MANAGING VIRTUAL STORAGE SYSTEMS: AN APPROACH USING DEPENDENCY ANALYSIS

Andrzej Kochut *
Computer Science Department
University of Maryland
College Park, Maryland 20742, USA
kochut@cs.umd.edu

Gautam Kar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
gkar@us.ibm.com

Abstract:

We present an approach for managing the performance of virtual storage systems by experimentally identifying the dependencies that exist between various components that comprise the system. Specifically, we show how one may profile dependencies between each logical volume exported by a storage system and components that this volume uses. To do so the technique estimates the arrival rate and size of requests issued to the internal system component as a functions of arrival rate and size of requests issued to the logical volume. The complete dependency profile of the system consists of a set of such functions for READ and WRITE operations separately and for each pair: logical volume - internal system component. The empirical technique of obtaining such profiles for typical existing storage systems is presented. We propose the use of Common Information Model (CIM) as a way to express dependency and performance information in an architecture-independent manner. The dependencies between components are computed as a fraction of bandwidth that is passed on to the sub-components. We discuss how the dependency profile of the system may be used to perform root-cause analysis and early Service Level Agreement violation notification. We also demonstrate the use of the method by applying it to a Linux system using software RAID.

Keywords:

Dependency analysis, virtual storage systems, root-cause analysis, Service-Level Agreement, systems monitoring.

1. Introduction

Fast growth in demand for big, efficient and reliable storage systems has led to the development of very complex and heterogeneous architectures. With this increase in architectural complexity and size of storage, new challenges for design, monitoring and management have emerged. The share of management costs in the Total Cost of

*Work done while author was an intern at IBM T. J. Watson Research Center.

Ownership for storage systems has increased steadily, creating a need for storage management systems that are scalable and autonomous. Management problems associated with storage systems start with the design of the system itself. The tasks of optimal allocation of data objects to logical volumes, configuration of software and hardware components, determining root-causes of problems, and predicting the violation of Service Level Agreement (SLA) prove to be very difficult. There is substantial amount of work done in the area of virtual storage design. The suite of storage design algorithms presented in [1] and [2] may be used to automatically design and configure a set of RAID arrays, given performance and reliability requirements of the request streams. In [9] authors show a way to automatically configure Storage Area Network to suit the needs of predicted data transmission. Difficulty in the design of the system stems from the trade-off between the utilization of components of the storage system and the nature of guarantees in the SLA used by the users of the system. Moreover, the design of the storage system is an ongoing process, because needs/demands of the users change constantly, making it necessary to re-evaluate storage requirements and reprovision storage allocation. Since, today, storage systems, in the form of SANs, are shared between multiple servers, and hence multiple applications, such reprovisioning is necessary to honor SLAs between the storage service layer and the applications.

In this paper we concentrate on aspects of performance problem detection and resolution in distributed environments consisting of virtual storage systems, e.g. SAN. The special emphasis of this research has been to understand how problems identified at the storage layer can be related to problems manifested at the application layer. The approach we have taken is to develop methods to characterize and compute dependencies that exist between virtual storage entities that applications and file systems use, and physical storage entities, such as RAID drives, into which the virtual entities are mapped. We propose modeling the dependencies between components of the system as a set of functions representing the arrival rate and size of requests issued to the internal component as a function of arrival rate and size of requests issued to the logical volume exported by the system. These functions are numerical representations of the storage policies implemented by the system. Storage policy defines the way data is stored and retrieved, what communication media are used to transfer the data, and what additional operations are performed while executing the request. For example, RAID1 ([4], [6]) storage policy defines on which physical devices the data is replicated, from where the data is read (load balancing), how the system behaves when one of the devices fails, etc. Figure 1 depicts a virtual storage system. Logical volumes *LV1* and *LV2* use a fiber switch to transfer the data to and from hard drives and virtual storage systems, such as IBM's Enterprise Storage Server (ESS). Data objects (*DO1* through *DO5*) are mapped onto logical volumes. Streams of requests (*S1* through *S6*) access the data objects.

Although it is sometimes possible to obtain an analytical model of the storage policy (one of the examples may be found in [8]), in general storage systems it may be very difficult to obtain a model that is close to reality. Thus we propose an empirical technique for estimating the dependency profile by applying controlled, variable load to logical volumes and observing the impact it has on internal system components. Our approach requires instrumentation of the system that can supply the management application with data about the component's performance. Because of the heterogeneity of the storage systems, the data needs to be expressed in a uniform, standard way to enable interoperability between components supplied by different vendors. To achieve

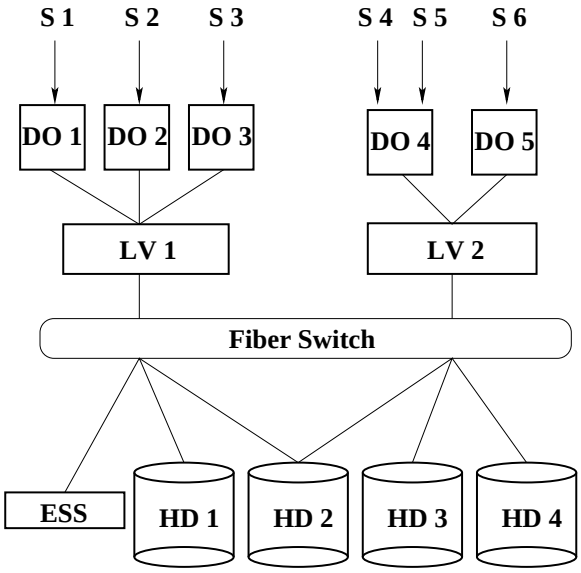

Figure 1. Example of a virtual storage system. Logical volumes *LV1* and *LV2* are mapped onto physical drives *HD1* through *HD4* and *ESS* using fiber switch. Data objects (*DO1* through *DO5*) are mapped onto logical volumes. Streams *S1* through *S6* access data objects.

this goal we propose the use of Common Information Model (CIM) as defined in [5]. CIM is a set of classes that may be used to describe system components as well as metrics associated with each component. By use of this standard, the management application may gather system-wide information about each component's performance and use it to prepare dependency profiles and later monitor the behavior of the system. Our method is similar to Active Dependency Discovery (ADD) technique presented in [3], where the authors perturb a system's components in order to estimate dependencies between them. However ADD is used for application-level dependencies while our method quantifies the performance dependencies between the elements of the storage subsystem.

The remainder of this article is organized as follows. Section 2 describes the way we model the system and use the model for computing dependencies. In Section 3 an example application of our method to the Linux Software RAID is presented. Finally, in Section 4 we discuss our future research plans.

2. Active Modeling Technique

Today's storage systems are usually heterogeneous, capable of storing terabytes of data and employing complex storage policies. A virtual storage system exports to its users a set of logical volumes that resemble physical devices. However, usually they are not mapped directly to the underlying physical devices in a simple one-to-one manner. Instead, the mapping is generally very complex. All these factors make a virtual storage system very difficult to design, model and monitor.

2.1 Model of the Virtual Storage System

We model the storage system as a set of independent components treated as "black-boxes" that execute requested actions by interacting with other components of the system. Each storage component presented with a request (i.e. READ or WRITE operation issued by another component or upper-level subsystem) performs internal computations and may issue requests to one or more other lower-level components. The request may be fulfilled in synchronous or asynchronous fashion. Moreover, we assume that each component may buffer requests and perform various optimizations on them before requesting processing from other components. For example, it is quite typical for storage systems to coalesce requests in order to increase sequentiality of physical disk access. In our example from Figure 1, the model of the system consists of eight components corresponding to two logical volumes, fiber switch, ESS and four hard drives. The way the components interact is decided by a set of policies that are defined for the system. For instance, policy of the logical volume *LV1* determines what actions are performed once the *LV1* is presented with the READ request. A simple scenario may involve issuing two READ requests to two hard drives and then transmitting the data using one of the channels of the fiber switch. However, in more realistic, real-world situations, the sequence of events may be considerably more complex. Various mirroring and striping techniques, multiple levels of caching, and other storage optimization strategies may make the process very tough to model and understand.

We believe that due to the complexity and heterogeneity of storage systems it is very important to have a modeling technique that could be applied with different levels of granularity. For example, we may want to model IBM's ESS either as a single storage element exporting storage space characterized by the size of the storage, its performance and reliability properties, or we may want to model it at a greater level of detail taking into consideration its internal structure. The choice between various levels of detail depends on available performance data as well as on the precision of predictions we want to make. Our model is flexible, making it possible to choose the level of detail that is suitable for a particular system. In a more elaborate scenario, instead of one component representing ESS, we can have many interacting components corresponding to elements of the drive arrays, cluster caches and other performance-related elements of the architecture.

The second part of our model consists of sets of metrics associated with each storage component. We identified three metrics as being crucial for understanding the behavior of a system's component: request arrival rate, size of requests, and request service time. An important metric that may be computed based on the above values is the utilization of a component. We use the standard definition of utilization as a multiplication of arrival rate and service time. The utilization indicates how close a given component is to becoming a bottleneck.

In order to make it possible to gather system-wide information in the heterogeneous virtual storage system we propose the use of Common Information Model (CIM) defined by Distributed Management Task Force. CIM is a common data model for describing management information. In particular, the standard contains a set of classes describing virtual storage subsystem as well as general framework for expressing metrics associated with system's elements. Each vendor (hardware or software) that contributed to the storage system that we want to monitor should provide the performance

metrics described in the standard. The management application may access the performance data of all hardware and software elements constituting the virtual storage system. In the example from Figure 1, manufacturers of hard drives, fiber switch, and ESS should provide specific software that could populate performance information in the CIM schema. Similarly, software vendors should provide corresponding performance information about logical volumes. We believe that existing software drivers may be easily extended with the performance metrics we require. In order to use CIM with our method it is sufficient to add instances representing the performance metrics to the class *MetricDefinition*. Modeling of components and their dependencies may be done using classes from *CIMDevice* schema. A good example of modeling the storage system using CIM may be found in [7]. In Section 3 we present an example of application of our method to the Linux software RAID. We demonstrate how to extend the information exported by each device participating in the virtual storage subsystem with performance metrics required by our method. The extension task proves to be easy and non-intrusive.

2.2 Dependency Discovery Technique

We describe a method to dynamically gather information about the way components of a virtual storage system interact with each other. Our method consists of three major stages: instrumenting the system, applying controlled load to the collection of logical volumes, measuring the metrics (identified in the previous section) associated with the physical storage volumes, and finally, analyzing the obtained data and preparing the dependency profiles.

In the instrumentation stage, each component that may influence the performance should export the metrics described in Section 2.1. The use of CIM makes it possible to have multi-vendor systems be monitored using a common data model. In practice, each vendor would supply its specific CIM provider (piece of code publishing the performance data in the common CIM data repository). We have implemented an experimental CIM provider for our Linux system described in Section 3.

The second stage is devoted to inter-component dependency discovery and quantification. In order to do so, we propose estimating the impact of each of the logical volumes on the physical system components. Given a logical volume V and a component C of the storage system, we want to measure the way a stream of requests applied to V affects the system component C. We model this dependency as a set of functions computing arrival rates and sizes of requests issued to component C in terms of arrival rates and sizes of requests issued to logical volume V. We call this set of functions a **dependency profile** of component C on logical volume V. In the general case, each type of operation issued to logical volume may cause an arbitrary operation on any other component of the system. However, in most of the storage systems the situation is simpler. The WRITE operation causes only WRITE operations to be issued to other components, and similarly the READ operation causes only READ operations to be issued to other components. However, it may be the case that the WRITE request issued to a logical volume causes a READ request to be issued to one of the components. For example, if the storage system uses indexing, the index may be read in order to obtain information needed to perform the WRITE operation. Thus the full *dependency profile* of a component C on logical volume V consists of four functions:

- $READ\_RATE(V, C, type, rate, size)$ - average rate of arrival of READ requests to the component *C*, given that requests of type *type* arrive at *V* with average rate *rate* and are of average size *size* and all remaining logical volumes remain idle.
- $READ\_SIZE(V, C, type, rate, size)$ - average size of arrival of READ requests to the component *C*, given that requests of type *type* arrive at *V* with average rate *rate* and are of average size *size* and all remaining logical volumes remain idle.
- $WRITE\_RATE(V, C, type, rate, size)$ - average rate of arrival of WRITE requests to the component *C*, given that requests of type *type* arrive at *V* with average rate *rate* and are of average size *size* and all remaining logical volumes remain idle.
- $WRITE\_SIZE(V, C, type, rate, size)$ - average size of arrival of WRITE requests to the component *C*, given that requests of type *type* arrive at *V* with average rate *rate* and are of average size *size* and all remaining logical volumes remain idle.

All of the above functions characterize the impact of requests issued to the logical volume *V* on the internal system component *C*. This interaction is determined by a number of factors, the most important of which is the storage policy of the storage system. This is a set of algorithms governing the way the data is stored and retrieved. For instance, in the example from Figure 1 the storage policy determines what operations are performed once the READ request is issued to the logical volume *LV1*. It decides from which physical drives the data should be read as well as what communication channel in the fiber switch should be used to transmit the data. Some of these choices may be random. For example, if *LV1* implements RAID 1 algorithm and uses drives *HD1* and *HD2*, then the system may randomly choose the drive from which the data should be read. It is quite common to have load balancing techniques that use randomization to maximize the parallelism of data access.

Complexities of the virtual storage systems make it very difficult to compute the above functions analytically. We propose estimating them by applying controlled variable load to components of the system. Assuming that all performance-critical elements were instrumented and the data is readily available for the management application using the CIM standard, we can observe the impact of request streams issued to the logical volume on components of the system. In this way, by varying the type (i.e. READ or WRITE), arrival rate, and size of requests issued to the logical volume, we may obtain sample values of the functions constituting the *dependency profile*. During the measurements we increase the load until the utilization of one of the components approaches 1. At that point we know that the maximum capability of the subsystem was reached, and sampling may stop. In order to obtain service time profile of the component, we apply varying load to this component and observe the variation of its performance metrics. By varying arrival request rate and size of requests, we may obtain samples of the service time function. Similarly, as in the case of computation of the *dependency profiles* we stop increasing load once one of the components becomes fully utilized.

After obtaining *dependency profiles* for each pair (logical volume - component) as well as service time functions for each component of the system, it is possible to determine the dependencies between elements of the storage system. We propose using effective bandwidth as a measure of dependency. More precisely, for each logical volume *V* we identify a set of components that *V* depends upon. This set may be identified by inspecting *dependency profiles*. If the size and rate functions for a volume-component pair are non zero, it means there is a dependency between these components. For each component *C* on which *V* depends, we compute the bandwidth demand put on component *C* by traffic coming into component *V* as

$$BW(V, C, type, rate, size) = WRITE\_RATE(V, C, type, rate, size) * WRITE\_SIZE(V, C, type, rate, size) + READ\_RATE(V, C, type, rate, size) * READ\_SIZE(V, C, type, rate, size)$$

As a measure of the **dependency strength** between logical volume *V* and component *C* we elected to use the coefficients obtained by first order linear regression applied to pairs: bandwidth incoming to *V*, bandwidth demand put upon component *C*. We believe, that the distribution of the bandwidth depends primarily upon the storage policy, and not the size and arrival rates of requests. Experiments that we have conducted with Linux Software RAID support this assumption. However, in a general case, it may be possible that the dependency between components changes with the load applied to the system (i.e. changes in the arrival rate or average request size of the stream accessing logical volume). In that case we have a dynamic dependency quantification updated each time the traffic pattern changes. However we believe that this situation is not likely in the real-world storage systems. Thus, in the article we assume that the dependency strength may be modeled using linear regression on bandwidth distribution.

2.3 Applications of the Method

Dependency information, as collected in the previous section, may be used to build a dependency graph. While monitoring a set of applications, e.g. a file system, if a performance deterioration is observed, it would be possible to narrow down the set of probable causes by traversing the graph. The root-cause of the problem may then be determined by analyzing the set of suspected components resulting from the traversal. In our interpretation, for pair of components *V* and *C*, a dependency strength of 1 indicates that full bandwidth is passed on to component *C*. Values higher than 1 suggest that additional traffic is generated in order to execute requests. Value of zero denotes lack of any dependence between *V* and *C*. Values higher than zero, but smaller than one imply partial dependence between components *V* and *C*. These interpretations may be used as hints while traversing the graph in search of the root-cause of the problem. Figure 2 depicts example dependency graph for system from Figure 1 for READ operation. Arrows denote dependency between components of the system. Values associated with arrows in the graph denote strengths of dependencies. For example, data object *DO2* depends on logical volume *LV1* with strength 1. *LV1* depends upon *ESS* with strength equal to 0.8, which means that 0.8 of the bandwidth of READ operations issued to *LV 1* is “handed over” to *ESS*.

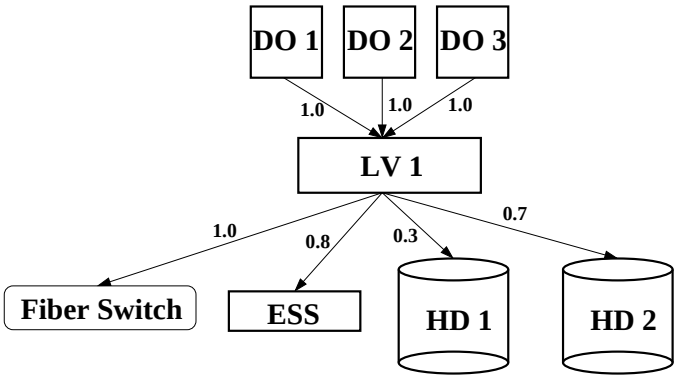

Figure 2. Example of the dependency graph for READ operation for the storage system from Figure 1.

Another application of the model is SLA violation prevention. Assuming that for a given system all of the performance profiles have been obtained, we can monitor each component of the system during its operation and alert the users (i.e. the operating system that uses the storage subsystem) about the possibility of SLA violation. Performance related SLA requirements are typically expressed as constraints on the maximum time spent by the system on processing a request, given that the arrival rate of requests and their sizes remain within the declared bounds. To forecast an SLA violation we propose monitoring the utilization of each component. Whenever the utilization of a component approaches one, the probability of queuing delays on this component increases. Using the dependency information it is possible to determine which data objects and requests streams may be affected by the delay and issue an alert to owners/issuers. The notification may be used by the users to throttle down the request rate or to take other preventive actions.

The third possible use of our model is data objects placement. Given an existing virtual storage system, the act of assigning data objects to logical volumes is a difficult one. The obtained *dependency profile* may be used to predict utilization of internal components given characteristics of request streams accessing the data objects and placement of data objects. Thus we may use our profiles as input to a verification stage in the optimization algorithm, searching the exponential space of possible placements. However, issues related to this application are beyond the scope of this article.

3. Method application to Linux Software RAID

To demonstrate the use of our method we present its application to an example virtual storage system. Our experimental system consists of Linux RedHat 7.1 running kernel 2.4.9. The virtual storage system consists of two hard drives and two logical volumes: RAID 0 and RAID 1. Figure 3 depicts this configuration.

Two physical hard drives *HDA* and *HDC* were partitioned into *HDA1*, *HDA2*, *HDA3*, and *HDC1*, *HDC2*, respectively. The first logical volume implemented software RAID 0 algorithm and used three partitions *HDA1*, *HDA2*, and *HDC1*. The second volume, implementing software RAID 1 algorithm, spanned two partitions:

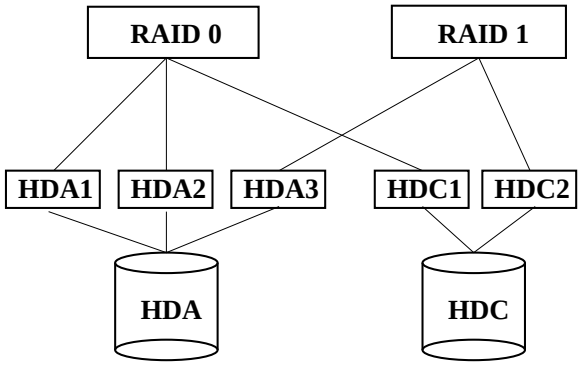

Figure 3. Linux software RAID architecture used for the experiment. Logical volume implementing RAID 0 algorithm spans partitions *HDA1*, *HDA2*, and *HDC1*. RAID 1 uses partitions *HDA3* and *HDC2*.

HDA3, and *HDC2*. To instrument the system we extended the standard Linux kernel statistics with metrics defined in Section 2.1. For each block device we obtained: average request arrival rate, average service time, and average size of the READ and WRITE requests (averaged over short intervals). The instrumentation we added is non-intrusive and relies on the data structures already present in the kernel. We believe that obtaining these metrics is quite easy for most of the storage architectures.

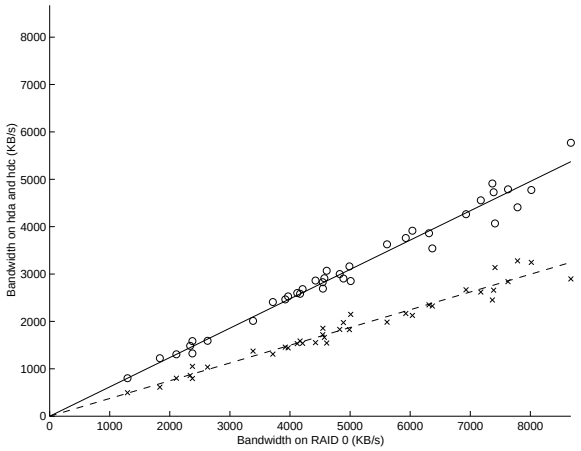

Figure 4. Dependency between logical volume RAID 0 and components HDA and HDC for READ operation. X axis represents bandwidth observed on logical volume RAID 0. Y axis denotes bandwidth observed on HDA (circles), and bandwidth observed on HDC (crosses). The solid line represents first-order linear regression model of the dependency between RAID 0 and HDA. Dashed line represents first-order linear regression model of the dependency between RAID 0 and HDC.

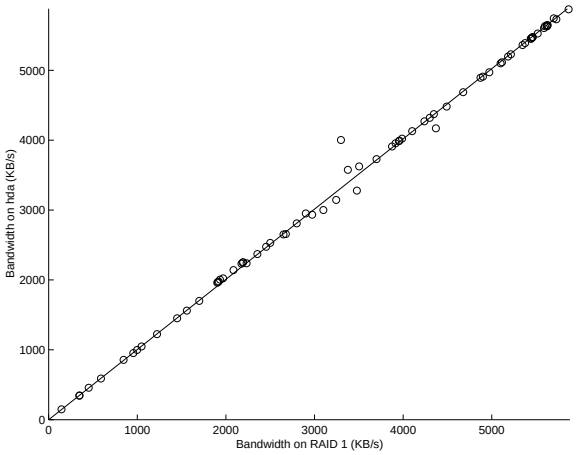

Figure 5. Dependency between logical volume RAID 1 and component HDA for WRITE operation. X axis represents bandwidth observed on logical volume RAID 1. Y axis denotes bandwidth observed on HDA (circles). The solid line represents first-order linear regression model of the dependency between RAID 1 and HDA.

Internal component	Logical Volume	
	RAID 0	RAID 1
HDA	0.62	0.65
HDA1	0.31	0.00
HDA2	0.31	0.00
HDA3	0.00	0.65
HDC	0.37	0.34
HDC1	0.37	0.00
HDC2	0.00	0.34

(a)

Internal component	Logical Volume	
	RAID 0	RAID 1
HDA	0.67	1.00
HDA1	0.33	0.00
HDA2	0.33	0.00
HDA3	0.00	1.00
HDC	0.33	0.99
HDC1	0.33	0.00
HDC2	0.00	0.99

(b)

Table 1. Estimation of dependency strengths for READ operation (a) and WRITE operation (b) obtained using our method.

The process of gathering the dependency information consisted of applying a controlled varying load to components of the system (one component at a time with remaining components being idle). Requests were randomly scattered over the volumes to minimize the effect of OS level caching. We developed an application that applied increasing load as long as all components of the system were under-utilized. First, we applied load to the RAID 0 logical volume. As a result we obtained samples of the rate and size dependency functions for various sizes and request rates arriving to RAID 0. The arrival rates and sizes of requests issued to all components as a result of the experiment were gathered. We used Matlab statistical toolbox to compute the first-order linear regression for the sample points representing bandwidth applied to logical volume and internal component. The resulting dependencies between RAID 0 and components *HDA* and *HDC* are depicted in Figure 3. Figure 4 shows the dependency between RAID 1 and component *HDA* for WRITE operations.

Similar sampling and regression modeling was performed for RAID 1 volume. The results are summarized in Table 1 (a). Values in the table denote the fraction of the bandwidth applied to the logical volume (columns) that were "passed on" to the internal components (rows of the table). For example, 0.37 of the amount of bandwidth observed on RAID 0 appeared on *HDC*. It may be seen, that the method precisely identified nearly equal split of work between three equal-sized partitions (*HDA1*, *HDA2*, and *HDC1*) caused by striping. Workload relayed to *HDA* was equally split among *HDA1* and *HDA2*. The load-balancing algorithm of Linux software RAID 1 showed bias toward the drive *HDA*. Results for WRITE requests and both logical volumes are presented in Table 1 (b). Again we can observe equal split of workload among all partitions participating in the RAID 0. As expected in the case of mirroring, full incoming workload of the RAID 1 volume was relayed to both partitions participating in the mirror. Presented results show that the method properly identified dependencies between components of the Linux software RAID.

4. Conclusions and Future work

This paper presents an empirical method for detecting and quantifying performance dependencies between components of the virtual storage system. We have proposed a flexible model and a set of metrics that may be used to quantify these dependencies. Use of the CIM standard enables our solution to be used in heterogeneous environment. We have described an empirical technique for quantifying the inter-component dependencies by applying controlled load to logical volumes exported by the system and monitoring the effect it has on the internal system components. We have proposed effective bandwidth as a measure of dependency and have showed an application of it in a Linux environment using software RAID.

The results reported in this paper are based on experiments conducted on a simple virtual storage system. One of the areas we are pursuing as future research work is to relate the dependency knowledge obtained at the storage layer to performance metrics measured at the application layer. In addition, our future research plans also involve investigation of the correspondence between workload applied to a logical volume *V* and utilization of internal system component *C*. It is possible to use the *dependency profiles* defined in this article and knowledge about service time of *C* to predict the utilization of *C* caused by the request stream applied to the logical volume *V*. Moreover, the total predicted utilization of the component *C* may be estimated as a sum of utilizations of all volumes that use *C*. This information will help a systems designer in the allocation of data objects with associated quality of service, such as data base table spaces, to virtual storage entities such a logical units.

References

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

[2] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: quickly finding near-optimal storage system designs. *HP Technical Report*, 2001.

[3] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. *International Symposium on Itegrated Network Management*, May 2001.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys, Vol. 26 No. 2, pp. 145-185*, 1994.

[5] DMTF. Common information model specification. *http://www.dmtf.org*, June 1999.

[6] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). *International Conference on Management of Data (SIGMOD)*, pages 109–116, 1988.

[7] J. Schott. Modeling storage. *DMTF System/Device Working Group, http://www.dmtf.org/*, 2001.

[8] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. *In Proc. of the Ninth International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS)*, August 2001.

[9] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: Automatic storage area network fabric design. *In Conference on File and Storage Technologies (FAST), Monterey, CA.*, January 2002.