures the concepts of physical and logical resource sharing, dependencies and correlations among different workloads, and fluctuations in the performance experienced by clients as a result of workload or system changes. DECISIONQOS does not require detailed descriptions as its initial input; system administrators can just supply whatever information is available to them, and DECISIONQOS will supplement and/or amend it by dynamically observing the system's behavior.

DECISIONQOS relieves users from the burdens (common in rule-based systems) of coding policies into unstructured sets of event-condition-action rules. Such rule sets are hard to tune, modify, and maintain, for they require users to foresee at rule-creation time all the relevant families of system states, the threshold values that determine when actions should be taken, and the particular actions prescribed for each state. In contrast, DECISIONQOS hides from users the complexity of individual decisions, letting users concentrate on the declarative, high-level aspects of system behavior. The important point about this work however is not that a declarative specification is preferable but rather that the only rational course of action in QoS management is to relate policy to observation. This is because provisioning for QoS has intrinsic uncertainties that can only be solved by observing the system. The end result is that DECISIONQOS does not depend on human experts, and is significantly more resilient to the inevitable changes that will arise in practical systems.

References

[1] G.A. Alvarez, E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[2] ANSI. SCSI architecture model - 2 (SAM-2), September 2002. Draft Standard, Project 1157-D, Revision 24.

[3] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151, 1998.

[4] J. F. Barnes and R. Pandey. CacheL: Language support for customizable caching policies. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[5] M. Bearden, S. Garg, and W.J. Lee. Integrating goal specification in policy-based management. In *Proc. Int'l Workshop on Policies for Distributed Systems and Networks*, January 2001.

[6] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W.N. Mills III, and Y. Diao. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Sys. J.*, 41(3), September 2002.

[7] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. *IETF RFC 2475*, 1998.

[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] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: An environment for goal-driven requirements engineering. In *Proc. ICSE'98 - 20th Intl. Conference on Software Engineering*, 1998.

[10] V. Dhar and H.E. Pople. Rule-based versus structure-based models for explaining and generating expert behavior. *Comm. ACM*, 30(6), June 1987.

[11] D. Feitelson and M. Naaman. Self-tuning systems. *IEEE Software*, 16(2):52–60, 1999.

[12] J. Ghosh and A. Nag. *An Overview of Radial Basis Function Networks*. Radial Basis Function Neural Network Theory and Applications, Physica-Verlag, 2000.

[13] C. Green. Application of theorem proving to problem solving. In B. L. Webber and N. J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 202–222. Kaufmann, Los Altos, CA, 1981.

[14] F. Hayes-Roth. Rule-based Systems. *Comm. ACM*, 28(9), September 1985.

[15] C. Hewitt. Planner: A language for proving theorems in robots. In *Proc. of the 1st IJCAI*, pages 295–301, Washington, DC, 1969.

[16] C. Hewitt. Procedural embedding of knowledge in planner. In *Proc. of the 2nd IJCAI*, pages 167–182, London, UK, 1971.

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

[18] T. Kohonen. *Self-Organizing and Associative Memory 3rd ed.* Springer-Verlag, 1988.

[19] R. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of the Sixth IFIP Congress (Information Processing 74)*, pages 569–574, Stockholm, Sweden, August 1974.

[20] D.B. Leake. *Case-Based Reasoning: Experiences, Lessons and Future Directions*. AAAI Press, 1996.

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

[22] E. Lupu M. Sloman. Security and management policy specification. *IEEE Network*, March 2002.

[23] B. Moore. Network Working Group – RFC3060. Policy Core Information Model – Version 1 Specification. http://www.ietf.org/rfc/rfc3060.txt, 2001.

[24] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Internal Representations Through Error Propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Experiments in the Microstructure of Cognition, Vol. 1*. MIT Press, 1986.

[25] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease. Polus: Growing storage QoS management beyond a 4-year old kid. In *Proc. 3rd Conf. on File and Storage Technologies*, March 2004.

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

[27] 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.

[28] J. Wilkes. Travelling to Rome: QoS specifications for automated storage system management. In D. Hutchinson L. Wolf and R. Steinmetz, editors, *Proceedings of 9th International Workshop on Quality Of Service (IWQoS)*, pages 75–91. Springer Verlag, June 1991.

[29] T. Winograd. Frame representations and the declarative/procedural controversy. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 357–370. Kaufmann, Los Altos, CA, 1985.

[30] J. A. Zinky, D. E. Bakken, and R. D. Schantz. Architectural support for Quality-of-Service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.