Root-Cause Analysis of SAN Performance Problems: An I/O Path Affine Search Approach

David Breitgand, Ealan Henis, Edya Ladan-Mozes[*1]*, Onn Shehory, Elena Yerushalmi*
IBM Haifa Research Labs
Haifa University Campus
Mount Carmel, Haifa
31905, Israel
{*davidbr,ealan,onn*}*@il.ibm.com*

Abstract

We present a novel algorithm, called IPASS, for root cause analysis of performance problems in Storage Area Networks (SANs). The algorithm uses configuration information available in a typical SAN to construct I/O paths, that connect between consumers and providers of the storage resources. When a performance problem is reported for a storage consumer in the SAN, IPASS uses the configuration information in an on-line manner to construct an I/O path for this consumer. As the path construction advances, IPASS performs an *informed search* for the root cause of the problem. The underlying rationale is that if the performance problem registered at the storage consumer is indeed related to the SAN itself, the root causes of the problem are more likely to be found on the relevant I/O paths within the SAN. We evaluate the performance of IPASS analytically and empirically, comparing it to known, informed and uninformed search algorithms. Our simulations suggest that IPASS scales 7 to 10 times better than the reference algorithms. Although our primary target domain is SAN, IPASS is a generic algorithm. Therefore, we believe that IPASS can be efficiently used as a building block for performance management solutions in other contexts as well.

Keywords

storage area network, SAN, root cause analysis, search, network management

1. Introduction and Motivation

A Storage Area Network (SAN) is a dedicated high-speed network connecting multiple storage servers (usually being powerful hosts) to multiple highly capable storage devices. Figure 1 depicts a typical SAN architecture. A SAN is optimized for carrying only I/O traffic between the storage servers and the storage devices and, possibly, among the storage devices themselves without the server's intervention[*2]. A SAN does not carry application traffic. The latter is handled by a separate messaging network such as LAN. This approach off-loads the main messaging network, simplifies sharing of storage resources, improves data availability, eliminates the bottlenecks associated with the directly

[*1] Edya Ladad-Mozes is currently affiliated with the CS dept. at Tel-Aviv University. This work was done when she was affiliated with IBM-HRL.

[*2] Although sometimes the term SAN refers only to the switching fabric, in the context of this work, we use the term SAN to include the storage servers, and the back-end storage devices as well.

0-7803-9087-3/05/$20.00 ©2005 IEEE

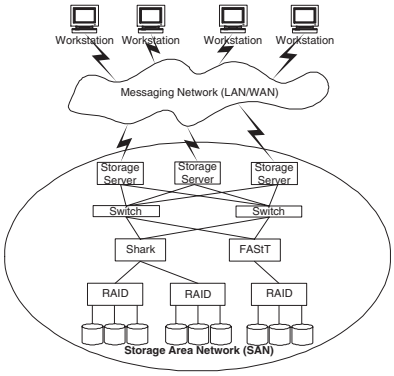


Figure 1: Typical SAN Architecture

attached storage model, and enables new storage models, such as server clustering [8, 7]. Consequently, SANs become a leading solution for organizing storage resources in large enterprizes for efficient handling of ever growing multi-terabyte volumes of business data.

The SAN model allows virtualization of the heterogeneous storage resources across the enterprize by creating a pool of storage that can be shared by multiple consumers. This pool consists of *Logical Units (LU)*, where each LU is backed by *blocks* on one or more physical devices. Potentially, any consumer can be configured to access any LU by setting up *mappings* between the logical volumes of the consumer hosts and the appropriate LUs. The mappings may be quite complex and include multiple levels of indirection. The mappings are, usually, administered through configuration management tools, and constitute an essential part of the SAN configuration. In this paper, we propose an explicit usage of configuration information, such as mappings between the hosts logical volumes and SAN logical units and physical back-end devices, for identifying root causes of SAN performance problems.

Performance of a computer system is measured with respect to the operational characteristics which are important to the applications. Typically, application and system performance is assessed in terms of *throughput* and *response* time. There are two primary sources of performance problems in SANs that may affect the overall application performance: (a) malfunctioning or insufficient capacity of the entities comprising the SAN; (b) application-induced contention due to the centralized storage access model (*i.e.*, the storage pool model).

To provide protection and security for different applications using the shared storage pool, and also to reduce contention, a configuration management mechanism, known as *zoning*, is often used. Zones partition the SAN into logical groupings of devices that can share information. While being an efficient and versatile management tool, zoning cannot totally eliminate the potential for contention. Namely, contention may still occur within the zone, albeit with a lower probability. It should be understood, that the possibility of contention is intrinsic to SANs, as it stems from the shared pool model of storage access. The latter is also one of the more important advantages of the SAN model.

As shown in Figure 1, a SAN is a complex system that consists of multiple physical and logical entities, such as storage servers, switches, disk arrays, logical volumes, logical units, *etc.*, being interconnected in a complex way. Large SANs may include thousands of different inter-related logical and physical entities. When an application performance problem is detected and reported either by an application's user, or by an automatic monitoring tool, the root cause of this performance problem can be anywhere in the system, including the SAN, LAN, storage server, database, application server, client machine, *etc*. This work focuses on determining root causes of performance problems by identifying entities (physical and logical) within the SAN that are more likely to cause the performance problem at hand. These entities are termed *problematic entities*. The problematic performance behavior of an entity is reflected by the unusual values of its operational variables, also known as performance metrics. Common examples of performance metrics are *mean queue length, mean response time, mean throughput*, *etc.* Typical performance management techniques concentrate on monitoring the performance metrics of the individual elements comprising the SAN, and reporting the deviations from their normative behavior (*e.g.*, as defined by the local pre-specified performance-related thresholds), to the manager of the system. The entities that violate their performance thresholds are the problematic entities. However, firstly, not all problematic entities that may exist simultaneously in a complex system such as SAN, have the same importance for the system-level performance. Secondly, only a few of them may be in the root of the performance problem detected at the SAN level, on the application side. And, thirdly, since it is very difficult to specify the performance thresholds with high accuracy in advance, false alarms are possible.

Presently, the administrator is expected to determine manually the relationships between the reported threshold violations pertaining to the problematic entities scattered all over the system, and performance problems detected at the application level. This is done based on her knowledge of the system. While being feasible for small SANs (consisting of dozens of elements), this approach does not scale. Firstly, the number of alarms that the managed system generates may become very large. Secondly, the entity-level alarms on their own are insufficient for determining the root causes of the system-wide problems that manifest themselves at the application level. Therefore, a scalable root cause analysis for SAN requires extending the entity-level management tools with methods that take into account relationships among the entities. However, the number of relationships between the managed entities in the SAN grows exponentially with the number of SAN entities. This makes a brute force approach infeasible, and calls for more efficient algorithms for searching the inter-related entities.

In this work, we present an I/O Path Affine SAN Search solution (IPASS), a heuristic that, on average, reduces the exponential complexity of the search to a low polynomial one. Similarly to [2, 5, 4], IPASS models a SAN as a graph. Using this graph, IPASS focuses its search for root-cause entities on the I/O paths that emanate from the node representing the SAN entry point of an application that reports a performance problem. The rationale for this heuristic is that the SAN-related performance problems of an application usually result from bottlenecks in the storage devices. These are most likely to be found on the I/O paths of this application. It is further hypothesized that the application perfor-

mance problem should manifest itself through changes in the operational characteristics of the entities on the I/O path.

IPASS assumes that thresholds defining the normal range of functioning are defined for every entity in the SAN. Setting these thresholds in such a way that the number of false alarms would be minimal, while all the true alarms would still be reported is a tough research problem. The threshold definition problem is out of the scope of this paper.

2. Related Work

Relatively few publications deal with performance management in Storage Area Networks, and, specifically, with the root cause analysis of performance problems in SANs. Partially, this can be attributed to the fact that this field is relatively new. In [2], Chambliss *et al.* suggested an algorithm that looks for the root causes of performance problems by systematically traversing a weighted graph that models dependencies between the entities in the SAN. When traversing the graph starting from some entity for which a problem was detected, the algorithm prefers as the next entity, the one for which some score function (*e.g.*, the magnitude of a threshold violation) is maximal. Essentially, the search algorithm proposed in [2] is an informed greedy search. The main problem with this approach is the need to know the model (*i.e.*, the dependencies among the entities) in advance.

Kochut *et al.* [4] presented a three-stage performance management algorithm, where the third stage attempts to find SAN elements that may be the source of a problem. In similarity to our algorithm their solution utilizes the combination of static data on graph topology and configuration and dynamic performance measurements on the elements of the SAN. However, the search performed in stage two of their algorithm is exhaustive - all SAN elements are checked. This approach may work well for small SANs, but it would not scale. Additionally, the suggested algorithm does not cover all types of performance problems. It should typically work well for contention problem, however other problems will usually remain undetected. Similarly to [2], the algorithm assumes the knowledge of dependencies among the entities in the SAN in advance. In [5], Kochut *et al.* proposed a technique to discover the dependencies among the SAN entities, and in particular, the data paths, dynamically. However, this approach is computationally expensive and would typically work well for a rather small SAN.

When attempting to locate a small number of unique states in a large state space, search algorithms are a natural solution approach [6]. The search for root cause entities in a SAN is exactly such a problem, as the SAN consists of multiple entities, of which only a few are the root cause of a specific performance problem. In principle, it is possible to locate root cause entities by running an exhaustive search, *e.g.*, breadth-first search (BFS). Although BFS guarantees that a solution will be found if one exists, its complexity, even for the average case, is of the order of the search space. Depth-first search (DFS) might perform better for some search spaces, however the complexity is still high.

One may suggest that the SAN search space is sufficiently small for using BFS and DFS. This claim may be correct if what one seeks is a single off-line search of the SAN. Large SANs consist of thousands of entities, and a combinatorial number of links among the entities. On each entity, multiple performance parameters are measured and, for mean-

ingful performance analysis, these should be measured rather frequently. As a result, an exhaustive on-line search of the SAN, its links, and their dynamically changing performance readings, is infeasible. Therefore, the search space must be pruned.

Search can be significantly improved by the use of domain-specific information. Although the worst-case complexity of informed search algorithms [1] is similar to the complexity of the exhaustive search, the average case complexity is usually significantly better. In the case of a SAN configuration information, and, in particular, mapping, addressing links between entities, and zoning information can direct the search to the more relevant SAN entities. This exactly what our study suggests: the search is restricted to I/O paths in the SAN, and I/O paths can be inferred using the configuration data, which is available as part of system configuration. Complete and optimal informed search algorithms are, for example, $A*$ and its derivatives. Those algorithms use heuristics to constrain the search space, however in the worst case they may search the entire space. Our approach takes one step further in restricting the space: we allow search only on I/O paths. As a result, our algorithm should rarely search the entire space, and usually it will search only a small subset thereof.

3. Model and Problem Statement

We model a SAN as a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. An example of a SAN graph is given in Figure 2. There are two types of nodes: *regular-nodes*, and *super-nodes*. A regular node models either a physical (hardware) or a logical (software) atomic entity, such as *Host Bus Adapter (HBA)*, *Logical Volume*, *Logical Unit*, *etc*. A super-node models a collection of nodes and edges as detailed below.

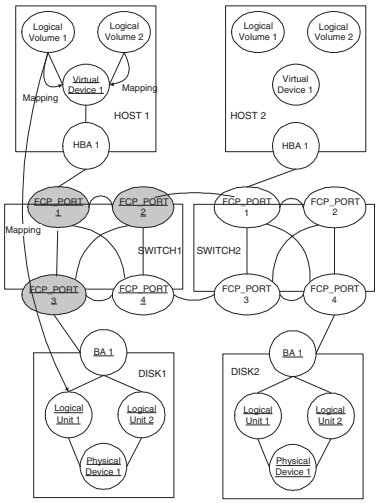


Figure 2: Typical SAN Graph (not all mappings are shown)

Each edge represents a link connecting a pair of nodes or super-nodes, and refers

to a physical (*e.g.*, communication link), or logical (*e.g.*, address mapping) relationship between the nodes. Consequently, there are two types of links: *physical*, and *logical*. In contrast to the physical links, the logical links may connect the nodes that are not directly connected in the physical topology of the SAN. Logical links are *directed*.

Common to all storage virtualization techniques is explicit mapping of the contiguous segments of the virtual storage space (provided by the virtual devices) onto contiguous segments of the physical storage space (provided by the back-end physical devices). The mapping usually comes in form of *mapping tables* that can be manipulated by provisioning tools and by human administrators. In this paper, we assume that the mapping tables are available to our algorithm as part of the SAN's logical topology. Address mappings found in the mapping tables are represented by the logical links between the nodes in the SAN graph. For example, the logical link between Logical Volume 1 of Host 1, in Figure 2, and the Logical Unit 1 offered by the Disk Controller 1, represents the address mapping between the consumer of the storage and its provider.

The super-nodes bundle the regular nodes sharing the same properties (and edges connecting these nodes) into a single logical entity. This allows for coarser granularity of presentation, explicit modelling of relationships among the entities, and helps to speed up the navigation of the SAN graph. A switch is an example of a super-node. In our model, it can be represented as a collection of ports.

Each node has a vector R of attributes representing the resources of this node. Resource examples are capacity of a disk controller, its bandwidth, *etc*. With respect to each resource in R, the entity corresponding to a specific node may act as either a *consumer*, or a *provider*, or a *mediator*. For example, a file server consumes storage capacity, a disk array provides it, and a switch port mediates between the two. It is important to notice that, in our model, the roles of the entities with respect to the resources are fixed.

Each resource has one or more *metrics* associated with it. At any given point in time, each metric has an operational value measured directly from the entity to which this metric belongs. Useful SAN performance management metrics include queue length, throughput, capacity, utilization, *etc*. For each metric, there is a set of the pre-computed *thresholds* that define the range in which the operational values of this metric should reside under normal operation conditions. The nodes at which, for one or more metrics, the operational values violate their thresholds are termed *problematic nodes*. The thresholds can be recalculated in the course of operation and reset to new values. However, the problem of threshold calculation is out of the scope of this work.

The SAN graph topology is discerned from the physical and logical topologies of the underlying SAN. These topologies are assumed to be known in advance or, alternatively, detected by standard tools (e.g., IBM Tivoli SAN Manager(ITSANM) [3]). The SAN logical topology usually enforces restrictions on top of the physical topology, such as limitations on node access. For example, a RAID array may be physically connected to a switch that facilitates direct connections to multiple file servers. However, the RAID, usually, can only be accessed by these servers through the Logical Volume Manager (LVM), and not directly. In our model, the SAN graph does not include edges that correspond to physical connections which are prohibited by the logical topology.

Another important type of configuration information is zoning. It defines the logical

groups of devices that may share information. Figure 2 shows two such zones in the SAN. The ”gray— ports belong to one zone, and the ”white” ports belong to another one. The address mappings between the consumers and providers of the storage resources are allowed only between the entities belonging to the same zone. There are a few types of zones that are commonly used in SANs. In Section 6 we explain more about the zones. One type, the *port-based* zoning, which enumerates the ports that can ”see” each other, is used in our simulation studies.

Definition 1 (Problem Statement) *Given the graph representation of the SAN, the operational values of the resource metrics for the entities comprising it, and the indication of a performance problem at a specific node (*e.g.*, threshold violation at an application server), devise an algorithm that would find the root cause entities within the SAN,* i.e.*, the entities from which the performance problem likely has originated.*

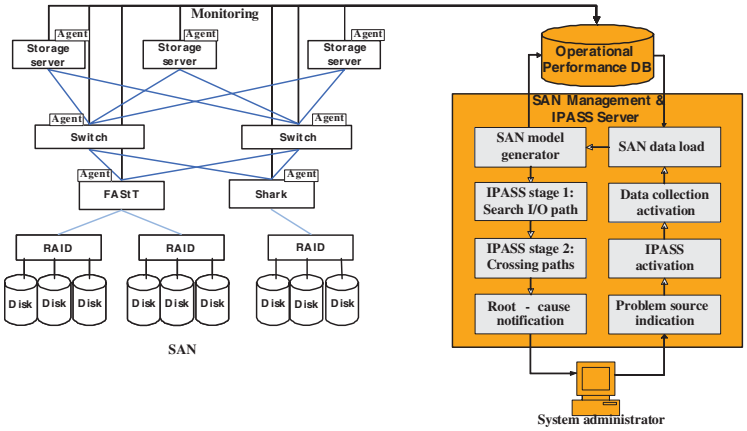


Figure 3: The root cause analysis tool architecture

4. Architecture

Figure 3 depicts the architecture of the IPASS root cause analysis tool. There are four major components in the architecture.

- Monitoring agents: software components that periodically perform measurements of the operational values of resource metrics at the SAN entities, and report this information to the operational database.
- Operational database: a database that serves as a repository of configuration and performance information produced by the monitoring agents.
- Performance analysis front-end: the realization of the IPASS algorithm described in the following section. The data for analysis is retrieved from the operational database. The operation of the IPASS algorithm may be guided by an administrator of the SAN either through the GUI or programmatically.

- GUI: this is an optional part of the architecture. It presents to the system administrator convenient means of invoking the IPASS algorithm and controlling its execution

5. The IPASS Algorithm

IPASS is an informed search algorithm traversing the SAN graph*3. It assumes the availability of mapping tables and, possibly, other configuration information. The main underlying idea of the algorithm is that given an initial problematic node, root cause problematic entities are most likely to be located on the I/O paths emanating from this node. In functional SANs, these paths form a directed acyclic graph DAG that starts from a consumer C, proceeds through mediators and terminates at the relevant providers' nodes.

The IPASS algorithm consists of two major stages: a first stage in which I/O paths emanating from C are searched for problematic nodes $PN1$; an optional second stage in which I/O paths crossing through nodes in $PN1$ are search for additional problematic nodes $PN2$. The first stage, although can be implemented in several ways, is essentially a DFS-like search guided by mapping and zoning information, that starts from C and gradually constructs DAG. The goal of this search is to find a set of problematic nodes that belong to the I/O path emanating from C. The rationale for this is that these nodes are the more likely ones to be related to the performance problem recorded at C. In a basic IPASS setting, the algorithm fully explores the I/O path collecting all the problematic nodes. In a more conservative setting, IPASS continues its search until either N (being a parameter of the algorithm) problematic nodes are found, or the I/O path is fully explored*4*.

The details of the first stage (in its basic form) are presented in Figure 4. For simplicity, we do not include zoning configuration information treatment into the algorithm presentation. It is implicitly assumed that the algorithm operates within a single zone. As is the case with the classic DFS search, initially all nodes in the SAN graph are marked *white*. When the node is visited it is marked *gray*, and when all its neighbors have been explored, it is marked *black*. The first stage of IPASS halts when it finds all problematic nodes on the I/O path. It requires a trivial modification to allow an early stop when N problematic nodes are found.

By using the SAN graph and the mapping information within, the search is restricted to I/O paths. As is commonly the case for informed search algorithms, the additional information prunes the search space. Within the subgraph of the I/O paths emanating from the starting point, this search is essentially a DFS search. From this we can conclude that *the search is complete*: if a solution exists, the algorithm will find it.

The second stage of the IPASS algorithm is not mandatory, and is activated upon user request. Once a SAN administrator reviews the set of problematic nodes reported, PN, she may decide to request further search. The second stage of the search traverses the graph through I/O paths that cross the nodes reported in PN. On these I/O paths, prob-

*3 We refer here to the SAN graph including the mapping links.

*4 For simplicity, we present IPASS in its simplest form. In particular, we represent the switching fabric as a single node with all port nodes connected to it. In actual implementations, however, it may be important to distinguish between different switches inside the fabric, and represent the fabric as a mesh. Searching the mesh efficiently going from consumers to producers is a non-trivial algorithmic task, since the ports in the mesh lack additional information defining their relationships with consumers and producers. To this end, we developed an extension of IPASS, to be presented in a longer version of this paper.

```
1.  SG : the graph presentation of the SAN (topology + mappings)
2.  C : the initial problematic node(consumer)
3.  P ← C.getMappings().getTargets() : target nodes (providers for C according to mappings)
4.  N:the number of problematic nodes to search for
5.  PN ← ∅ : the set of problematic nodes found at stage 1
6.  found ← 0 : the number of problematic nodes found
7.  DAG ← ∅ : I/O Path that is being constructed
ipass1(SG, C) {
8.  while P ≢ ∅ do
9.    buildIOPath(C)
10. return PN }
buildIOPath(cur) {
11. cur.color ← GRAY
12. IOneighbors ← computeIONeighbors(cur)
13. if (IOneighbors ≡ ∅) {
14.   cur.color ← BLACK
15.   return FALSE
16. }
17. foreach nbr in IOneighbors do
18.   if(nbr.color == WHITE) {
19.     DAG.insert(cur, nbr) //inserts nbr after cur into DAG
20.     if (nbr violates threshold)
21.       PN.add(nbr)
22.     if (nbr ∈ P) {
23.       nbr.color ← BLACK
24.       P.remove(nbr)
25.       return TRUE
26.     }
27.     if (buildIOPath(nbr) == FALSE) {
28.       DAG.removeBranch(cur) //removes nbr and all entities in DAG that follow nbr from
DAG and PN
29.       return FALSE
30.     }
31. } }
computeIONeighbors(node nd) {
32. resultList ← ∅
33. foreach neighbor n of nd in SG do
34.   if a physical link (nd, n) exists
35.     resultSet.add(n)
36.   mappings ← nd.getMappings()
37.   resultSet ← resultSet ⋂ mappings
38.   return resultList }
```

Figure 4: Stage 1: search I/O paths from a storage consumer to its storage providers

lematic nodes are searched for. The rationale of this stage is that the root cause of a storage performance problem on one I/O path may reside on another, crossing I/O path. For instance, two applications may be using different logical units backed by the same disk controller. However one of the applications may create excessive loads at the controller, thus affecting another one.

Within the zone, the second stage traverses at most $N \cdot z$ crossing I/O paths. The search within each I/O path is similar to the search in the first stage. The details of the second stage are in Figure 5. For the sake of simplicity the algorithm presented in Figure 5 does not deal with the overlappings between the I/O paths that are being constructed. In the

actual implementation, path overlapping is taken into account in order to maintain the optimality of the search, and explore each sub-path only once.

6. Algorithm Evaluation

Let complexity of the search algorithm be the number of entities in the SAN graph that it has to inspect, until it finds all the problematic entities that may be relevant to the performance problem at hand.

To analyze the performance of the IPASS algorithm, let us consider again the typical SAN topology presented in Figure 2. In the first stage of the algorithm, an I/O path emanating from the entry point into the SAN is constructed. One such path is shown by the bold line in the figure. Obviously, in the worst case, IPASS has to construct the full path before it discovers a problematic entity (in this example, Physical Device 14) that may be in the root of the performance problem. Since, I/O paths always starts at some consumer and always terminates at some provider node, the maximal length of an I/O path equals the diameter of the SAN graph, d. One may observe, that since the same consumer node may be mapped to multiple provider nodes located in the back-end storage devices being "behind" the switch fabric. Each mapping may, in the worst case relate the same consumer node to a different back-end device. Thus, an I/O path may be split approximately half-way, "on the exit" from the switching fabric with each branch leading to different device. Therefore the complexity of the IPASS algorithm, in the worst case, is $O(d)$. The actual value of the multiplicative constant factor depends on the maximal out-degree of the consumer nodes with respect to the address mapping links, M.

Thus, in the worst case, IPASS considerably outperforms the uninformed exhaustive search algorithms such as BFS, whose complexity is $\Theta(V + E)$, and DFS, whose com-

```
1.  ipass2(SG, PN) {
2.    // variables of stage 1 are available for use in stage 2
3.    PN1 ← PN : the set of problematic nodes found at Stage 1
4.    PN2 ← ∅ : the set of problematic nodes found at Stage 2
5.    P ← ∅ : set of target nodes (as in Stage 1)
6.    consumers : all consumer nodes in the zone
7.    while (PN1.size ¿ 0) do {
8.      currentNode ← PN1.first
9.      PN1.remove(currentNode)
10.     P.add(currentNode)
11.     foreach c ∈ consumers do {
12.       buildIOPath(c) //builds I/O path starting at c, and terminating at the target node in P
13.       //and terminating at the target node in P (results are in PN)
14.       PN2.add(PN)
15.       P ← c.getMappings().getTargets()
16.       ipass1(SG,currentNode)
17.       PN2.add(PN)
18.     }
19.   }
20.   return PN2
21. }
```

Figure 5: IPASS stage 2: search crossing I/O paths

plexity is $\Theta(V + E)$. However, the worst case is not necessarily the most common one. It may still be claimed thus, that in practice, the difference between all search algorithms in terms of their average complexity would be marginal. In order to explore the relative performance of IPASS in the average case, we conducted a comprehensive simulation study that is described in the next subsection.

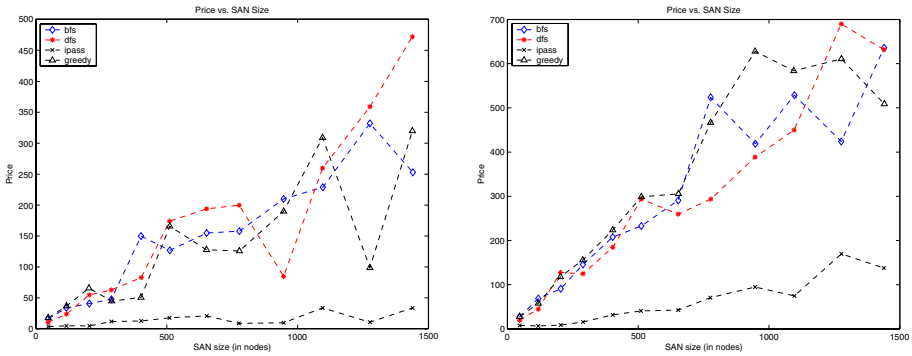
6.1 Simulation Study

Actual traces containing the SAN performance problems are difficult to obtain. Note that even if such traces were publicly available, their value would be limited, as each trace represents a specific case study of a specific system. Therefore, the results obtained through the trace-driven simulation of this kind would be difficult to generalize. Consequently, we perform a comprehensive comparative study through simulation that is not based on any specific performance trace from a real system, trying instead to cover possible behavior of a wide range of systems. Our simulations are organized as follows.

- Prior to each run, we randomly generate a SAN topology similar to that of Figure 2. The SAN graph's actual form is controlled by five main parameters: number of hosts, an average number of logical volumes per host, share factor, *i.e.*, $\frac{number\ of\ providers}{number\ of\ consumers}$, average size of the port-based zones, and maximal mapping out-degree
- Next, an entry point into the SAN, which in our study is always a logical volume is randomly chosen, and an I/O path for this node is constructed off-line
- On the I/O path constructed in the previous step, we randomly choose a "root cause" node, and inject a failure into this node with a pre-specified probability p of problem propagation upstream from this node. We term it the *probability of problem diffusion*. We then perform a number of Bernoulli trials on the upstream nodes, to simulate the problem diffusion in the network. The nodes with induced problem together with the root cause node form the *root cause set*.
- Next, we randomly draw a number of nodes in the SAN that are outside the constructed I/O path, and inject "irrelevant" problems (manifested as threshold violations) into these nodes to satisfy a pre-specified ratio of true problems: $P_{true} = \frac{real\ problems}{all\ problems}$.
- When the above steps are completed, we run four different search algorithms: IPASS (first stage), BFS, DFS, and Greedy DFS, to find the given number of the nodes from the root cause set. We assume that there is an oracle that can tell whether the node found by an algorithm belongs to the root cause set or not.

The Greedy DFS algorithm is essentially a prioritized DFS, that at each stage prefers first to explore the neighbors that exhibit threshold violations. The intuition behind this heuristic is that the Greedy DFS tries first to explore branches that seem more suspicious than the other ones. The number of nodes visited by each algorithm is recorded and represents its "cost" in the given run. For each set of configuration parameters, each run is repeated 100 times in order to eliminate statistical dependency and obtain the meaningful average values. Due to the lack of space, we present only the more important results highlighting the main traits in the algorithm's behavior.

Figure 6 shows the cost of search algorithms as a function of topology size. Figure 6(a) shows the cost of the algorithms (*i.e.*, the number of nodes in the topology that it inspects until the search goal is satisfied), for the search goal of finding only a single relevant



(a) searching for a single problematic node (b) searching for all problematic nodes

Figure 6: Cost of the Search Algorithms as a Function of Topology Size

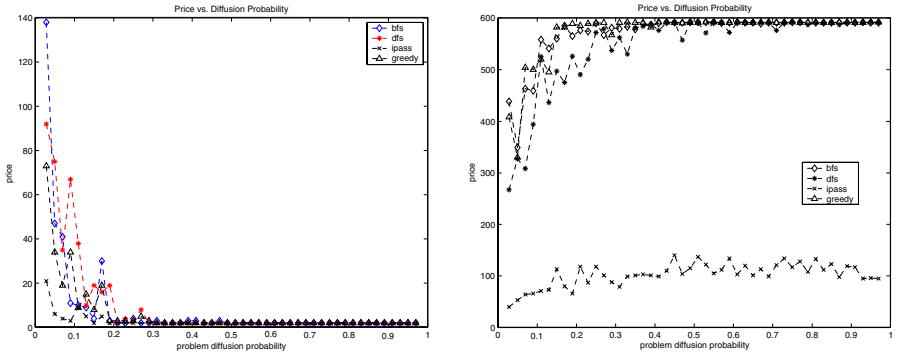
problematic node. Recall that the problematic node is considered relevant, if it belongs to the I/O path of the consumer that initially reported that there exists a performance problem. Figure 6(b) reports the similar results, but in this case the search goal was set to find *all* relevant problematic nodes.

In both cases, $P_{true} = 0.7$, *i.e.*, 70% of all problems that were injected were relevant. The ratio between providers (backend devices) and consumers was set to 0.7, and the average zone included 10% of the consumer ports. In this set of experiments, only one root cause problem was injected, and the probability of problem diffusion in this set of experiments was set to 1%. One may notice that these settings imply that there will be very few problematic nodes relatively to the total size of the topology. However, 70% of these nodes are *relevant* to the performance problem being investigated. One may also expect that when the algorithms search for all problematic nodes and not just for a single one, their cost should increase on the average, as is indeed the case (see Figure 6(b)). Notice that the graphs exhibit a high variance. This is much more pronounced for the uninformed search algorithms. Although the variance for the uninformed search algorithms decreases as the number of replications increases, it would still be substantial unless the outliers (corresponding to the worst case runs) are censored. The figures present the uncensored data. The random nature of problem injection and topology generation, however, inevitabely results in large outliers for the uninformed search algorithms. IPASS is less susceptible to this problems because in the worst case its performance is only a constant fraction of the topology. Despite the outliers, the presented graphs are sufficiently informative to derive conclusions regarding the relative average case performance of IPASS.

Figure 6(a) shows, that while the cost of IPASS stays flat as the SAN size grows, the cost of the uninformed search (DFS, BFS, Greedy DFS) grows linearly with the size of the topology. As one may expect, when the algorithms search for all problematic nodes, their average cost should increase, as is indeed the case with all the simulated search algorithms (see Figure 6(b)). The cost of both informed and uninformed search algorithms grows linearly with the size of the SAN. However, the slope for the uninformed search algorithms is approximately 1, and the slope for IPASS is much smaller. This suggests that

IPASS scales considerably better than the uninformed search algorithms. As observed, the gain in effectiveness ranges from the multiplicative factor of 7 to an order of magnitude.

If diffusion probability, the initial number of root cause problems, and the P_{true} ratio remain fixed, then the major important factor that affects the performance of IPASS relatively to the uninformed search is the average zone size. Indeed, within the fabric, IPASS cannot get assistance from the mapping information, because there is usually no fixed mapping between any combination of input and output ports. Therefore, when constructing the I/O paths within the zone, IPASS will have to explore multiple irrelevant branches that will be subsequently pruned from the I/O path. Thus, as the zone size increases, so does the cost of IPASS. Our simulations suggest that the efficiency gain of IPASS is inversely proportional to the zone size.



(a) searching for single problematic node (b) searching for all problematic nodes

Figure 7: Cost of Search Algorithms as a Function of Problem Diffusion Probability

Given the fixed topology parameters, the major factor that affects the efficiency of IPASS is the probability of problem diffusion. Figures 7(a,b) show the cost of the algorithms as a function of the problem diffusion probability for a fixed SAN size: 32 hosts, approximately 1600 nodes in total. As one can see, for the smaller probabilities, *i.e.*, for the case when there are fewer relevant problematic nodes in the SAN topology, IPASS performs an order of magnitude better than the uninformed search algorithm. Figure 7(a) suggests that as the diffusion probability increases, the cost gain of IPASS becomes marginal. When probability of diffusion passes over 20% all algorithms perform similarly in terms of their cost, when it is required to find only a single relevant problematic node because the chances to encounter a relevant node after only a few steps is become very high. However, as seen from Figure 7(b), if the algorithms are required to find all the relevant problematic nodes, then IPASS is clearly superior. As in the previously described set of experiments, the cost gain of IPASS ranges from the multiplicative factor of 7 to an order of magnitude. One may also notice from Figure 7(b) that the cost of IPASS remains flat, as was suggested by the worst case analysis in Section 5.

In the light of the above, we conclude that in the average case, the informed search algorithm like IPASS is clearly preferable to the naive uninformed search algorithms, as

it provides a substantial efficiency gain. The naive search algorithms do not scale well, while IPASS exhibits a much better potential for scaling.

7. Conclusion and Future Work

We presented a novel informed search algorithm, IPASS, that can serve as an efficient building block for root cause analysis in Storage Area Networks. As our analysis and simulation study suggest, IPASS is factor 7 to an order of magnitude more efficient than the uninformed search algorithms that are often suggested as a naive alternative. We demonstrated that while the naive algorithms scale poorly, IPASS exhibits an acceptable scalability. The primary target domain of IPASS is performance management of SANs. However, it is a generic algorithm that can be extended to other domains.

Our future directions include deployment of the IPASS prototype in an actual SAN, and gaining further understanding with it. Another important extension of the presented work is development of automatic threshold setting for SAN entities with respect to the application Service Level Objectives. Currently we assume that the sufficiently accurate thresholds are set manually in the system. Although this is a usual assumption, we believe that it is not realistic for large systems.

8. Acknowledgments

We thank Norman Bobroff, Kirk Beaty and Gautam Kar for their valuable comments.

References

[1] S. Russell and P. Norvig. *Introduction to Artificial Intelligence - a Modern Approach*. Prentice Hall, 1995.

[2] D. Chambliss and P. Pandey. Aurora: Technology for storage performance managemement. US Patent ARC920020048US1: A System and Method for Managing the Performance of a Computer System based on Operational Characteristics of the System Components, December 2002.

[3] IBM Tivoli. IBM Tivoli Storage Area Network Manager (ITSANM). `http://www.ibm.com/software/info/testinfo.jsp?uid=IC00009`, 2003.

[4] A. Kochut, N. Bobroff, K. Beaty, and G. Kar. Management issues in storage area networks: Detection and isolation of performance problems. In *IFIP/IEEE 9th International Network Operation and Management Symposium (NOMS'04)*, pages 593–604, Seol, Korea, mar 2004.

[5] A. Kochut and G. Kar. Managing virtual storage systems: An approach using dependency analysis. In *IFIP/IEEE 8th International Symposium on Integrated Network Management(IM'03)*, pages 593–604, Colorado Springs, Colorado, USA, mar 2003.

[6] T. H. Cormen and C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[7] J. Tate. Virtualization in a SAN. `www.redbooks.ibm.com/redpapers/pdfs/redp3633.pdf`, 2003.

[8] J. Tate, A. Bernasconi, P. Mescher, and F. Scholten. *Introduction to Storage Area Networks*. Number SG24-5470-01 in IBM Red Books. IBM, apr 2003.