Zodiac: Efficient Impact Analysis for Storage Area Networks

Aameek Singh
Georgia Institute of Technology

Madhukar Korupolu Kaladhar Voruganti
IBM Almaden Research Center

Abstract

Currently, the fields of impact analysis and policy based management are two important storage management topics that are not being treated in an integrated manner. Policy-based storage management is being adopted by most storage vendors because it lets system administrators specify high level policies and moves the complexity of enforcing these policies to the underlying management software. Similarly, proactive impact analysis is becoming an important aspect of storage management because system administrators want to assess the impact of making a change before actually making it. Impact analysis is increasingly becoming a complex task when one is dealing with a large number of devices and workloads. Adding the policy dimension to impact analysis (that is, what policies are being violated due to a particular action) makes this problem even more complex.

In this paper we describe a new framework and a set of optimization techniques that combine the fields of impact analysis and policy management. In this framework system administrators define policies for performance, interoperability, security, availability, and then proactively assess the impact of desired changes on both the system observables and policies. Additionally, the proposed optimizations help to reduce the amount of data and the number of policies that need to be evaluated. This improves the response time of impact analysis operations. Finally, we also propose a new policy classification scheme that classifies policies based on the algorithms that can be used to optimize their evaluation. Such a classification is useful in order to efficiently evaluate user-defined policies. We present an experimental study that quantitatively analyzes the framework and algorithms on real life storage area network policies. The algorithms presented in this paper can be leveraged by existing impact analysis and policy engine tools.

1 Introduction

The size and scale of the storage infrastructure of most organizations is increasing at a very rapid rate. Organizations are digitizing and persistently storing more types of data, and are also keeping old data longer, for compliance and business intelligence mining purposes. The number of system administrators required to manage storage also increases as a function of the growth in storage because there is a limit to the amount of storage that can be managed by a single administrator. This limit is due to the number of complex tasks that a system administrator has to perform such as change analysis, provisioning, performance bottleneck analysis, capacity planning, disaster recovery planning and security analysis.

The focus of this paper is on one of these important problems namely *change analysis*. This paper provides a framework and a set of algorithms that help system administrators to proactively assess the impact of making changes in a storage area network (SAN) before making the actual change. Currently, administrators perform impact analysis manually, based on their past experience and rules of thumbs (best practices). For example, when a new host is added, the administrators have to make sure that Windows and Linux hosts are not put into the same zone or while adding a new workload, they have to ensure that the intermediate switches do not get saturated.

Manually analyzing the impact of a particular change does not scale well as the size of the SAN infrastructure increases with respect to the number of devices, best practices policies, and number of applications. Thus, deployment of new applications, hosts and storage controllers takes in the order of days or weeks because system administrators deploy the system and then reactively try to correct the problems associated with the deployment. Typically change management tools have been very reactive in their scope in that they keep snapshots of the previous state of the system, and the administrators either revert to or compare the current state with a previous state after encountering a problem.

Additionally, administrators do not have a way of assessing the impact of their proposed change with respect to a future state of the system. For example, a system administrator could potentially allocate increased band-

width to an application by taking only the current load into account. However, this could conflict with other scheduled jobs or known trends in workload surges that will increase the load on the system in the future. Thus, it is important for system administrators to assess the impact of their action not just with respect to the current state of the system but also with respect to future events.

1.1 Contributions

In order to address the above described problems, we present the **Zodiac** framework. The Zodiac framework enables system administrators to proactively assess the impact of their actions on a variety of system parameters like resource utilizations and existing system policies, before making those changes. Proactive change management analysis is an important problem and is currently receiving the deserved attention [26, 21, 23, 29]. Through Zodiac, we make the following contributions:

1. Integration with Policy based Management: The key aspect of our analysis framework is that it is tightly integrated with policy based storage management. Currently, policy-based management is being incorporated into most vendor's storage management solutions. Best-practices, service class goals, interoperability constraints, are specified as policies in the system. Thus, in essence we are combining the areas of impact analysis and policy based-management. Zodiac allows administrators to specify their rules of thumb or best practices with respect to interoperability, performance, availability, security as *policies*. It then assesses the impact of user actions by checking which of these policies are being violated or triggered. Zodiac also assesses the impact of creating new policies.

2. Scalability and Efficiency: Most system administrators want to assess the impact of their changes in real-time. A quick feedback on a proposed change encourages a system administrator to try out many alternatives. The three major components that contribute towards the execution time of impact analysis processing are: a) number of policies b) size of the storage infra-structure c) analysis time window (that is assess the impact of an action for a time window of a day, a week or a month). An impact analysis engine should be able to scale upto big SANs with 1000 hosts (found in many of today's data centers) and a few hundred policies. In this paper, we provide algorithms and data structures that help to reduce the amount of SAN data that is examined during impact analysis, the number of policies that need to be evaluated, and a framework for performing temporal impact analysis.

3. Classification Framework: One of the interesting result of the algorithm design effort in Zodiac is that we have designed a new method for classifying SAN policies based on the optimization techniques they employ. This, in turn, can also be used by general SAN policy evaluation engines to optimize their evaluation mechanisms. During policy specification period, policy designers can specify the policy type (as per this classification) as a hint to the policy engine to optimize its evaluation.

The rest of the paper is organized as follows. Section-2 provides the necessary background with respect to policy definitions, and SAN operations. Related work is presented in Section-3. The overview of our architecture is presented in Section-4 followed by the details of our implementation in Section-5. In Section-6, we discuss three important optimization algorithms that help speed up the overall analysis process. The experimental framework and the results evaluating these optimizations are presented in Section-7. We discuss related optimizations in Section-8. Finally, we conclude in Section-9.

2 Background

This section presents the necessary background material for this paper. Section-2.1 contains a discussion on the type of SAN policies considered in this paper. Section-2.2 provides the details of the storage resource models and Section-2.3 presents a list of what-if operations that one can perform. In summary, one can define various policies on the SAN resources and using our framework, analyze the impact of certain operations on both the SAN and its associated policies.

2.1 Policy Background

The term *policy* is often used by different people in different contexts to mean different things. For example, the terms *best practices, rule of thumbs, constraints, threshold violations, goals, rules and service classes* have been referred to as policies by different people. Currently, most standardization bodies such as IETF, DMTF, and SNIA refer to policy as a 4-field tuple where the fields correspond to an *if* condition, a *then* clause, a *priority or business value* of the policy and a *scope* that decides when the policy should be executed. The *then* clause portion can generate indications, or trigger the execution of other operation (action policies), or it can simply be informative in nature (write a message to the log). [1] describes the various SAN policies found relevant by domain experts. In this paper, within the storage area network (SAN) domain, we deal with the following types of policies:

- **Interoperability:** These policies describe what devices are interoperable (or not) with each other.
- **Performance:** These policies are violation policies that notify users if the performance of their applications (throughput, IOPs or latency) violate certain threshold values.
- **Capacity:** These policies notify users if they are crossing a percentage (threshold) of the storage space has been allotted to them.
- **Security and Access Control:** Zoning and LUN masking policies are the most common SAN access control policies. Zoning determines a set of ports that can transfer data to each other in the set. Similarly, LUN masking controls host access (via its ports) to storage volumes at the storage controller.
- **Availability:** These policies control the number of redundant paths from the host to the storage array.
- **Backup/Recovery:** These policies specify the recovery time recovery point, recovery distance, copy size and copy frequency to facilitate continuous copy and point-in-time copy solutions.

2.2 Storage Resource Model

In order to perform impact analysis, storage resource models are used to model the underlying storage infrastructure. A storage resource model consists of a schema corresponding to various **entities** like hosts, host bus adapters (HBAs), switches, controllers, the entity **attributes** (e.g. vendor, firmware level), container relationships between the entities (HBA is contained with a host), and connectivity between the entities (fabric design). These entities and attributes are used during the definition of a policy as part of the *if-condition* and the *then clause*. For a specific policy, the entities and the attributes that it uses are called its *dependent* entities and *dependent* attributes respectively. The SNIA SMI-S [28] model presents a general framework for naming and modeling storage resources.

In addition to the schema, a storage resource model also captures the behavioral aspects of the entities. The behavioral aspects, called **metrics**, represent how a resource behaves under different workload and configuration conditions. The behavioral models are either analytically specified by a domain expert [30], or deduced by observing a live system [3] or a combination of both.

Figure-1 shows the basic SAN resource model that we consider in this paper. Our resource model consists of hosts, HBAs, ports, switches, storage controllers, zones, and volume entities, and host to HBA, port to HBA, port to zone containment relationships and port to port connection relationships. In addition, there exists a *parent entity class* called $device$, which contains all the SAN devices. The $device$ entity class can be used to define global policies like *all devices should have unique WWNs*. Please note that our framework and techniques are not limited to this model only but instead can also be used in more generalized storage infrastructure models.

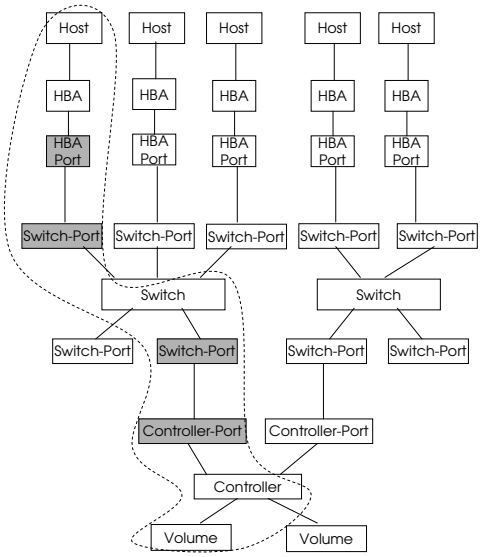

Figure 1: SAN Resource Model. A single SAN path is highlighted. Shaded ports represent zone containment.

2.3 SAN Operations:

Using Zodiac, the following types of operations can be analyzed for impact.

- *Addition/Deletion of Physical Resources* like hosts, switches, HBAs, and storage arrays.
- *Addition/Deletion of Logical Resources* like volumes and zones.
- *Access control operations* by adding or removing ports to zones or similar LUN masking operations.
- *Addition/Deletion of Workloads*. Also, the requirements (throughput, latency) of a workload can be modified. We represent a workload as a set of flows. A flow can be thought of a data path (Figure-1) between a host and a storage controller. A flow starts at a host port, and goes through intermediate switch ports and ends at a storage controller port.
- *Addition/Deletion of Policies*. Please note that we do not focus on conflict detection analysis within policies in this paper.

3 Related Work

With the growth in amount of storage resources, there has been a strong initiative for automating various manage-

ment tasks and making systems self-sufficient [2, 4, 18, 10]. Most of this research has focused on various planning tasks - capacity planning, including Minerva [2], Hippodrome [4], Ergastulum [5]; fabric planning like Appia [32, 33], and disaster recovery planning [22, 23].

Impact analysis, also referred to as "what-if" or change-management analysis, is another closely related management task. Some of the planning work described above can actually be used for such analysis. For example, Ergastulum [5] can be used to analyze storage subsystems and Keeton et al's [23] helps in analyzing disaster recovery scenarios. Another recent work by Thereska et al. [29] provides what-if analysis using the Self-* framework [18]. There also exist tools and simulator like [21, 34] that provide impact analysis for storage controllers. Most of the what-if approaches utilize device and behavioral models for resources. Significant amount of research has been done both in developing such models [30, 3, 27, 31, 34] and using those models [14, 11, 25, 8].

Zodiac is different from existing impact analysis work, due to its close integration with policy based management. Using Zodiac, an administrator can analyze the impact of operations not only on system resources but also on system policies. In addition, the analysis accounts for all subsequent actions triggered by policy executions. As we describe later, efficient analysis of policies is non-trivial and critical for overall performance.

The Onaro SANscreen product [26] provides a similar predictive change management functionality. However, from the scarce amount of published information, we believe that they only analyze the impact for a small set of policies (mainly security) and do not consider any triggered policy actions. We believe this to be an important shortcoming, since typically administrators would specify policy actions in order to correct erroneous events and would be most interested in analyzing the impact of those triggered actions. The EMC SAN Advisor [16] tool provides support for policy evaluations, but is not an impact analysis tool. Secondly, it pre-packages its policies and does not allow specification of custom policies.

In the policies domain, there has been work in the areas of policy specification [12, 7], conflict detection [17] and resource management [24]. The SNIA-SMI [28] is also developing a policy specification model for SANs. To the best of our knowledge, there does not exist any SAN impact analysis framework for policies. [1] proposed a policy based validation framework, which is typically used as a periodic configuration checker and is not suitable for interactive impact analysis.

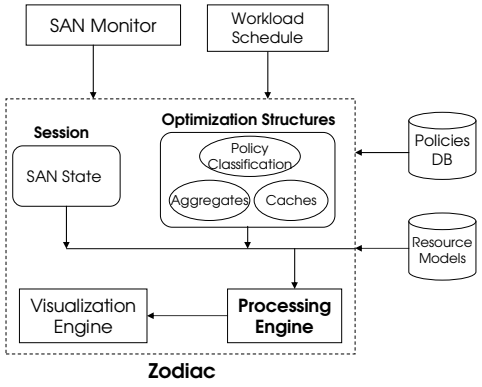

Figure 2: Architecture

4 Architecture Overview

In this section, we provide an overview of the Zodiac architecture and its interaction with other SAN modules.

4.1 Zodiac: Big Picture

The goal of the impact analysis engine, like Zodiac, is to predict the state and behavior of the SAN once a desired operation is performed. In order to evaluate the new state, the engine needs to interact with various SAN modules to get the relevant information, like device attributes, policies. The overall picture of such an eco-system is shown in Figure-2. In this eco-system, Zodiac interacts with the following modules:

- *SAN Monitor:* The foremost input requirement is the state of the SAN, which is obtained from a SAN Monitor like [15, 19, 20, 6]. It consists of the physical configuration (fabric design), resource attributes (HBA vendor, number of HBAs in a host) and logical information like Zoning/LUN-Masking.
- *Workload Schedule:* In order to predict the behavior of the SAN, Zodiac also needs to know the schedule of the workload. For example, if a backup job is scheduled for 3 AM, then the engine needs to account for the additional traffic generated due to the backup during that duration.
- *Policy Database:* A unique characteristic of the Zodiac impact analysis engine is its integration with policy based management. The policies are specified in a high level specification language like Ponder [12] or XML [1] and stored in a policy database.
- *Resource Models*: As described earlier, Zodiac uses a model based approach to evaluate the behavior of SAN resources. For this, we require a resource models database that provides such behavioral models. There has been significant work in

the area of modeling and simulation of SAN resources [30, 3, 27, 31, 11, 25, 8, 9, 34] and we leverage that. Note that the design of Zodiac is independent of the resource models and can work with any approach.

Given these modules, Zodiac takes as input the operation that the administrator wants to perform and the time at which the impact needs to be measured (immediate or after n hours) and initiates the analysis.

4.2 Zodiac: Internal Design

Internally, Zodiac engine is composed of the following primary components:

- **SAN-State:** In Zodiac, the impact analysis occurs in a *session*, during which an administrator can analyze the impact of multiple operations incrementally. So, a first operation could be - *what happens if I add two hosts?* After the engine evaluates the impact, an administrator can perform an incremental operation - *what if I add* another *two hosts?* The SAN state component maintains the intermediate states of the SAN, so that such incremental operations can be analyzed. When an analysis session is initialized, the SAN state is populated by the current snapshot of the SAN, obtained from the SAN Monitor.
- **Optimization Structures:** As mentioned earlier, for efficient policy evaluation, Zodiac maintains intelligent data structures that optimize the overall evaluation. These three primary structures (caches, policy classes and aggregates) are explained in detail in Section-6.
- **Processing Engine:** The processing engine is responsible for efficiently evaluating the impact of operations using the SAN state and the rest of the internal data structures. It is the main work horse of Zodiac.
- **Visualization Engine:** Another important component of Zodiac is its visualization engine. The visualization engine primarily provide two kinds of output. First, it can provide an overall picture of the SAN, with various entity metrics and can highlight interesting entities, e.g. the ones that violated certain policies. Secondly, with the incorporation of temporal analysis, an administrator can plot interesting metrics with time.

5 Zodiac: System Details

In this section, we provide the details about the internal data structures being used by Zodiac to represent SANs (Section-5.1) and how the policy evaluation framework uses these data structures (Section-5.2). In Section-5.3, we describe the inadequacy of the current evaluation approach before proposing various optimizations in the next section.

5.1 SAN Representation

For efficient impact analysis, it is critical that SAN is represented in an optimal form. This is because all policies and resource metric computations would obtain required data through this SAN data structure. In Zodiac, the SAN is represented as a graph with entities as nodes and network links or containment relationships (HBA is contained within a host) as edges. A sample SAN as a graph is shown in Figure-1. A single SAN "path" has been highlighted. Note that it is possible to have more than one switch in the path.

Each entity in the graph has a number of attribute-value pairs, e.g. the host entity has attributes like vendor, model and OS. In addition, each entity contains pointers to its immediate neighbors (Host has a pointer to its HBA, which has a pointer to its HBA-Port and so on). This immediate neighbor maintenance and extensive use of pointers with zero duplication of data allows this graph to be maintained in memory even for huge SANs (1000 hosts).

There are two possible alternatives to this kind of immediate-neighbor representation of the SAN. We discuss the alternatives and justify our choice below:

1. **Alternative-Paths**: Assume a best practices policy requiring a Vendor-A host to be only connected to Vendor-S controller. Its evaluation would require a traversal of the graph starting at the host and going through all paths to all connected controllers. In fact many policies actually require traversals along "paths" in the graph[1]. This could indicate storing the SAN as a collection of paths and iterating over the relevant paths for each policy evaluation, preventing costly traversals over the graph. However, the number of paths in a big SAN could be enormous, and thus, prohibitive to maintain the path information in-memory. Also, the number of new paths created with an addition of a single entity (e.g. a switch) would be exponential, thus making the design unscalable.
2. **Alternative-SC**: Even without paths, it is possible to "short-circuit" the traversals by keeping information about entities further into the graph. For example, a host could also keep pointers to all connected storage. While this scheme does work for some policies, many interoperability policies, that filter paths of the graph based on some properties of

an intermediate entity, cannot be evaluated. For example, a policy that requires a Vendor-A host, *connected to a Vendor-W switch*, to be only connected to Vendor-S storage, cannot be evaluated efficiently using such a representation, since it is still required to traverse to the intermediate entity and filter based on it. However, this idea is useful and we actually use a modified version of this in our optimization schemes described later.

5.2 Policy Evaluation

In current policy evaluation engines, policies are specified in a high level specification language like Ponder [12], XML [1]. The engine converts the policy into executable code that can evaluate the policy when triggered. This uses an underlying data layer, e.g. based on SMI-S, that obtains the required data for evaluation. It is this automatic code generation, that needs to be heavily optimized for efficient impact analysis and we discuss various optimizations in Section-6.

In Zodiac, the data is obtained through our SAN data structure. For evaluating a policy like *all Vendor-A hosts should be connected to Vendor-S controllers*, a graph traversal is required (obtaining storage controllers connected to Vendor-A hosts). In order to do such traversals, each entity in the graph supports an API that is used to get to *any* other connected entity in the graph (by doing recursive calls to immediate neighbors). For example, hosts support a $getController()$ function that returns all connected storage controllers. The functions are implemented by looking up immediate neighbors (HBAs), calling their respective $getController()$ functions, aggregating the results and removing duplicates. The neighbors would recursively do the same with their immediate neighbors until the call reaches the desired entity (storage controller). Similarly for getting all connected edge switches, core switches or volumes. This API is also useful for our caching optimization. It caches the results of these function calls at all intermediate nodes for reuse in later policy evaluations.

However, even this API suffers from the limitation of the Alternative-SC scheme presented above. That is, how to obtain controllers connected only through a particular vendor switch. To facilitate this, the entity API allows for passing of filters that can be applied at intermediate nodes in the path. For our example, the filter would be *Switch.Vendor="W"*. Now, the host would call the HBA's $getController()$ function with the filter *Switch.Vendor="W"*. When this call recursively reaches the switch, it would check if it satisfies the filter and only the switches that do, continue the recursion to their neighbors. Those that do not satisfy the filter return null.

The use of filters prevents unnecessary traversals on the paths that do not yield any results (e.g. paths to the controllers connected through switches from other vendors). The filters support many comparison operations like $=, \neq, >, \geq, <, \leq, \in$ and logical OR, AND and NOT on filters are also supported. The caching scheme incorporates filters as well (Section-6.2). The Alternative-SC presented above, can not use this filter based scheme since the possible number of filters can be enormous and thus always storing information in-memory for each such filter would be infeasible.

Notice that not all filters provide traversal optimizations. The filters that are at the "edge" of a path do not help. For example, a best practices policy - *if a Vendor-A host connected to a Vendor-W switch accesses storage from a Vendor-S controller, then the controller should have a firmware level $> x$*. In this case, the policy requires getting controllers with the filter *Switch.Vendor="W" AND Controller.Vendor="S"*. While the first term helps reduce the number of paths traversed, the second term does not – we still have to check every controller connected through the appropriate switches. Therefore, we prefer not to apply the filters at the edge, instead obtaining all edge entities (controllers in this case) and then checking for all conditions ($Vendor$ and $FirmwareLevel$). This helps in bringing more useful data into the entity caches.

It is also important to mention that the traversal of the graph can also be done only for logical connections (due to zoning). This is facilitated by providing equivalent API functions for traversing links with end-points in particular zone, e.g. $getControllerLogical(Z)$ obtains all connected controllers in Zone Z, i.e. all controllers reachable through a path containing ports (HBA ports, switch ports, controller ports) in zone Z.

Given the above framework, we next discuss why the current policy evaluation approach is inefficient for impact analysis.

5.3 Impact Analysis: Inadequacies of Current Approach

During impact analysis, a SAN operation can trigger multiple policies to be evaluated. For example, a host being added into the SAN would require evaluation of intrinsic host policies (policies on basic attributes of the host - *all hosts should be from a single vendor*), various host interoperability policies with other connected devices, zoning policies, and so on. With the popular policy-based autonomic computing initiative, it is highly likely that the number of policies in a SAN would be very large. So it is imperative that only the relevant set of policies are evaluated. For example, for the host-addition case, a switch and controller interoperability policy should not be evaluated.

The current policy evaluation engines [1] use a coarse classification of **scopes**. In such a scheme, each policy is designated a $Scope$ to denote the class of entities, it is relevant to. In addition, it is possible to sub-scope the policy as *intra-entity* - evaluation on attributes of a single entity class or *inter-entity* - evaluation on attributes of more than one entity class [1]. The motivation for such classification is to allow administrators, to do a policy check only for a select class of entities and policies in the SAN. Unfortunately, this form of classification is not efficient for impact-analysis due to the following reasons:

- **Lack of granularity**: Consider the policy which requires a Vendor-A host to be connected only to Vendor-S storage controller. Naively, such a policy would be classified into the Host scope and the Storage scope. Thus, whenever a new host is added to the SAN, it will be evaluated and similarly, when a controller is added. However, consider the addition of a new link between an edge and a core switch. Such a link could cause hosts to be connected to new storage controllers, and thus the policy would still need to be evaluated and so, the policy also needs to be added to the Switch scope. With only scope as the classification criteria, *any* switch related event would trigger this policy. It is possible to further $sub\text{-}scope$ the policy to be an *inter-entity*. However, it still will be clubbed with other switch inter-entity policies, which will cause un-necessary evaluations.
- **Failure to identify relevant SAN region**: The current scoping mechanism fails to identify the region of the SAN that needs to be traversed for policy evaluation. Consider the two policies: (a) *All Vendor-A hosts should be connected to Vendor-S storage*, and (b) *All hosts should have atleast one and atmost four disjoint paths to controllers*. Both the policies would have identical scopes (host, controller and switch) and sub-scopes (inter-entity). Now, when a switch-controller link is added to the SAN, evaluation of (a) should traverse *only* the newly created paths – ensure that all new host-storage connections satisfy the policy; there is no need to traverse a path that has already been found to satisfy that policy. However, the same is not true for (b). Its evaluation would require traversing many old paths. The current scoping mechanism fails to identify policies of type (a) and would end up evaluating many old paths in order to provide a correct and general solution.

The current policy evaluation engines also **fail to exploit the locality of data** across various policies. For example, two distinct policies might require obtaining the storage controllers connected to the same host. In such a scenario, it is best to obtain the results for one and cache them to use it for the other. To the best of our knowledge, the current policy engines do not provide such caching schemes and rely on the underlying SMI-S data layer [1] to do this caching (which could still require evaluating expensive $join$ operations). This is, in part, due to the low number of policies in current SANs and the fact that currently, the policy checking process is primarily a non-real-time, scheduled task with periodic reporting (typically daily or weekly reporting periods). As we show in Section-7, a caching scheme could have drastic performance benefits and help in interactive real-time analysis.

Such an efficiency is critical especially in the presence of **action policies**. Such policies when triggered initiate automatic operations on the SAN ("then" clause of the policy). These are typically designed as compensating actions for certain events and can do rezoning, introduce new workloads, change current workload characteristics, schedule workloads and more. For example, a policy for a write-only workload like *if Controller-A utilization increases beyond 95%, write the rest of the data on Controller-B*. Thus, when the policy is triggered, a new flow is created between the host writing the data and Controller-B, and policies related to that event need to be checked. The action might also do rezoning to put Controller-B ports in the same zone as the host and so, all zoning related policies would end up being evaluated. Overall, such a chain of events can lead to multiple executions of many policies. The caching scheme, combined with the policy classification, significantly helps in these scenarios.

6 Zodiac Impact Analysis: Optimizations

In this section, we present various optimizations in the Zodiac architecture that are critical for the scalability and efficiency of impact analysis. Zodiac uses optimizations along three dimensions.

1. *Relevant Evaluation*: Finding relevant policies and relevant regions of the SAN affected by the operation. This is accomplished using policy classification and is described in Section-6.1.
2. *Commonality of Data Accessed*: Exploiting data locality across policies or across evaluation for different entity instances. This is achieved by using caching, described in Section-6.2.
3. *Aggregation*: Efficient evaluation of certain classes of policies by keeping certain aggregate data structures. This scheme is described in Section-6.3.

All three optimizations are independent of each other and can be used individually. However, as we show later

in our results, the best performance is achieved by the combination of all three optimizations.

6.1 Policy Classification

The first policy evaluation optimization in Zodiac is policy classification. Policy classification helps in identifying the relevant regions of the SAN and the relevant policies, whenever an operation is performed. In order to identify the relevant SAN region affected by an operation, we classify the policies into four categories described below. Only the "if" condition of the policy is used for classification. Also, each policy class has a set of operations, which are the ones that can trigger the policy. This mapping of operations to policies can be made easily due to our classification scheme and is used to find the relevant set of policies.

1. **Entity-Class (EC) Policies**: These policies are defined only on the instances of a single entity class. For example, *all HBAs should be from the same vendor*, and *all Vendor-W switches must have a firmware level* $> x$. Such policies do not require any graph traversals, rather a scan of the list of instances of the entity class. The relevant operations for this class of policies are addition/deletion of an entity-instance or modification of a "dependent" attribute of an instance like changing the firmware level of a switch (for our second example above). Additionally, EC policies can be subdivided into two types:

   – *Individual (EC-Ind) Policy*: A policy that holds on every instance of the entity class. For example, *all switches must be from Vendor-W*. This class of policies has the characteristic that whenever an instance of the entity class is added/modified, the policy only needs to be evaluated on the new member.

   – *Collection (EC-Col) Policy*: A policy that holds on a collection of instances of the entity class. For example, *the number of ports of type X in the fabric is less than N* and also *all HBAs should be from the same vendor*[1]. This class of policies might require checking all instances for final evaluation.

2. **Single-Path (SPTH) Policies**: These policies are defined on more than one entity on a single path of the SAN. For example, *all Vendor-A hosts must be connected to Vendor-S storage*. Importantly, SPTH policies have the characteristic that the policy is required to hold on *each* path. In our example, each and every path between hosts and storages must satisfy this policy. This characteristic implies that on application of **any** operation, there is no need to evaluate this policy on old paths. Only new paths need to be checked. The relevant operations for these policies are addition/deletion/modification of paths or modification of a "dependent" attribute (vendor name) of a "dependent" entity (storage controller) on the path.

3. **Multiple-Paths (MPTH) Policies**: These policies are defined across multiple paths of the SAN. For example, *all hosts should have atleast two and atmost four disjoint paths to storage*, and *a Vendor-A host should be connected to atmost five controllers*. MPTH policies cannot be decomposed to hold on individual paths for *every* operation. For the examples, adding a host requires checking only for the new paths created, whereas adding a switch-controller link requires checks on earlier paths as well. We are working on developing a notion distinguishing between the two cases[2]. In this paper, we consider MPTH policy as affecting all paths. The relevant operations for these policies are addition/deletion/modification of paths or modification of a "dependent" attribute of a "dependent" entity on the path.

4. **Zoning/LUN-Masking (ZL) Policies**: These policies are defined on zones or LUN-Mask sets of the SAN. For example, *a zone should have atmost N ports*, and *a zone should not have both windows or linux hosts*. For our discussion, we only use zone policies, though the same approach can be used for LUN-Masking policies. Notice that these policies are similar to EC policies with entity-class being analogously replaced by zones or LUN-Mask sets. Just as EC policies are defined on attributes of entity instances, ZL policies are defined on attributes of zone instances. Also similar to EC policies, Zone policies can be collection policies, requiring evaluation over multiple zones, (e.g. *the number of zones in the fabric should be atmost N*)[3] and individual policies, requiring evaluation only over an added/modified zone (e.g. *all hosts in the zone must be from the same vendor*). Also, within a zone, a policy might require evaluation over only the added/modified component ($Zone\text{-}Member\text{-}Ind$) or all components ($Zone\text{-}Member\text{-}Col$). An example of a $Zone\text{-}Member\text{-}Ind$ policy is *all hosts in the zone should be windows*, and an example of $Zone\text{-}Member\text{-}Col$ policy is *a zone should have atmost N ports*. The relevant operations for this class of policies are addition/deletion of a zone instance or modification of an instance (addition/deletion of ports in the zone).

Note that the aim of this classification is not to semantically classify all conceivable policies, but rather to identify the policies that can be optimized for evaluation. Having said that, using our classification scheme, it was indeed possible to classify all policies mentioned in [1], the only public set of SAN policies collected from administrators and domain experts. The basic difference

between the classification scheme in [1] and our scheme stems from the fact that it classifies policies based on specification criteria, while we use the internal execution criteria for the classification. This helps us in generating optimized evaluation code by checking only the relevant regions of the SAN.

6.2 Caching

The second optimization we propose, uses a caching scheme to cache relevant data at *all* nodes of the SAN resource graph. Such a scheme is extremely useful in an impact-analysis framework due to the commonality of data accessed in the following scenarios:

1. *Multiple executions of a single policy*: A single policy might be executed multiple times on the same entity instance due to the chaining of actions, defined in the *then* clause of the triggered policies. Any previous evaluation data can be easily reused.

2. *Execution of a single policy for different instances of entities*: For example, consider an operation of adding a policy like *all Vendor-A hosts should be connected to Vendor-S storage*. For impact analysis, the policy needs to be evaluated for all hosts. In our immediate-neighbor scheme, for the evaluation of this policy, a host, say Host-H, would call its HBA's $getController()$ function, which in turn would call its ports' $getController()$ function, which would call the edge switch (say Switch-L) and so on. Now, when any other host connected to Switch-L calls its $getController()$ function, it can reuse the data obtained during the previous evaluation for Host-H. Note that with no replacement, the caching implies that traversal of any edge during a policy evaluation for all entity instances is done *atmost* once. This is due to the fact that after traversing an edge {u,v} once, the required data from v would be available in the cache at u, thus preventing its repeated traversal.

3. *Locality of data required across multiple policies*: It is also possible, and often the case, that multiple policies require accessing different attributes of the same entity. As mentioned earlier, we do not apply filters to the "edge" entities (e.g. controllers for a $getController()$ call) and retrieve the full list of entities. Now, this cached entry can be used by multiple policies, even when their "dependent" attributes are different.

As mentioned earlier, the caching scheme incorporates filters as well. Whenever an API function is called with a filter, the entity saves the filter along with the results of the function call and a **cache hit** at an entity occurs only when there is a complete match, i.e. the cached entry has the same API function call as the new request and the associated filters are also the same. This condition can be relaxed by allowing a partial match, in which the cached entry is for the same function call, but can have a more general filter. For example, assume a cache entry for $getController()$ with the filter *Switch.Vendor="W"*. Now, if the new request requires controllers with the filter *Switch.Vendor="W" AND Switch.FirmwareLevel* $>$ *x*, the result can be computed from the cached data itself. We leave this for future work. Also, the current caching scheme uses LRU for replacement.

6.3 Aggregation

It is also possible to improve the efficiency of policy execution by keeping certain aggregate data structures. For example, consider a policy which mandates that *the number of ports in a zone must be atleast M and atmost N*. With every addition/deletion of a port in the zone, this policy needs to be evaluated. However, each evaluation would require counting the number of ports in the zone. Imagine keeping an aggregate data structure that keeps the number of ports in every zone. Now, whenever a port is added/deleted, the policy evaluation reduces to a single check of the current count value.

We have identified the following three classes of policies that can simple aggregate data structures:

1. *Unique*: This class of policies require a certain attribute of entities to be unique. For example, policies like *the WWNs of all devices should be unique*, *all Fibre Channel switches must have unique domain IDs*. For these class of policies, a hashtable is generated on the attribute and whenever an operation triggers this policy, the policy is evaluated by looking up that hashtable. This aggregate data structure can provide good performance improvements especially in big SANs (Section-7). Note that such an aggregate is kept only for EC and ZL policies (where it is easier to identify addition/deletion). However, there does not appear to be any realistic SPTH or MPTH unique policies.

2. *Counts*: These policies require counting a certain attribute of an entity. Keeping the count of the attribute prevents repeated counting whenever the policy is required to be evaluated. Instead, the count aggregate is incremented/decremented when the entity is added/deleted. A count aggregate is used only for EC and ZL policies. While SPTH and MPTH count policies do exist (e.g. *there must be atmost N hops between host and storage* and *there must be atleast one and atmost four disjoint paths between host and storage* respectively), maintaining the counts is tricky and we do not use an aggregate.

3. *Transformable*: It is easy to see that the policy evaluation complexity is roughly of the order $EC\text{-}Ind \approx Zone\text{-}Member\text{-}Ind < EC\text{-}Col \approx Zone\text{-}Member\text{-}Col < SPTH < MPTH$. It is actually possible to transform many policies into a lower complexity policy by keeping additional information about some of the dependent entities. For example, consider a policy like *all storage should be from the same vendor*. This policy is an EC-Col for entity class - Storage. However, keeping information about the current type of storage (T) in the system, the policy can be reduced to an equivalent EC-Ind policy – *all storage should be of type* T. Similarly, a Zone-Member-Col policy like *a zone should not be both windows and linux hosts* can be transformed into multiple Zone-Member-Ind policies *there should be only type* T_i *hosts in zone* Z_i, where T_i is the current type of hosts in the Z_i. For these transformed policies, a pointer to the entity that provides the value to aggregate is also stored. This is required, since when the entity is deleted, the aggregate structure can be invalidated (can be re-populated using another entity, if existing).

For all other policies, we currently do not use any aggregate data structures.

7 Experimental Setup and Results

In this section, we evaluate our proposed optimizations as compared to the base policy evaluation provided by current engines. We start by describing our experimental setup beginning with the policy set.

7.1 Microbenchmarks

With the policy based management being in a nascent state so far, there does not exist any public set of policies that is used in a **real** SAN environment. The list of policies contained in [1] is indicative of the *type* of possible policies and not an accurate "trace" for an actual SAN policy set. As a result, it is tough to analyze the overall and cumulative benefits of the optimizations for a real SAN. To overcome this, we try to demonstrate the benefits of optimizations for different categories of policies individually. As mentioned earlier, since we have been able to classify all policies in [1] according to our scheme, the benefits would be additive and overall useful for a real SAN as well. In addition, this provides a good way of comparing the optimization techniques for each policy class.

We selected a set of 7 policies (Figure-3) as our working sample. Four of them are EC policies, two are path policies (SPTH and MPTH) and one is a zone policy. All 7 policies are classified according to the classification mechanisms presented in Section-6.1. Any aggregates that are possible for policies are also shown. For this set of policies, we will evaluate the effectiveness of our optimizations individually.

#	Policy	Classification
1	Every HBA that has a vendor name V and model M should have a firmware level either n1, n2 or n3	EC
2	No two devices in the system can have the same WWN (World Wide Name)	EC Unique
3	The number of ports of type X in the fabric is less than N	EC Counts
4	The SAN should not have mixed storage type such as SSA, FC and SCSI parallel	EC-Col to EC-Ind (Transform)
5	An ESS array is not available to open systems if an iSeries system is configured to array	SPTH
6	A HBA cannot be used to access both tape and disk drives	MPTH to SPTH (Transform)
7	No two different host types should exist in the same zone	Zone-Member-Col to Zone-Member-Ind

Figure 3: Policy Set

7.2 Storage Area Network

An important design goal for Zodiac was scalability and ability to perform impact analysis efficiently even on huge SANs of 1000 hosts and 200 controllers. Since it was not possible to construct such large SANs in the lab, for our experiments, we emulated four different sized SANs. Please note that in practice, Zodiac can work with any real SAN using an SMI-S compliant data store.

In our experimental SANs, we used hosts with two HBAs each and each HBA having two ports. Storage controllers had four ports each. The fabric was a core-edge design with 16-port edge and 128-port core switches. Each switch left certain ports unallocated, to simulate SANs constructed with future growth in mind. The four different configurations correspond to different number of hosts and controllers:

- **1000-200**: First configuration is an example of a big SAN, found in many data centers. It consists of 1000 hosts and 200 controllers with each host accessing all controllers (full connectivity). There were 100 zones.
- **750-150**: This configuration uses 750 hosts and 150 controllers with full connectivity. There were 75 zones.
- **500-100**: This configuration has 500 hosts, 100 controllers and 50 zones.
- **250-50**: This configuration is a relatively smaller SAN with 250 hosts, 50 controllers and 25 zones.

7.3 Implementation Techniques

For our experiments, we evaluate each of the policies in the policy-set with the following techniques:

- **base**: This technique is the naive implementation, in which there is no identification of the relevant region of the SAN. Only information available is the vanilla scope of the policy. Due to lack of classification logic, this implementation implies that the evaluation engine uses the same logic of code generation for all policies (check for all paths and all entities). Also, there is no intermediate caching and no aggregate data structures are used.
- **class**: This implementation uses the classification mechanism on top of the base framework. Thus, it is possible to optimize policies by only evaluating over a relevant SAN region, but no caching or aggregation is used.
- **cach**: This implementation technique caching on top of the *base* framework. No classification or aggregation is used.
- **agg**: This implementation technique only uses aggregate data structures for the policies (where ever possible). There is no caching or classification.
- **all**: This implementation uses a combination of all three optimization techniques.

Using these five classes of implementation, we intend to show (a) inadequacy of the base policy, (b) advantages of each optimization technique and (c) the performance of the **all** implementation. Zodiac is currently running on a P4 1.8 GHz machine with 512 MB RAM.

7.4 Policy Evaluation

In this section, we present our results of evaluating each policy 100 times to simulate scenarios of chaining and execution for multiple instances (e.g. adding 10 hosts). The policies are evaluated for all four SAN configurations (X-axis). The Y-axis plots the time taken to evaluate the policies in milliseconds. The results have been averaged over 10 runs.

7.4.1 Policy-1

"Every HBA that has a vendor name V and model M should have a firmware level either n1, n2 or n3"
The first policy requires a certain condition to hold on an HBA entity class. We analyze the impact of the policy when an HBA is added. The $base$ implementation will trigger the HBA scope and try to evaluate this policy. Due to its lack of classification logic, it will end up evaluating the policy afresh and thus, for all HBA instances. The $class$ implementation would identify it to be an EC-Ind policy and only evaluate on the new HBA entity. The $cach$ implementation does not help since there is no traversal of the graph. The agg implementation also does not help. As a result, all implementation is equivalent to having only $class$ optimization. Figure-4 shows the results for the different SAN configurations.

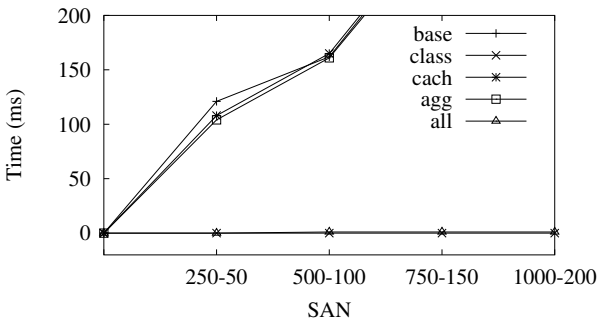

Figure 4: Policy-1. $class$, all provide maximum benefit

As seen from the graph, there is a significant difference between the best optimized evaluation (all) and the $base$ evaluation. Also, as the size of the SAN increases, the costs for the $base$ implementation increase, while the all implementation stays the same, since irrespective of SAN size, it only needs to evaluate the policy for the newly added HBA.

7.4.2 Policy-2

"No two devices in the system can have the same WWN."
The second policy ensures uniqueness of world wide names (WWNs). We analyze the impact when a new host is added. The $base$ implementation will trigger the $device$ scope without classification logic and check that all devices have unique WWNs. The $class$ implementation will only check that the new host has a unique WWN. The $cach$ implementation performs similar to $base$. The agg implementation will create a hashtable, and do hashtbale lookups. The all implementation also uses the hashtable and only checks the new host.

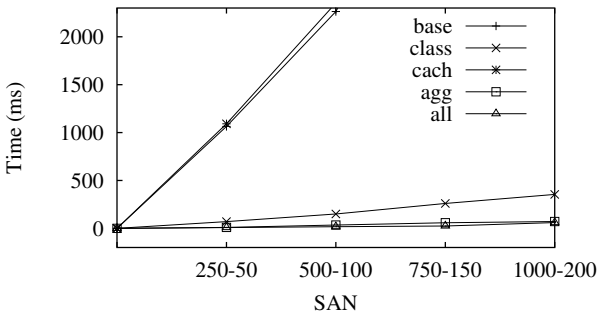

Figure 5: Policy-2. agg, all provide maximum benefit

As Figure-5 shows, agg and all perform much better than the $base$ implementation. $class$ performs better than $base$ by recognizing that only the new host needs to be checked.

7.4.3 Policy-3

"The number of ports of type X in the fabric is less than N."
The third policy limits the total number of ports in the fabric. We analyze the impact of adding a new host with 4 ports to the SAN. For each added port, the $base$ implementation will count the total number of ports in the fabric. The $class$ implementation performs no better, since it is an EC-Col policy. The $cach$ implementation also does not help. The agg implementation keeps a count of the number of ports and only increments the count and checks against the upper limit. The all implementation also exploits the aggregate keeping.

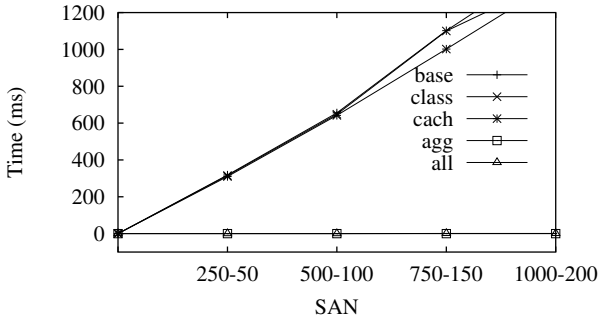

Figure 6: Policy-3. agg, all provide maximum benefit

As can be seen from Figure-6, agg and all perform significantly better due to the ability of aggregating the required information for the policy evaluation.

7.4.4 Policy-4

"The SAN should not have mixed storage type such as SSA, FC and SCSI parallel"
The fourth policy ensures that the SAN has uniform storage type. For this policy, we analyze the impact of adding a new storage controller. The $base$ implementation will trigger the storage scope and evaluate the policy ensuring all controllers are the same type. The $cach$ implementation will not help. The $class$ implementation only checks that the newly added controller is the same type as every other controller. The agg implementation will transform the policy to an EC-Ind policy by keeping an aggregate value of the current controller type, T in the SAN. However, without classification, it would end up checking that all controllers have the type T. The all implementation combines the classification logic and the aggregate transformation to only check for the new controller.

Figure-7 shows the result with all performing the best, while $class$ and agg doing better than $base$ and $cach$. The difference between the best and poor implementations is small since the total number of controllers is small.

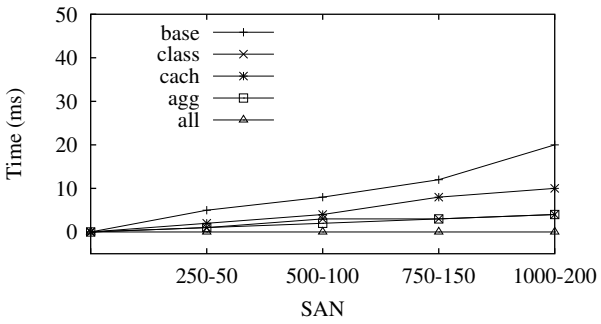

Figure 7: Policy-4. all provides maximum benefit

7.4.5 Policy-5

"An ESS array is not available to open systems if an iSeries system is configured to array."
The fifth policy is an SPTH policy that checks that an iSeries open systems host does not work if an ESS array is used with the controller. We analyze the impact of adding a new host to the SAN for this policy. The $base$ implementation ensures that all iSeries open systems hosts do not have any ESS controllers connected to them. This requires calling the $getController()$ API functions of the host entities and will cause traversals of the graph for all host-storage connections. The $class$ implementation identifies that it being an SPTH, only the new created paths (paths between the newly added host and the connected storage controllers) need to be checked. The $cach$ implementation will run similar to $base$, but will cache all function call results at intermediate nodes (As mentioned before, it would mean that each edge will be traversed atmost once). The agg implementation does not help and the all implementation would use both the classification logic and the caching.

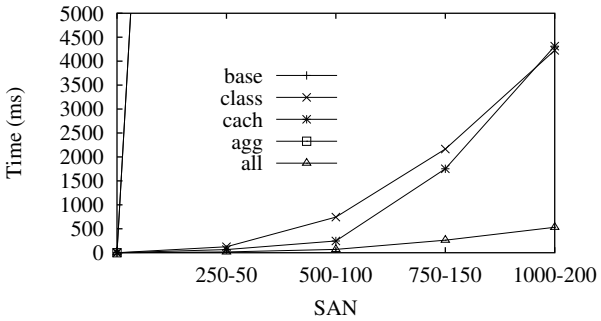

Figure 8: Policy-5. Only all provides maximum benefit

As shown in Figure-8, the $base$ and agg implementation perform extremely poorly (multiple orders of magnitude in difference) due to multiple traversals for the huge SAN graph. On the other hand, $class$ and $cach$ are able

to optimize significantly and their combination in the all implementation provides drastic overall benefits. It also scales extremely well with the increasing SAN size.

7.4.6 Policy-6

"A HBA cannot be used to access both tape and disk drives."
The sixth policy is an MPTH policy which requires checking that each HBA is either connected to tape or disk storage. We analyze the impact of adding a host with 2 HBAs to the SAN. The $base$ implementation would check the policy for all existing HBAs. The $cach$ implementation would do the same, except the caching of results at intermediate nodes. The $class$ implementation does not optimize in this case since it considers it an MPTH policy and checks for all paths (Section-6.1). The agg implementation transforms the policy to an SPTH by keeping aggregate for the type of storage being accessed, but checks for all HBAs due to the lack of classification logic. The all implementation is able to transform the policy and then use the SPTH classification logic to only check for the newly added HBAs.

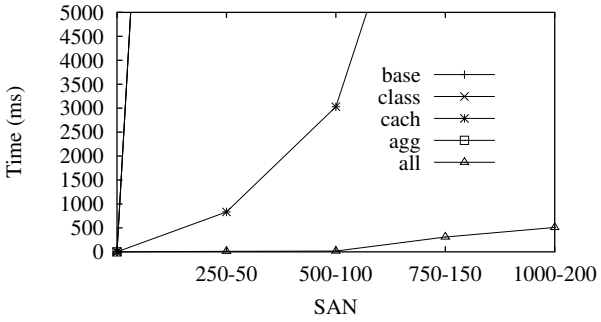

Figure 9: Policy-6. Only all provides maximum benefit

Figure-9 shows the results. The $base$, $class$ and agg implementation perform much poorly then the $cach$ implementation, since the $cach$ implementation reuses data collected once for the other. The all implementation performs the best by combining all optimizations.

7.4.7 Policy-7

"No two different host types should exist in the same zone."
The seventh policy requires that all host types should be the same in zones. We analyze the impact of adding a host HBA port to a zone. The $base$ implementation would check that all hosts in each zone are of the same type. The $class$ implementation would check only for the affected zone. The $cach$ implementation would be the same as $base$. The agg implementation would keep an aggregate host type for each zone and check the policy for all zones. The all implementation would combine the aggregate with the classification and only check for the affected zone, that the new host has the same type as the aggregate host type value. Figure-10 shows the results. Again all implementation performs the best, though the difference between all implementations is small.

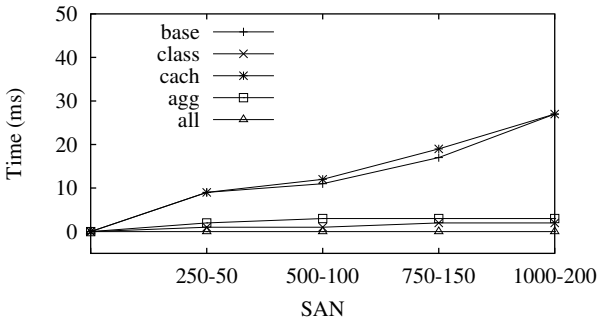

Figure 10: Policy-7. all provides maximum benefit

8 Discussion

One of the main objectives of the Zodiac framework is to efficiently perform impact analysis for policy enabled SANs. It is important to note that the optimizations described in this paper attack the problem at a higher conceptual level, manipulating the design and evaluation of policies. In the overall impact analysis picture, more optimizations will be plugged-in at other layers. For example, another layer of optimizations is while obtaining the data required for policy evaluation from the SMI-S data provider. In the CIM architecture [13], the CIM client obtains data from the provider over the network. This process can be optimized by techniques like batching of requests, pre-fetching and data caching at the client. Another important layer is the query language used for evaluating the policies. For example, it is possible to evaluate the policies using SQL by designing a local database scheme which is populated by the CIM client. While we continue to investigate such optimizations, Zodiac has been designed in a manner that it is easily possible to accommodate these into the overall framework.

9 Conclusions and Future Work

In this paper, we presented Zodiac - an efficient impact analysis framework for storage area networks. Zodiac enables system administrators to do proactive change analysis, by evaluating the impact of their proposed changes This analysis is fast, scalable and thorough. It includes the impact on SAN resources, existing policies and also, due to the actions triggered by any of the violated policies. In order to make the system efficient,

we proposed three optimizations - classification, caching and aggregation. Based on our analysis and experimental results, we find that each optimization has a niche of evaluation scenarios where it is most effective. For example, caching helps the most during the evaluation of path policies. Overall, a combination of the three optimization techniques yields the maximum benefits.

In future, we intend to follow two lines of work. The first includes developing more optimization techniques - smarter analysis for MPTH policies and use of parallelism (works for SPTH policies), to name a few and the design of a policy specification language that allows determination of these optimizations. The second direction explores the integration of the impact analysis framework with various SAN planning tools in order to provide better overall designs and potentially suggesting appropriate system policies for given design requirements.

References

[1] AGRAWAL, D., GILES, J., LEE, K., VORUGANTI, K., AND ADIB, K. Policy-Based Validation of SAN Configuration. *POLICY '04*.

[2] ALVAREZ, G., BOROWSKY, E., GO, S., ROMER, T., SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An Automated Resource Provisioning tool for large-scale Storage Systems. *ACM Trans. Comput. Syst. 19*, 4 (2001).

[3] ANDERSON, E. Simple table-based modeling of storage devices. *HP Labs Tech Report HPL-SSP-2001-4* (2001).

[4] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running Circles Around Storage Administration. In *FAST* (2002).

[5] ANDERSON, E., KALLAHALLA, M., SPENCE, S., SWAMINATHAN, R., AND WANG, Q. Ergastulum: Quickly finding near-optimal Storage System Designs. *HP Tech Report HPL-SSP-2001-5* (2001).

[6] ASSOCIATES, C. BrightStor. *http://www.ca.com* (2005).

[7] BANDARA, A., LUPU, E., AND RUSSO, A. Using Event Calculus to Formalise Policy Specification and Analysis. *POLICY '03*.

[8] BERENBRINK, P., BRINKMANN, A., AND SCHEIDELER, C. SimLab - A Simulation Environment for Storage Area Networks. In *Workshop on Parallel and Distributed Processing (PDP)* (2001).

[9] BUCY, J., GANGER, G., AND CONTRIBUTORS. The DiskSim Simulation Environment. *CMU-CS-03-102* (2003).

[10] CHAUDHURI, S., AND NARASAYYA, V. AutoAdmin 'what-if' Index Analysis Utility. In *SIGMOD* (1998).

[11] COHEN, I., CHASE, J., GOLDSZMIDT, M., KELLY, T., AND SYMONS, J. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI* (2004).

[12] DAMIANOU, N., DULAY, N., LUPU, E., AND SLOMAN, M. The Ponder Policy Specification Language. In *POLICY* (2001).

[13] DMTF. Common Information Model. *http://www.dmtf.org*.

[14] DOYLE, R., CHASE, J., ASAD, O., W., AND VAHDAT, A. Model-based resource provisioning in a web service utility. In *USITS* (2003).

[15] EMC. Control Center. *http://www.emc.com* (2005).

[16] EMC. SAN Advisor. *http://www.emc.com* (2005).

[17] FU, Z., WU, S., HUANG, H., LOH, K., GONG, F., BALDINE, I., AND XU, C. IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution. In *Workshop on Policies for Distributed Systems and Networks* (2001).

[18] GANGER, G., STRUNK, J., AND KLOSTERMAN, A. Self-* Storage: Brick-based Storage with Automated Administration. *CMU Tech Report CMU-CS-03-178* (2003).

[19] HP. StorageWorks SAN.

[20] IBM. TotalStorage Productivity Center.

[21] INTELLIMAGIC. Disc Magic. *http://www.intellimagic.nl* (2005).

[22] KEETON, K. Designing for disasters. In *FAST* (2004).

[23] KEETON, K., AND MERCHANT, A. A Framework for Evaluating Storage System Dependability. In *International Conference on Dependable Systems and Networks (DSN'04)* (2004).

[24] LYMBEROPOULOS, L., LUPU, E., AND SLOMAN, M. An Adaptive Policy-based Framework for Network Services Management. *Journal of Networks and System Management 11*, 3 (2003).

[25] MOLERO, X., SILLA, F., SANTONJA, V., AND DUATO, J. Modeling and Simulation of Storage Area Networks. In *MASCOTS* (2000).

[26] ONARO. SANscreen. *http://www.onaro.com*.

[27] RUEMMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (1994).

[28] SNIA. Storage Management Initiative. *http://www.snia.org*.

[29] THERESKA, E., NARAYANAN, D., AND GANGER, G. Towards self-predicting systems: What if you could ask "what-if"? In *Workshop on Self-adaptive and Autonomic Comp. Systems* (2005).

[30] VARKI, E., MERCHANT, A., XU, J., AND QIU, X. Issues and Challegenges in the Performance Analysis of Real Disk Arrays. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (2004).

[31] WANG, M., AU, K., AILAMAKI, A., BROCKWELL, A., FALOUTSOS, C., AND GANGER, G. Storage device performance prediction with CART models. *SIGMETRICS Performance Eval. Review 32*, 1 (2004).

[32] WARD, J., SULLIVAN, M., SHAHOUMIAN, T., AND WILKES, J. Appia: Automatic Storage Area Network Fabric Design. In *FAST '02*.

[33] WARD, J., SULLIVAN, M., SHAHOUMIAN, T., WILKES, J., WU, R., AND BEYER, D. Appia and the HP SAN Designer: Automatic Storage Area Network Fabric Design. In *HP Tech. Conference* (2003).

[34] WILKES, J. The Pantheon Storage-System Simulator. *HP Labs Tech Report HPL-SSP-95-14* (1995).

Notes

[1] This policy is also a collection policy since in order to evaluate the policy for the new instance, it is required to get information about existing instances.

[2] Informally, typically an operation affecting only the "principal" entity of the policy (host in the examples) does not require checking old paths.

[3] Such a policy is required since the switches have a limit on the number of zones they can handle