Polus : Growing Storage QoS Management Beyond a “Four-year Old Kid”

Sandeep Uttamchandani Kaladhar Voruganti Sudarshan Srinivasan*
John Palmer David Pease

{sandeepu,kaladhar,smsriniv,jdp,dpease}@us.ibm.com

IBM Almaden Research Center,
650 Harry Road, San José, California 95120, USA

Abstract

Policy-based storage management has been advertised as the silver bullet to overcome the complexity that limits the amount of storage that can be managed by system administrators. Key to this approach are: a mechanism to specify quality of service (QoS) goals; a canonical virtual model of storage devices and operations; and the mapping of the high level QoS goals to low level storage device actions. In spite of prior research and industrial standards the latter problem results in complex, manual, error-prone processes that burden system administrators and prevent the widespread acceptance of policy-based storage management. This paper proposes the Polus framework which specifically addresses this open problem.

Polus removes the need for system administrators to write code that maps the QoS goals to low level system actions. Instead, it generates this mapping code by using a combination of rule-of-thumb specification mechanism, a reasoning engine and a learning engine to change the implementation paradigm of policy-based storage management. This paper also provides a quantitative analysis of the Polus framework within the context of a storage area network (SAN) file system to verify the feasibility of this new approach.

1 Introduction

Capacity planning, application/storage performance management, backup/restore operations, configuration management, security, and availability analysis are some of the key storage management responsibilities of a system administrator. Typically, storage administrators write scripts that automate many of these storage management tasks. As the number of business service level agreements, department policies, QoS goals, storage devices, protocols, applications, and users increases, it becomes difficult for system administrators to ensure performance, provisioning, availability and security goals by using ad hoc script writing approaches . Thus, systems management has been identified as one of the most important research areas by many leading researchers [9, 26]. Storage vendors are trying to add sophisticated systems management functionality into databases, file systems, storage controllers, storage resource managers, storage area network managers, capacity planning managers and other storage management software. The major focus of these products is to reduce management complexity by allowing a system administrator to specify high level QoS goals with respect to expected performance,availability, provisioning, and security, and to automatically transform these high level QoS goals into low level system actions.

Currently, this transformation process is built using the policy-based paradigm, where policies are specified as collection of rules that are in the ECA format (Event-Condition-Action) [14]. Rules define how the system behaves for different possible system states and goal values. At run-time, the management module simply invokes the rule that is applicable based on the event and system condition. Even though goal based storage management approach has been advocated as the silver bullet that can help to reduce the management complexity for system administrators, this approach has not gained much traction because current policy management frameworks are providing support for only simple and trivial storage management scenarios.

1.1 Problem Statement

Design of high level QoS goals to low level storage actions transformation mechanisms in management software is done by experts with many years of prior experience as system architects and administrators. However, even the experts are encountering the following types of problems while designing robust storage management systems:

Complexity: The level of details, required to write the specifications is non-trivial. ECA rules are written as a certain storage management action being taken when a system observable violates a predetermined threshold. The transformation code is specified as ECA rules (e.g. if $throughput\_goal\_violated$ AND $Sequential/Random\_ratio > 1$, then increase data

*IBM Summer intern, 2003. Affiliation: Department of Computer Science, University of Illinois at Urbana Champaign. E-mail:smsriniv@cs.uiuc.edu prefetching size by 20%) where actions are taken upon the violation of threshold values. It is difficult for the composers of ECA rules to: (1) choose which combination of system parameters to observe from a large set of possible observables; (2) determine appropriate threshold values after considering the interaction of a large set of system variables; and (3) select a specific corrective action from the large set of competing options. As the number of users, storage devices, storage management actions, and service level agreements increase, it becomes computationally exhaustive for system administrators or storage management tool developers to consider all the alternatives.

Brittleness: It is difficult for vendors to provide prepackaged transformation code with their products because this code becomes brittle with respect to changing system configurations, user workloads, and department/business constraints. It is difficult for the storage management vendors to envision all of the potential use case scenarios ahead of time, and thus, many of the current storage management solutions provide work-flow environments which, in turn, pass the responsibility of transforming high level QoS goals (via work-flow scripts) to an organization's system administrators and infrastructure planners.

To summarize, we restate the discussion-panel conclusion in FAST'03: Existing storage management frameworks are like a "four-year old kid"- "They mess up more than they are actually useful."

1.2 Bird's eye view and Contributions of Polus Framework

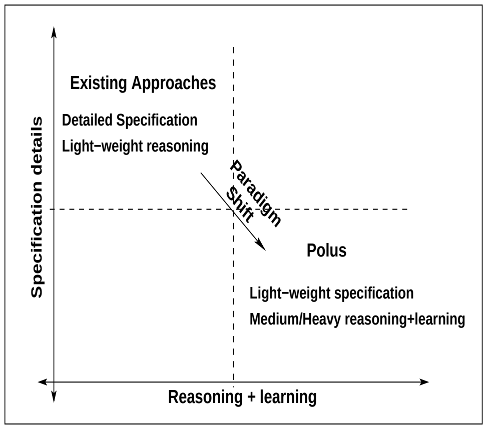


Figure 1. An innovative approach for QoS management

This paper proposes the Polus framework that tries to take away the complexity of writing policy mapping code from human experts and moves it to a combination of reasoning and learning engines (as shown in Figure 1). The Polus framework addresses the complexity and brittleness problems described in section 1.1, in the following manner.

In Polus, as shown in Figure 2, the system administrator inputs knowledge in the form of rules of thumb. For example, "To invoke Prefetch action *requires* memory", "Invoke Prefetch action *requires* the workload to be Sequential", "Prefetch action *improves* throughput." The generic form of the specification is (<*Relationship* >[*Type of relationship*]) where the *Relationship* is between *actions* and *resources, workload characteristics, system behavior.* The optional *Type of relationship* gives a hint about the nature of the relationship (e.g., *improves, requires*). It should be noted that the rule-of-thumb specifications do not require the system administrator to quantify the threshold values for actions, observables, workload characteristics and resources. In addition, he does not have to spell-out the details of the action to be invoked for a given state of resources, workloads, and observables.

The relationships defined in the rule-of-thumb specifications are quantified by the use of a learning engine. The system management actions, the state of the system resources/workload characteristics when the a particular action was taken, and the current values of the observables (e.g., throughput, latency, etc.) are monitored and stored as part of the knowledge base. The learning engine uses this monitored information to predict and quantify the relationships described in the specifications; for example, "Prefetching improves throughput when available memory is greater than 20 percent", and "Use prefetching when Sequential/Random ratio is greater than 0.4." It is important to note that the rule-of-thumb specifications help to prune the number of variables used in the interpolation function, which in turn helps improve the convergence rate of the learning function. For example, while interpolating relationships of the prefetch action, the system is not required to take into account observables related to security. In the current implementation of Polus, the learning engine does not discover additional relationships (apart from those in the specifications) and also assumes that the hints in the specifications are correct. In the future, these assumptions will be addressed using learning approaches such as "bagging" [5].

When a particular QoS goal is violated in the system, the Polus reasoning engine is invoked. The semantics of the reasoning engine are expressed in first-order predicate calculus and are similar to the thought-process that is implicit in ECA rules. For example, an action (x) can be invoked only if the resource required (*precondition*) for its invocation are available in the current-state (cs). The current-state is defined in terms of values of resources, workload characteristics, and observables. This is expressed in first-order predicate calculus as:

$$\forall\, x, invoke(x, cs) \;\Rightarrow$$
$$(available(cs) \;>\; precondition(x))$$

Using the current-state as input and the information (i.e., combination of specifications and learning) in the knowledge base as facts, the reasoning engine derives the actions to be invoked at run-time. For example, when the reasoning engine tries to invoke the prefetch action the *invoke* function is instantiated as $invoke(prefetch, cs)$ which will be true if

$$(available(cs) \;>\; precondition(prefetch))$$

In this predicate, cs is unified with the values that were passed as input to the reasoning engine, and the information of the prefetch action is retrieved from the knowledge base.

It is important to note that combinations of declarative specification and predicate calculus (as done in Polus) are the basis of the well-known field of logic based programming [16, 10]. Polus is applying and extending these concepts for the domain of storage management.

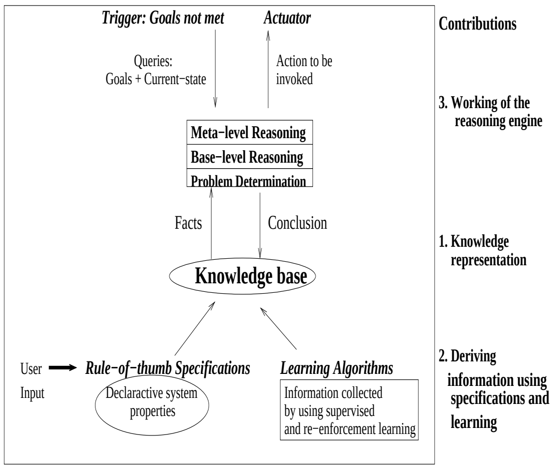

Trigger: Goals not met
Actuator
Queries: Goals + Current-state
Action to be invoked
Meta-level Reasoning
Base-level Reasoning
Problem Determination
Facts
Conclusion
Knowledge base
User Input
Rule-of-thumb Specifications
Declaractive system properties
Learning Algorithms
Information collected by using supervised and re-enforcement learning
Contributions
3. Working of the reasoning engine
1. Knowledge representation
2. Deriving information using specifications and learning

Figure 2. Contributions of Polus

1.3 Paper Organization

Sections 2 and 3 provide the details of ECA and Polus respectively. Section 4 describes the experimental framework. Section 5 presents the experiment results. Section 6 contains a discussion of the experiment results. A survey of previous expert systems, policy frameworks and storage management solutions is provided in section 7. Finally, section 8 presents our concluding remarks.

2 Background: QoS Management Using the ECA Approach

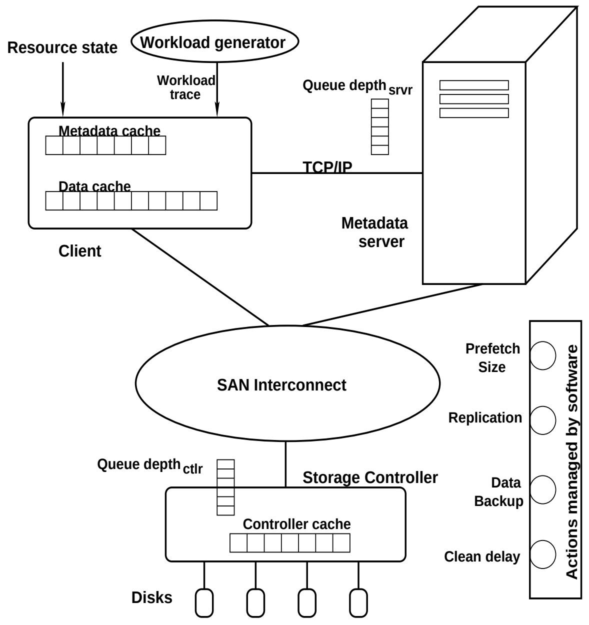


Figure 3. Simulator of a SAN file-system

In this paper, the effectiveness of the competing ECA and Polus management approaches will be discussed using the example of a storage area network (SAN) file system [12, 21]. As shown in figure 3, in a SAN file system, the clients contact a metadata server to obtain the necessary metadata for a particular file. Subsequently the clients go directly to the storage controllers via a SAN protocol to access the storage. The clients cache both the file metadata information and the user block data in two separate caches. In order to write ECA rules for this system, a system administrator needs to do the following:

Establish Goals: System administrators are usually interested in ensuring that certain performance (throughput, latency), reliability and security goals are being met in their SAN file system deployments. For example, they could specify their QoS goals as: (1) ensure that each client has a throughput of at least 40MBps; and (2) ensure that the system has 99.999 percent availability.

Determine the observables to analyze: System administrators have access to many static and dynamic system observables such as the available memory size, the SAN bandwidth being provided to a particular client, and the cache hit rate at the client, the metadata server or the storage controller. The system administrators also have to access to workload characteristics such as the read/write ratio, workload type (random or sequential), and the block size.

Assess the available actions: System administrators need to be aware of the different possible storage actions that they can perform to manage the storage, such as replication, migration, clean delay [18], request throttling, zoning etc.

Determine thresholds for the observables: Based on prior empirical data or experience, system administrators need to determine the threshold values which, when violated should result in the triggering of corrective management actions. For example, if $cache\_miss\_rate > 20\%$ then take a corrective action.

Select a particular action: If the threshold value of a particular observable is being violated then the system administrator needs to choose a correction action such as increasing the prefetch size or replication of data.

Determine the granularity of the action: For example, when the corrective action being taken is to increase the data prefetch size, then the system administrator needs to also specify the unit of the prefetch size increase.

To put it all together, if the QoS goal of 10 millisecond latency is not being met for a particular client, then the system administrator needs to write the following sets of ECA rules (not exhaustive) to find a remedy:

[Rule 1] If the throughput of a storage controller is at its maximum, then migrate this client's data to another controller that has the necessary available bandwidth.

[Rule 2] If the *client cache miss rate* > 20% and the workload is sequential then increase the data prefetch size by 4 objects.

[Rule 3] If a particular client is exceeding its allotted bandwidth then throttle its request.

Thus, for a particular QoS goal, the system administrator needs to evaluate the values of all the relevant system observables, assess whether they are violating a particular predetermined threshold value, and then choose a corrective action from a list of possible system management actions. The objective of Polus is to reduce the number of details that a system administrator needs to consider.

3 Details of the Polus Approach

This section presents the details of the Polus framework by both presenting the technical details of the framework and also by illustrating how Polus provides storage management guidance to a Storage Area Network (SAN) file system.

3.1 Polus Terminology

Before describing the system model, we define the terms *behavior, goals and actions.* In Polus, behavior is a defined as a set of QoS dimensions such as $\{throughput, latency, availability, reliability\}$. These dimensions are also referred to as observables. Goals are defined as threshold values on behavior dimensions. e.g. response-time for reads should less than 5 sec, a client's average throughput should be 150 MBps etc. The definition of actions is domain-specific. In Polus, the actions are divided into two categories: a) tunable parameters i.e., changing the value of configuration parameters such a prefetch size, clean-delay, etc., and b) internal actions such as migration, replication, and back-up of data.

The model of the system is shown in figure 4. It consists of two key entities: the management software and the managed system. The management software is assigned QoS goals and is responsible for ensuring that the managed system meets these goals. The interaction between the management software and the managed system is via *sensors* and *actuators.* Monitors gather information about the managed system, while the actuators effect the actions invoked by the management software on the managed system.

The managed system consists of physical *components* that interact to service the application requests. The components service the requests in a particular sequence, with each component processing the information before handing it to the next component. For example, within a SAN file system, the components are client machines, metadata servers, storage controllers and disks. A component has pre-defined properties such as the maximum number of requests that it can service, the error rate, the average down-time etc. Each component consists of one of more atomic entities referred to as *resources.* In other words, resources serve as an abstraction to refer to components in a generic fashion. In Polus, the possible resources are memory, CPU, network, and storage. For example, the storage controller component consists of memory and storage resources, the client machine component consists of CPU, memory and network resource.

Sensors collects information about the *state* of the managed system. The state of the system is defined as a quadruple $<S$, $R_{current}$, IP, $B_{current}>$, defined as follows:

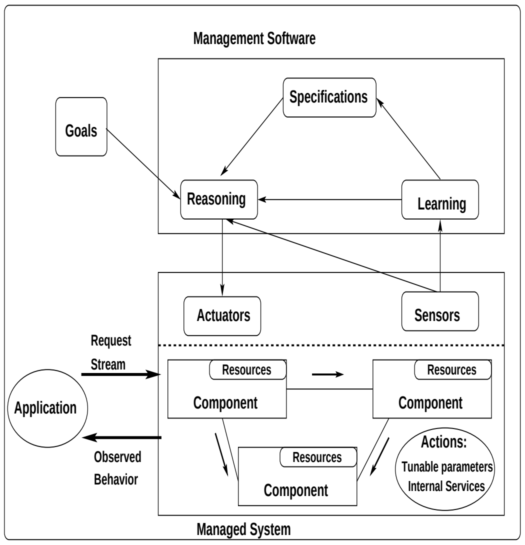


Figure 4. System model

- $Workload\ Characteristics\ (S)$: S is represented as set of measurable parameters that characterize the application requests. These characteristics are dynamic and constantly changing. For example, in a storage system, typical dimensions are the read/write ratio of access requests, the access pattern (sequential/random), and block size of requests.
- $R_{current}$: Represents the current usage of the resources on a per component basis.
- $Invocation\,Path\ (IP)$: Represents the sequence in which the components are invoked while servicing the application requests. For example, client machine $\rightarrow$ storage controller 2 $\rightarrow$ LUN 8 represents a possible invocation path
- $B_{current}$: Represents the current values of the behavior dimensions.

Actuators invoke the *actions* selected by the management software. The impact of invoking an action is not constant and is a function of the *state*. Furthermore, whether an action can be invoked or not is dependent on the *state* (i.e. not all actions are applicable within a particular *state*).

3.2 Polus Framework

The three main parts of the Polus framework are:

Modeling of information: The model describes how actions such as prefetching, replication, etc are represented in the system.

Generation of the knowledge base: How rule-of-thumb specifications and learning are used in conjunction to generate the details of the information model.

Reasoning: During the detection of a QoS violation, the reasoning engine decides which action to invoke using the information in the knowledge base. The management semantics of the reasoning engine are expressed using first-order predicate calculus.

3.2.1 Modeling of Information

In Polus, an action is represented as a software object and is referred to as an *action object.* The attributes of the action object are a triplet of the form $< I,\ P,\ B >$, defined as follows:

$Implications\ I$: This defines the impact of action invocation on the system behavior. The value of the impact function is dependent on the current state.

$Preconditions\ P$: This represents the dependencies of the action on system state (i.e. the prerequisites for invoking the action). The prerequisites are defined in terms of thresholds on resources and workload characteristics. Preconditions can be visualized as defining boundaries to the state-space, since the impact of invoking the action beyond the boundaries is not captured by the action object. Preconditions are represented as exclusion/inclusion lists.

$Base\ behavior\ B$: This attribute defines the details associated with the actual action invocation. This includes parameters that need to be passed during invocation function(e.g., to invoke prefetching, the prefetch size needs to be passed), the increment size, and the transient resource costs for action invocation. The details of action invocation are defined in terms of unit invocation, which is similar to that used in implications.

Figure 5 gives the example of the prefetch action object. More complex actions such as replication have additional base invocation parameters such as number of replicas, the data-set to replicate, selecting the component where the replicas will be stored. Polus assumes that the semantics for generating the values for these replication parameters will come from a separate resource planning tool.

3.2.2 Generation of Knowledge base

In Polus, the details of the action object are generated using a combination of rule-of-thumb specifications and learning. Figure 6 shows how the prefetch object is internally maintained by Polus. The details of the action object may not all be available when the system starts

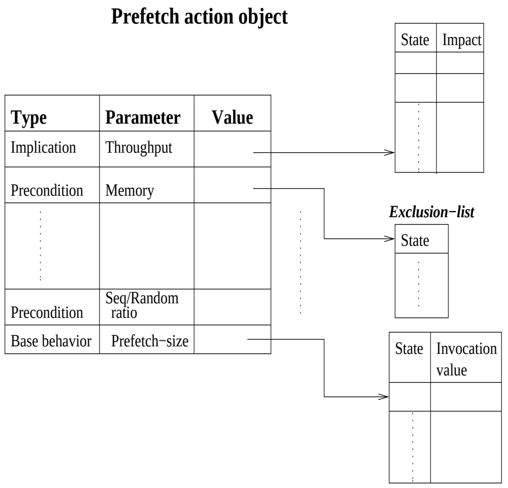


Figure 5. Example of a Prefetch action object

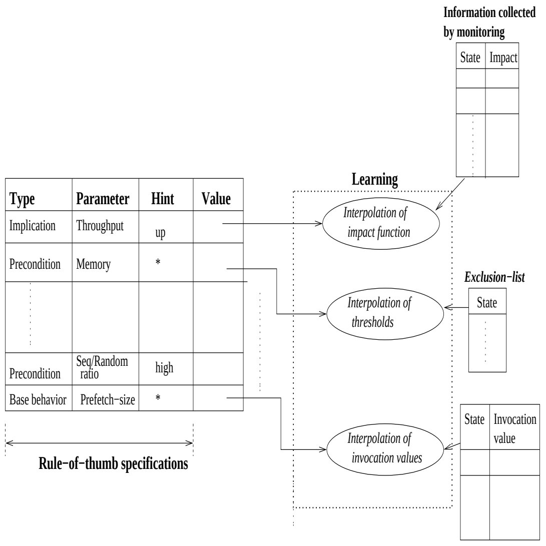


Figure 6. Prefetch action object internally maintained by Polus

off. A part of the information is generated using user-defined specifications (rule-of-thumb) and rest is generated using learning functions (as will be explained in this subsection). There is a "confidence" value associated with every piece of generated information. The confidence value is based on the error value of the learning algorithm that signifies the difference between the predicted value and the actual observed value.

Rule of thumb specifications

The rule of thumb specifications are simple declarative statements. They fundermentally serve two purposes:

- They specify the relationship between actions, resources, workload characteristics, and behavior dimensions. For example, prefetching requires memory.
- They specify "hints" for the possible values associated with the relationships. For example, prefetching is dependent on the sequential/random ratio. A higher value of this ratio is more favorable. This hint helps the learning algorithm to perform linear classification within the behavior space (i.e., if invoking prefetching did not improve throughput when the sequential/random ratio = β, then the hint allows the learning algorithm to interpolate that for all values less than or equal to β, invoking prefetching may not be beneficial).

The template for the rule-of-thumb specifications has similar categories as those of the action objects:

```
<action name = PREFETCH>
  <behavior_implications>
     <implication dimension = throughput
           impact = up>
  </behavior_implications>
  <preconditions>
     <precond dimension = sequential/random
           ratio value = high>
     <precond dimension = read/write ratio
           value = high>
     <precond dimension = memory value = *>
  </preconditions>
   <base invocation>
     <function name = change_prefetch_size>
     <parameter type = float>
   </base invocation>
</action>
```

Learning

Learning algorithms quantify the rule of thumb specifications and they interpolate the value sets for the relationships defined in the specifications. A learning algorithm is treated as a black box that interpolates information for the $(n+1)^{th}$ data point given a previous sample of n data points. The rule-of-thumb specifications help in pruning the learning space. For example, the implication of invoking prefetching is a function of all the observables, i.e., $Implication(Prefetching) \rightarrow f(all\ observables)$. Using the rule of thumb specifications, the interpolation of the implication function is: $Implication(Prefetching) \rightarrow f(throughput, latency)$.

Given that specifications prune the learning space, the question of what happens if the rule-of-thumb specifications are incomplete arises. The current implementation of Polus does not handle this scenario as it assumes that the specifications are complete. However, one can overcome incomplete specifications using existing machine learning approaches such as "bagging" [5], which discover unspecified relationships and add them to the specifications.

In Polus, the process of learning is a combination of off-line training and on-line refinement. Initially, when the management software is installed, learning is an off-line process, which means that the learning algorithms are just recording the system state along-with the actions invoked by the administrator. After a sufficient number of training data points are recorded, the learning algorithm switches to the on-line approach in which it keeps refining the interpolation function generated using the training data points. This refinement is based on the difference between the interpolated value and the value actually obtained from the invocation (also referred to as re-enforcement learning [20]).

Conjunction of specifications and learning

The attributes of the action object are derived using a combination of specifications and learning. In the current implementation of Polus, the rule-of-thumb specifications forms the static part of the knowledge base i.e., the system does not discover additional relationships. The information associated with the learning algorithms forms the dynamic part of the knowledge base as this information is constantly updated by monitoring the system and refining the interpolation functions.

The reasoning engine accesses the information in the knowledge base. The access is a query of the form *Does Action X affect throughput and if yes, by how much?*. The query is translated into a sub-query which is first handled by the specification sub-part of the knowledge base that looks-up to see if there is a relation defined between Action X and throughput. If yes, it generates another query for the learning sub-part that contains the interpolation function for action X, current-state and throughput. Figure 7 illustrates this in the context of prefetching.

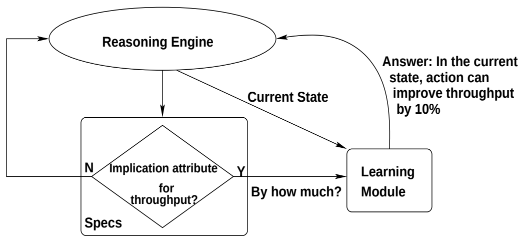

Reasoning Engine
Answer: In the current state, action can improve throughput by 10%
Current State
N
Implication attribute for throughput?
Y
By how much?
Learning Module
Specs

Figure 7. Conjunction of knowledge with the specifications and learning module

3.2.3 Reasoning

The reasoning engine is triggered whenever any of the assigned goals are violated. The objective of the reasoning engine is to select the action(s) that need to be invoked at run-time, in response to the violation of goals. The operation of the reasoning engine is a three step process:

Problem determination: Analyzes the components in the invocation path; determines components that are saturated or the components that need to be tuned for the current state.

Base-level reasoning: Searches the action objects, based on the queries generated by the problem determination module (can be visualized as an object interpreter).

Meta-level reasoning: The meta-level is responsible for higher-order optimizations such as deciding between multiple candidate actions based on invocation cost, error functions, side-effects and so on.

In the SAN file system example, if the prescribed throughput goal of 100 MBps is being violated, the steps that are taken by the reasoning engine to rectify the problem are described in the following subsections.

Problem determination

Input: The current state and the goals being violated.
Output: The components whose behavior needs to change and the type of change. This is expressed as a query of the form:

$$\phi = \{(c, b, change) | c \epsilon Component, \\ b \epsilon Behavior, b_{new} = (1 + change) b_{current}\}$$

Approach: Problem determination has been an area of ongoing research. We briefly describe a simplified approach to illustrate how problem determination, base-reasoning and meta-reasoning work together.

- Determine the components being used in the invocation path.
- Compare the current behavior of these components with the static capabilities of the components. This is similar to system diagnosis using model-based reasoning [22].
- Additionally, compare the current state with a previous state (in which the goals were met). This change analysis generates additional parameters used to search the specifications and tune components in the invocation path that are not saturated. The changes in the system state can be along one or more of the following dimensions: a) Resources utilization (accounts for failures, resource-additions, and application request rate), b) Workload characteristics c) Assigned goals, d) Invocation path (accounts for changes in active datasets or physical components).

In the SAN file system example, by analyzing the invocation path (i.e. client machine, controller, disks), we determine that disks are saturated (i.e. the current I/O rate is the maximum they can support). Furthermore, a change analysis with a previous state reveals that client I/O request rate has increased by 40% and that the sequential/random ratio of the workload has changed from 0.1 to 0.7. Based on the problem determination analysis the following two queries get generated: Query 1: Select an action that improves the throughput of the disks by 25% (the fact that it is saturated will show-up in the preconditions of actions). Query 2: Select an action that improves the throughput of the (controller or client machine) by 25%, and is optimized for sequential workloads.

Base-level reasoning

Input: The set of queries ϕ derived by problem determination
Output: A set of candidate_actions that partially or completely satisfy elements in ϕ.
Approach:
The logic for searching the knowledge base is expressed in first-order predicate calculus. The logic captures the thought process that is implicit while writing imperative specifications. The information model of the action object makes it possible to express these semantics and derive the actions to be invoked "on-the-fly." A few examples of the thought process, expressed as first-order predicates, are described below.

At the high-level, a candidate action is one that affects the component in ϕ, satisfies the preconditions (p) in the current-state, and has the desired implications (i).

$$\forall\, a, s, \phi\; candidate(a, s, \phi) \;\Rightarrow\; component(a, \phi) \wedge p(a, s) \;\wedge\; i(a, s, \phi)$$

where: $\{\, a \,\epsilon\, Action,\;\; s \,\epsilon\, State\}$

Satisfy implications (i) is derived (as explained in Section 3.2.2) by combining the implication attribute present in the specifications ($Spec_I$) and the associated interpolation function ($interpolate$) derived by learning.

$$\forall\, a, s, \phi\; i(a, s, \phi) \;\Rightarrow\; \forall\, x\; Spec_I(a, x) \;\wedge\; (interpolate(x, s) \;>\; 0)$$

Similarly, satisfy preconditions (p) is defined as:

$$\forall\, a, s\; p(a, s) \;\Rightarrow\; \forall\, y\; Spec_P(a, y) \;\wedge\; \neg(Exclusion\_set(y) \,\epsilon\, Value(s))$$

At each step in the reasoning process, while selecting the candidate-actions, Polus maintains a log of the available choices and the option that was selected. This can later serve as an explanation to the human administrator, regarding how a specific action was selected.

In the example, assume the two candidate actions that are selected based on Query 1 and 2 are:

1. Invoking prefetching at the client machine
2. Invoking data replication at the disks

Meta-level operations

Input: A set of candidate actions, and ϕ generated by the problem determination module
Output: The actual set of actions to be invoked
Approach:
The aim is to select a candidate action, using an optimization function based on the following parameters:

- The transient overhead associated with the action invocation. For example, invoking replication has more overhead compared to invoking prefetching.

- The behavior side-effects of the action (i.e., an action, in addition to improving the violated goals, can possibly have a negative impact on other behavior dimensions). For example, invoking data backup improves availability, but it has a negative side-effect on throughput and latency.

- The confidence level associated with the details of the action object. As mentioned earlier, the confidence level is associated with the learning function error, measured as a difference between the predicted and the actual observed values.

An important requirement for this optimization is that it should not be based on short term goals (e.g., invoking replication has high overheads, but it might be beneficial in the long-run compared to invoking prefetching multiple times).

In Polus, the optimization algorithm is using n-step look ahead [20], which is extensively used in game theory. In n-step look ahead, the system simulates the impact of pursuing different options. The simulation is repeated n times, and the end result of the simulation is used to decide on the option to be selected in the current state. The simulation is based on the implication and precondition information of the action object. The n-step look ahead is just a rough estimate for the optimization, because external factors such as changes in workload or resources may render the predictions inaccurate. But it definitely helps in detecting instabilities arising due to cycles in state transitions (i.e. the system ping-pongs between two states, invoking the same set of actions repeatedly). It also helps in avoiding choices where a single action leads to a series of actions being invoked due to the cumulative side effects of actions.

Finally, in addition to considering the candidate actions independently, it is possible to reason with combinations of actions. For example, instead of considering the choices of invoking either prefetching or replication, it is possible to consider a combination of prefetching and replication as an additional candidate choice. Composite actions can be handled using vector arithmetic addition techniques but the description of these techniques is beyond the scope of this paper.

4 Experimental Setup

In the experimental setup, a SAN file system simulator is being used as the managed system. For the management software, we use an implementation of the Polus toolkit and compare it with its rule-based ECA counterpart. The Polus toolkit is built using ABLE (Agent Building and Learning Environment) [4]. ABLE provides the basic building blocks for Polus, namely learning algorithms such as neural networks, self-organizing map, JDBC connectivity for interfacing with the database, and data filters. The Polus modules are implemented as Java beans or agents. The implementation consist of agents for: Specifications (input), Reasoning, Learning, Sensors and Actuators. The rest of the section describes the implementation of the file system simulator.

The entities within the SAN file-system simulator are similar to those introduced in Section 2. The cost of atomic operations used in the simulator are shown in table 1. The simulator models the following actions that are invoked by the management software via actuators: Pre-fetch size tuning, Data replication, Backup and Clean-delay interval. I/O operations within the SAN file system are invoked by the client and can have multiple possible paths, depending on whether the data is cache or not. The simulator considers the following paths:

MHDH Metadata and data hit in the client

MHDM Metadata hit and data miss in client

MMDM Metadata and data miss in the client

The summation of each of these probabilities of the invocation paths $P_{MHDH} + P_{MHDM} + P_{MMDM} = 1$

The average I/O latency is given by:

$$L = P_{MHDH} * L_{MHDH} + P_{MHDM} * L_{MHDM} + P_{MMDM} * L_{MMDM}$$

where:

$$L_{MHDH} = L_m + L_d$$
$$L_{MHDM} = L_m + L_{DM}$$
$$L_{DM} = P_{Controller-hit} * Q_C * S_{Controller} + (1 - P_{Controller-hit}) * Q_C * S_{Disk}$$
$$L_{MMDM} = Queue\_depth_{Server} * S_{Server} + L_{DM}$$

Operation	Cost
Size of metadata object	1000 bytes
Latency to access from metadata cache (L_m)	20 μsec
Latency to read from datacache (L_d)	20 μsec
Service time of server (S_{server})	420 μsec
Service time from controller cache ($S_{controller}$)	0.6 $msec$
Service time of disk (S_{disk})	6 $msec$
Queue depth at controller (Q_C)	256 (max)
Size of metadata cache	128 MB
Size of data cache	1 GB
Size of controller cache	256 MB

Table 1. Cost of atomic operations in the file system simulator

The values for probabilities such as P_{MHDH}, and $P_{Controller-hit}$ are modeled by representing the caches as finite sized-arrays and keeping track of data blocks in the elements. Similarly, the average queue depth such as Q_C are actually modeled by using service time to complete each request.

Each action is modeled to reflect its impact on the invocation path, system resources and changes in the workload characteristics (table 2). To activate the actions in the file system simulator, specifications are fed into Polus and rule-based management. The specifications for a rule-based system consist of ECAs that describe the system-behavior for different system-states. The ECA specifications in this example run into 7 pages (76 rules). The exact Polus specifications that are fed are given in figure 8.

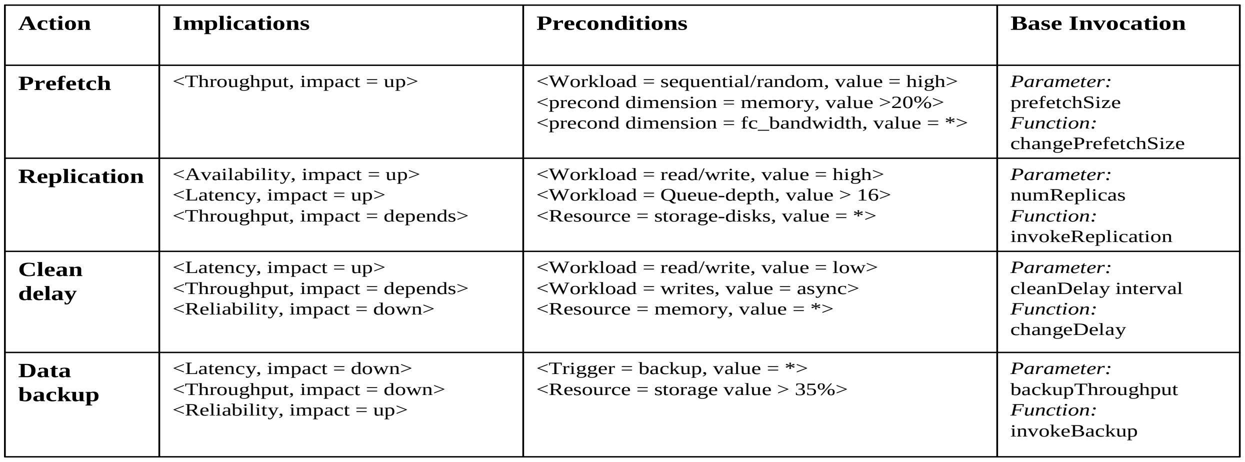
5 Experimental Analysis

The experimental analysis consists of a quantitative comparison of Polus and Rule-based systems for different system states. During evaluation, the values (thresholds and action invocation) for rule-based systems are assumed to be correct and empirically obtained from prior runs.

The file system is driven by a trace generator that imposes different states on the system. The generated

Action	Description	Invocation path parameters affected	Resources affected
Prefetching	Readahead of data and metadata	P_{MHDH} and P_{MHDM}	Data cache, metadata cache and interconnect bandwidth
Replication	Creates replica of data on a different volume in the controller	Q_C	Storage space (ignoring transient effects)
Data backup	Consider only transient effects since we are not considering availability	P_{MHDH}, P_{MHDM}, P_{MMDM} Q_C, Q_{Server}	Memory, interconnect bandwidth and storage space
Clean delay	Frequency at which dirty buffers are flushed to disks	Only for writes – P_{MHDH} bursty traffic for storage controller Q_C:	Metadata cache and data cache (metadata cannot be evicted till data is written)

Table 2. Modeling actions within the file system simulator



Action	Implications	Preconditions	Base Invocation
Prefetch	<Throughput, impact = up>	<Workload = sequential/random, value = high> <precond dimension = memory, value >20%> <precond dimension = fc_bandwidth, value = *>	*Parameter:* prefetchSize *Function:* changePrefetchSize
Replication	<Availability, impact = up> <Latency, impact = up> <Throughput, impact = depends>	<Workload = read/write, value = high> <Workload = Queue-depth, value > 16> <Resource = storage-disks, value = *>	*Parameter:* numReplicas *Function:* invokeReplication
Clean delay	<Latency, impact = up> <Throughput, impact = depends> <Reliability, impact = down>	<Workload = read/write, value = low> <Workload = writes, value = async> <Resource = memory, value = *>	*Parameter:* cleanDelay interval *Function:* changeDelay
Data backup	<Latency, impact = down> <Throughput, impact = down> <Reliability, impact = up>	<Trigger = backup, value = *> <Resource = storage value > 35%>	*Parameter:* backupThroughput *Function:* invokeBackup

Figure 8. Specifications fed to the Polus framework

state is a triplet of the form: <Workload characteristics, Available Resources, Goals >. The values for the goals are different than their current values. For each of these system states, we compare the response of both Polus and an ECA based rule system. Table 3 categorizes the possible system state.

The analysis of Polus and ECA for each of the categories is described as follows:

Category 1: Single action applicable

Analysis: The comparison is shown in figure 9. This is a simple category with a single candidate action. Polus generally selects the same action as an ECA-based system. ECA is assumed to have the right value for invocation while Polus uses the incremental approach for invocation. Learning improves the incremental approach by interpolating the starting point for incremental invocation.

Insights: The efficiency of the incremental algorithm is dependent on the impact function of the invoked action, which could be linear, quadratic, exponential and so on.

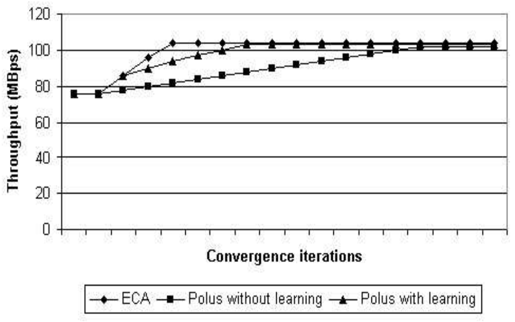


Figure 9. Comparing Polus and ECA for category 1 (single candidate action). In the graph, the throughput goal = 100 MBps.

Category 2: Multiple actions applicable

Analysis: This category (figure 10) exposes the "weak-spot" in Polus. When the candidate actions are indis-

Categories	Description	Example (File system simulator)
Category1: Single action applicable	The system states in this category are such that only a single candidate action is applicable, i.e., searching the specifications leads to a single candidate action	Workload: Sequential, read dominated with read/write ratio of 0.9, avg. queue-depth = 6 Current Throughput = 80 MBps Goal = 100 MBps Polus specification search: The only action that becomes applicable is Prefetching.
Category 2: Multiple actions applicable (they appear to have similar preconditions and/or implications)	In this category more than one action have similar preconditions and/or implications and are indistinguishable. In reality, these actions are not similar This category becomes increasingly common during initial bootstrapping, i.e., the system hasn't learnt values for preconditions and implications	Workload: Read dominated, sequential/random ratio = 0.2, average queue-depth = 8. Current Throughput = 80 MBps Goal = 100 MBps Polus specification search: Prefetching and Replication are selected as candidate actions. In reality Prefetching is not applicable as the workload is not sequential, but Polus does not have the threshold value for the sequential/random ratio in prefetch specifications
Category 3: More than one goal not met	In this category, more than one action needs to be invoked as a single action cannot satisfy the goal requirements	Workload: Sequential, read/write= 0.3 Current Throughput = 80 MBps Goal = 100 MBps , Current Latency = 6msec Goal = 4.5 msec Polus specification search: Prefetching and Clean delay are both invoked as the former improves throughput while the later improves latency
Category 4: Recurrent action invocation (One action, leads to chaininvocation of actions)	In this category, invocation of an action leads to a chain-invocation of a series of actions. Ability to detect and prevent recurrent action invocation is a required property of the management software	Workload: Trigger for data backup with window = 4 hours Polus specification search: Theoretically, backup can be invoked since the goals are being met. But invoking backup at this time will cause latency goals to be overshot
Category 5: No action applicable (Negation of previous actions required)	Actions are negated under two scenarios: to make resources available for another actions, and the workload preconditions change	Workload: Changes from large block sequential to small block random Polus specification search: Prefetching hurts performance as memory and storage resources are used for acquiring data that is never used

Table 3. Categorizing the possible system states

tinguishable, Polus tries them one-by-one till it either leads to a negative impact on the observable values or the goals are met. As shown in the graph, Polus initially selects the wrong action (i.e. prefetching). After the value dips further, Polus tries the next candidate action (i.e. replication). Learning adds the threshold values (in this example at the pre-conditions level) and enables distinguishing between the actions.

Insights: In systems with larger action-sets, it is quite possible that Polus never converges due to side-effects of trying wrong actions.

Category 3: More than one goal not met

Analysis: Rule-based systems will invoke a single action in each iteration without analyzing the combined impact of the actions. On the other hand, Polus considers different permutations to combine the actions (figure 11). This is beneficial when the two actions act on the same resources, such that the invocation of one action beyond a threshold can violate the pre-conditions of other actions. As shown in the graph, ECA does not meet the latency goal due to lack of memory resources. In its previous iteration Prefetching was invoked for throughput goals and the rules did not consider the combined state while deciding the value for prefetching. Learning refines the attributes of actions allowing better combination strategies.

Insights: Higher-order operation are powerful in deriving permutations that cannot be possibly defined statically.

Category 4: Recurrent action invocation

Analysis: The comparison in shown in figure 12. Polus uses look ahead while invoking actions to estimate the impact of the action on the goals. Rule-based systems don't have an equivalent of this (though it is possible

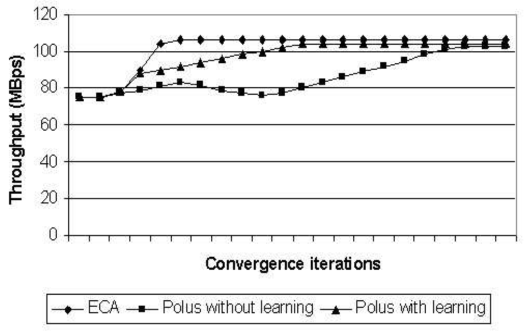


Figure 10. Comparing Polus and ECA for category 2 (multiple candidate actions). In the graph, the throughput goal = 100 MBps

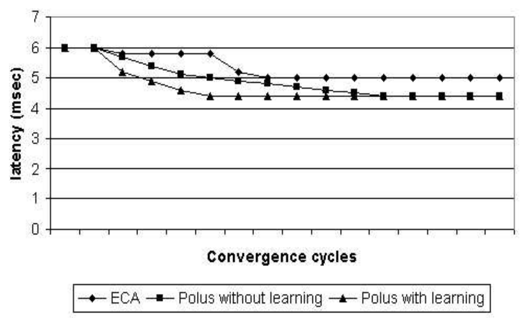


Figure 11. Comparing Polus and ECA for category 3 (more than one goal not met). In the graph, the latency goal = 4.5 msec

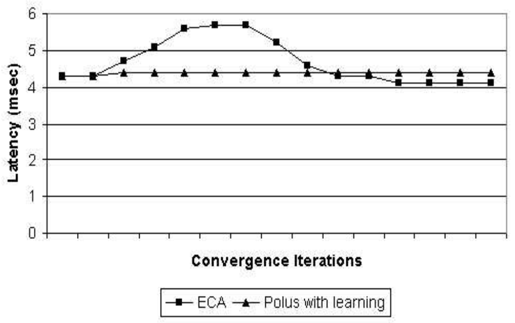


Figure 12. Comparing Polus and ECA for category 4 (recurrent action invocation). In the graph, the latency goal = 4.5 msec

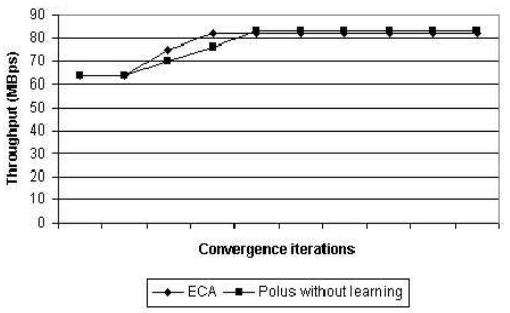


Figure 13. Comparing Polus and ECA for category 5 (negation of previous actions required). In the graph, the throughput goal = 80 MBps

to write separate rules to cater for this). As shown in the graph, ECA invokes the Backup action that leads to invocation of a series of actions (Replication in this example). Polus does a look-ahead and does not invoke Back-up during the current system-state. (For Back-up we are assuming a time-window)

Insights: Look-ahead is a required operation and effective only with some learning of the action model. Hence there is only a single curve in the graph.

Category 5: Negation of previous actions

Analysis: Both Polus and Rule-based systems can cater to negation of actions (figure 13). ECA can accommodate by writing separate rules. Learning does not play a significant role in this category

Insights: Explicit need to write separate rules in rule-based systems for negation whereas the Polus reasoning engine can account for this without any additional specifications.

6 Discussion

During the evaluation of the Polus framework, we made a few simplifying assumptions.

- The system state is assumed to remain constant during the time that an action is invoked. While it may be argued that this is an unrealistic assumption for actions that require significant time to complete, it is a reasonable one for most actions that involve small overhead.
- We do not consider the case where the specification provided by a administrator may be incomplete or contains incorrect information that may mislead the management software in its decision making.
- The experimental evaluation contains a single implementation of learning, reasoning and base invocation algorithms. These algorithms have not been fine tuned or optimized, and it is possible to plug-in more sophisticated algorithms for tree searching, incremental base invocation and learning. An evaluation by varying these algorithms is beyond the scope of this paper.

In real life deployments of Polus, system vendors can provide templates containing rules of thumb specifications and initial threshold values (obtained from training runs) for different workloads and system configurations. The adaptive behavior of the Polus framework will fine tune the knowledge base (i.e. threshold values) and tailor it to specific user environment and workload characteristics.

The evaluation framework presented in the paper is a system with just four possible actions. Hence, it is important to note that our aim was to understand the possible weaknesses of the Polus approach, and to get a first cut estimate of the number of iterations required in converging towards a specified QoS goal for different system states. As shown in the experiment section, the initial results show that this approach is promising; thus, we are currently implementing Polus as part of a real storage QoS management system. Polus can also be initially deployed as an aid to system administrators that allows them to perform what if analysis with respect to whether a system can support different QoS goals.

7 Related Work

Rome [25], Minerva [1], Hippodrome [2], and "attribute-managed storage" [13] projects from HP, SELF* project [9] from Carnegie Mellon, Storage Tank [21], SledRunner [6] projects from IBM, Control Centre product line from EMC, Storage Central product line from Veritas, and BrightStor product line from Computer Associates all aim to simplify storage management by automating different aspects of storage management. The Polus framework presented in this paper is complementary to these projects, since none of these projects specifically address the QoS goal transformation problem being addressed in this paper. Moreover, there is nothing inherent in the Polus framework that prevents its adoption by these different frameworks and products as part of their QoS solutions.

The Polus framework was built using specification, learning and reasoning techniques from the artificial intelligence (AI) domain. These technologies have a proven track record as they have been successfully used to build expert systems in medical, system configuration, video games and speech/handwriting processing application domains. To the best of our knowledge, Polus is the only system of its kind (in the domain of storage performance management) that integrates a rules-of-thumb specification model, reasoning (including higher-order operations) and a self-refining learning engine to manage a storage system.

Polus 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 [7], whereas, logic based systems overcome this problem by using a reasoning engine to combine facts/beliefs in the knowledge base to draw conclusions. The Polus specification of action attributes is similar to the declarative approach. Further, reasoning in Polus is a combination of specification search algorithms and higher-order operations. Polus 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 [17]. As explained earlier, Polus uses CBR as part of the reasoning engine to tie in the knowledge acquired by learning in the decision-making. Finally, learning in Polus systematically refines the specifications. It leverages research in the domain of machine learning algorithms such as neural networks and reinforcement learning [11, 15].

Currently, there are many competing policy specification standards [14, 19]. Polus can leverage any one these existing standards for specifying the base rules-of-thumb. Furthermore, there are no in-built dependencies that prevent Polus from leveraging the canonical SNIA SMI-S storage device standard [23] as the representation for low level system actions.

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 web-server configuration domain [24]. 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 zero (resource states, workload characteristics, goals, action set). In comparison, as shown in the experiment section, the combination of rule-of-thumb specification and a learning engine has a reasonable bootstrapping behavior. That is, the Polus approach is able to dynamically adapt even when it does not start from the most desired bootstrapped state. Mark, et al., [3] propose an approach to separate the goal from the base rule specification. They create a mapping between the rule and user-requirements, making it easy for validation and usage. The Polus approach is more sophisticated, in that it encodes the goal implications and uses them to automate the reasoning process.

Another approach [8] uses genetic algorithms for self-tuning. In this approach each system parameter is tuned by an individual algorithm and the genetic algorithm decides the best combination of algorithms. Unlike Polus, this approach does not allow refinement of the decision-making based on learning. Zinky, et al., [27] 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 to a QoS region. The QuO approach is static, as it does not implement semantics for reasoning about the various possible configurations.

8 Conclusion

Policy-based QoS management has been advocated as a "silver bullet" that can help to drastically increase the amount of storage that can be managed by system administrators. It is typically implemented using the ECA rules mechanism. However, as shown in this paper, the policy-based QoS management approach is not gaining much traction because of the associated difficulty in mapping high level QoS goals to low level system actions using the ECA approach.

In this paper we provide an alternative paradigm for mapping high level QoS goals to low level system actions. Our Polus approach leverages the proven AI techniques of learning and reasoning, and combines them with a declarative specification approach. Using this approach, system administrators can specify general rules of thumb to describe their knowledge instead of complex ECA rules containing detailed threshold values. In Polus, these threshold values are derived using learning algorithms. Furthermore, the use of reasoning engine allows Polus to automatically select the right action from amongst the various competing alternatives.

In conclusion, this paper presents a new approach towards how QoS goals can be mapped to low level system management actions. The aim of this approach is to reduce the number of inputs that are required from system administrators. We have analyzed the key concepts of this approach by using Polus to manage a simulated SAN file system and this is the first stepping stone towards the use of Polus-like approaches for managing real storage systems.

References

[1] G. 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 Trans. Comput. Syst.*, 19(4):483–518, Nov. 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. 1st Conf. on Filesystem and Storage Tech.*, pages 175–188, June 2002.

[3] M. Bearden, S. Garg, W. Lee, and A. van Moorsel. User-centric QoS policies, or saying what and how. Work-in-progress report, Proc. 11th Workshop on Distributed Systems: Operations and Management (DSOM), Dec. 2000.

[4] J. Bigus, D. Schlosnagle, J. Pilgrim, W. M. III, and Y. Diao. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Sys. J.*, 41(3), Sept. 2002.

[5] L. Breiman. Bagging predictors. *Mach. Learning*, 24(2):123–140, 1996.

[6] D. Chambliss, G. Alvarez, P. Pandey, R. M. D. Jadav, J. Xu, and T. Lee. Performance virtualization for large-scale storage systems. In *Proc. Symp. on Reliable Distributed Sys. (SRDS)*, Oct. 2003.

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

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

[9] G. Ganger, J. Strunk, and A. Klosterman. Self-* Storage: Brick-Based Storage with Automated Administration. Technical Report CMU-CS-03-178, Carnegie-Mellon University, Aug. 2003.

[10] M. Genesereth and M. Ginsberg. Logic Programming. *Comm. ACM*, 28(9), Sept. 1985.

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

[12] G. A. Gibson, D. F. Nagle, W. C. II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD Scalable Storage Systems. In *Proc. of Extreme Linux Workshop in the 1999 USENIX Ann. Tech. Conf.*, June 1999.

[13] R. Golding, E. Shriver, T. Sullivan, and J. Wilkes. Attribute-managed Storage. In *Proc. Workshop on Modeling and Specification of I/O*, Oct. 1995.

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

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

[16] R. Kowalski. Algorithm = logic + control. *Comm. ACM*, 22(7):424–436, 1979.

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

[18] J. Mogul. A better update policy. In *Proc. Summer 1994 USENIX Conf.*, pages 99–111, June 1994.

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

[20] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann Publishers Inc., 1992.

[21] D. Pease, J.Menon, B. Rees, L. Duyanovich, and B. Hillsber. IBM Storage Tank-A heterogeneous scalable SAN file system. *IBM Sys. J.*, 42(2):250–267, 2003.

[22] A. Singhal, G. Weiss, and J. P. Ros. A model-based reasoning approach to network monitoring. In *Proc. Workshop on Databases*, pages 41–44. ACM Press, 1997.

[23] Storage Networking Industry Association. SMI Specification version 1.0. http://www.snia.org, 2003.

[24] 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.

[25] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. *Lecture Notes in Computer Science 2092*, pages 75–91, 2001.

[26] J. Wilkes. Data Services - from data to containers. Keynote address, 2nd Conf. on Filesystems and Storage Tech. (FAST), Apr. 2003.

[27] 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.