Managing the File System from the Kernel

Shihabur Rahman Chowdhury*, Constantin Adam*, Frederick Wu*, John Rofrano*, and Raouf Boutaba†
*IBM TJ Watson Research Center
{schowdhu | cmadam | fywu | rofrano}@us.ibm.com

† David R. Cheriton School of Computer Science, University of Waterloo
rboutaba@uwaterloo.ca

***Abstract*—In this paper, we investigate the benefits of adding autonomic capabilities inside the operating system. We have developed and implemented a solution that focuses on three use cases (continuous file permission compliance, dynamic disk cleanup, and accidental removal protection) for the file system, and encapsulates all the respective file system monitoring, troubleshooting and error remedial operations in a Linux kernel module. The main benefits of this approach are the capability to detect issues instantly when they occur, and fix these issues transparently, with the invoking applications being unaware of their occurrence. These capabilities are not present in external agent architectures, including contemporary configuration management systems, like Puppet, Chef, or CFEngine. We have built a prototype and evaluated the performance of the most resource intensive use case, dynamic disk cleanup, using the FileBench and Postmark file system benchmarking tools.**

I. Introduction

In this paper we investigate the benefits of adding autonomic capabilities inside the operating system. We focus initially on making the file system self-managing. To perform our study, we have built a kernel module that allows the applications to express their access rights and disk usage requirements, and that autonomically checks and enforces these constraints on the underlying file system. The main advantage of this design, compared to external agent based architectures (i.e. contemporary configuration management systems, such as CFEngine, Puppet, or Chef, and other autonomic computing architectures, described in [1]), is its ability to detect any issues before they occur, take immediate actions to remediate them, and hide the occurrence of these events from the applications interacting with the file system.

Our system exposes an API which allows the developers to specify the disk cleanup and access rights policies for the files used by their applications. This design is in line with the DevOps idea of increasing efficiency and reducing the chance of failure through collaboration between developers and IT administrators. Before deploying applications in Production, IT administrators will test extensively the developer specified policies. This will reduce the chances of mis-configuration due to the lack of knowledge of application internals. Allowing the developers to specify the application policies will also reduce the configuration overhead of the applications once they are deployed.

Pushing more intelligence down to the operating system level also brings several important benefits to the management of large data centers. It streamlines the operation of the data center, by eliminating the need to monitor the file systems, raise alerts, or produce a large variety of tickets that cover issues such as adding or removing disk space, cleaning up disk, or detecting inappropriate access rights and modifying them. It also reduces human involvement in file system management tasks, leading to better resource utilization, less configuration errors, and protecting files that are essential to application operation from accidental deletion.

Our approach is inspired by the autonomic computing philosophy described in [2]. However, in contrast with [2], where each managed element (including the operating system) is monitored by an external agent (the M in the MAPE-K autonomic loop), we seek to add the autonomic management capabilities at the lowest possible level in a server: at the OS level. This design aims to eliminate the agents that monitor and manage the operating system, by including this functionality in the OS itself. For example, instead of monitoring if an application failed, the operating system can be instructed to keep it running, in the same way a service can be run using `initd`. A second example along these lines could be the elimination of `cron` jobs to periodically back up file systems. The operating system can be instructed to initiate backups whenever an essential file has been changed, or whenever a certain amount of change happens.

To further highlight the benefits of this design, consider the following two examples. First, files come with different access privileges. Changing these rights opens security loopholes. Using our approach, we can prevent any non-compliant access right change to take effect. To contrast, an external agent architecture will detect the access rights violation after it takes place, potentially leaving the system in a vulnerable state from the instant when the change takes place, until the moment when the external agent runs a new monitoring cycle, detects, and fixes that.

Second, in the UNIX world, we usually over-provision disk space, by carving volumes out of a larger disk, so that there is unallocated disk space to grow the volumes. Disk volume growth is usually a manual system administrator task. Even

The first author is a PhD student at David R. Cheriton School of Computer Science, University of Waterloo (sr2chowdhury@uwaterloo.ca). This work was submitted during his summer internship at IBM TJ Watson Research Center

more so, with hypervisors in the cloud, there is additional disk capacity to grow volumes. Consider a process that attempts to write a 16 GB core dump into a file system that has 2 GB free space left. If we use an unmodified operating system and an external monitoring agent this operation will fail, and an error code will be returned to the application. Depending on whether the application treats this operation as a transaction or not, the attempted write will abort, and the partial data written removed, or the disk will get full. Depending on the application's error handler, it will return an error message, or it may even crash. This creates more problems, and work because, in addition to understanding what went wrong, we are also now potentially left with the tasks of cleaning up the disk, and restarting the application. If the operating system is equipped with the capabilities of managing the file system by itself, it will detect the insufficient disk space before the write occurs, and will try several remediation actions (such as deleting or compressing files, expanding the file system, asking for more disk space from a hypervisor). If these actions are successful, the write operation will proceed, and the initial insufficient disk space situation will be completely transparent to the application. Our goal is for the OS to manage itself in such a way that applications rarely receive 'out of space' errors.

The rest of the paper is organized as follows. Section II reviews related work. Section III provides an overview of the system policies, use cases, architecture and the rationale behind our design decisions. Section IV provides a detailed description of the implementation. Section V provides an evaluation of the design. Finally, we conclude and outline future work in Section VI.

II. Related Work

Different research works performed in this field looked at adding self-managing capabilities at the operating system level, or building autonomic storage management solutions.

Self-managing capabilities have been built into various commercial and research operating systems. The SElf-awarE Computing model (SEEC) [3] allows developers to collaboratively create adaptive systems that understand user's goals and constantly monitor and re-enforce those goals. AcOS [4] is an Operating System that proposes an autonomic framework and demonstrates autonomic CPU allocation strategies. This work is done in the context of intelligent resource allocation to achieve user specified service-level objectives, while maintaining the CPU temperature under a threshold. NTFS, starting from Windows Server 2008 has self healing capabilities [5]: block level errors can be detected and corrected without user intervention. NITIX OS was the first commercially available OS [6] that claimed to have self-* capabilities. At the file system level, NITIX regularly performs a backup that would allow restoring the data in case of failures. Sun's ZFS [7] has self healing capabilities (can automatically restore data after a failure). It relies on a backup to restore data after some failure occurs. BORG [8] is a system that focuses on reorganizing the file blocks for better I/O performance. The Elastic Quotas file system [9] gives the users the illusion of having virtually unlimited disk space. To achieve this objective, it classifies files in 2 categories: regular and elastic. Only the regular files are accounted towards the quota limitation. The elastic files are subject to removal when the disk usage goes above a threshold, and the user exceeds his quota. The elastic files are managed through a duplicate directory structure and shadow users. It also provides the capability for the users to specify policies (i.e. remove files that are older than 60 days). The implementation is different from our work. The Elastic Quotas file system periodically scans and takes action. Instead, we only take action when required.

A number of research efforts have been made towards building autonomic storage management systems, and integrating them into the structure of data centers. The design and implementation of an autonomic storage manager is presented in [10]. It allows to specify allocation policies in terms of capacity and performance metrics. It also automatically raises alerts if these constraints are violated. This system performs resource allocation, by translating high-level policies into low-level commands, but it does not address in detail the self-managing or self-healing properties of an autonomic system. Nectar [11] is Microsoft's automated data and compute management framework for data center. Old data is automatically removed from the system and re-computation is avoided by leveraging the old results saved in the system. In [12] authors develop scc a storage configuration compiler for cluster applications that automates cluster configuration decisions based on formal specifications of application behavior and hardware properties. This compiler's ability to configure heterogeneous, rather than homogeneous cluster architectures, enables it to meet the application Service Level Agreements (SLAs) while achieving 2-4.5x cost savings. In [13] the authors propose Polus, a framework for policy based storage management. It removes the necessity to write codes that map the high level QoS requirements to low level device actions. Thus, reducing the complexity of the system administrators' jobs. Polus allows the SAs to express their requirements as a high level rule of thumb specification and learns about the system's conditions and quantifies these specifications to specific implementations. It also continuously monitors the system for QoS violations and performs the necessary actions to bring the system back to compliance.

III. System Overview

The autonomic file system manager presented in this paper has two major components. One component interacts with the user-space applications to setup policies that represent in a universal way the knowledge used to manage the file system. The second component interacts with the file system and implements the system behavior specified in the policies. In the rest of this section, we describe three use cases around which we have built our system, the policies and how they apply to the use cases, review the system architecture, and discuss our design choices.

A. Policies

We consider two types of policies: policies defined by users/applications, and system-wide policies. The former assign to the files disk cleanup categories, or access permission masks. The later configure the disk cleanup categories (e.g. set the maximum age of debug files as 2 days), and assign groups of files to a specific category (based on their location, or type). System-wide policies also define default rules (e.g. similar to 'umask' assigning default access rights to the files, a system-wide policy can maintain a mask of 'rwxr-x—' to all the files in a directory, or specify that world-writable files are not allowed on this file system). Finally, system-wide policies allow the system administrator to configure the policy precedence (e.g. define precedence between folder and group policies).

We have initially classified the files into four categories: required, debug, audit, and temporary. Each category has a set of disk cleanup rules associated with it. Once a user or an application assigns a file to a specific category, that file is by default entitled to the rules that apply to that category. Policies can be applied to individual files, to folders (same policy for all the files contained in the folder), or to groups of files, based on their type. Individual file policies override folder-wide or group-wide policies.

The significance of the disk cleanup categories is as follows. Files marked as required are essential for application operation, and they should never be deleted, under any circumstances. Temporary files are the first candidates for deletion. Temporary files (e.g. files used to install a package, or backup a database) have a shorter lifespan, and are used less frequently after the first use. Debug files (e.g. memory dump files) are similar to temporary files, but they can be kept for a longer amount of time to troubleshoot application, security, or performance issues. Audit files (e.g. log files) are usually kept for long periods of time, but they are accessed less, as they get older. After a specific period of time, these files can be compressed to save space.

A system-wide policy defines how to handle the files that have not been assigned to any disk cleanup category. These files can either be assigned to a default category (e.g. audit), or inherit the category of the folder containing it. An example of a policy (specified in JSON) is:

```
{
  "disk_cleanup_category" : "debug",
  "maximum_age"           : 2
}
```

This policy specifies that when a debug file is at least 2 days old, it can be deleted.

B. Use cases

Our system is capable of handling three types of problems, without requiring any human involvement: controlling access rights changes to files, keeping file system usage within specified boundaries, and preventing deletion of files which are marked as *required* by the applications.

1) Continuous File Permission Compliance: This use case is triggered when a user or application tries to change the permissions of files. While 'umask' sets up the initial permissions, nothing prevents the user from changing these permissions, until there is a security incident, or a failing audit. The file users or applications specify permission masks for the files they own. Our policies are more flexible than the initial umask, and can allow a range of permissible values, while preventing others. The autonomic kernel module discards any access rights changes that are incompatible with these permission masks. One way to detect this today is by scanning the entire file system and checking all the permissions, a potentially resource-intensive operation that impacts server performance. Our approach is not to allow this in the first place.

2) Dynamic Disk Cleanup: Dynamic disk cleanup reactively takes action when the disk usage violates existing policies. The remediation process involves three lines of defense: maintaining desired levels of free disk space, automatically handling out of space conditions, and raising alerts when everything else fails. This use case can be triggered by creation, deletion, or editing of files.

Best practice suggests maintaining a certain percentage of free space in file systems. Rather than having an agent monitor for file system utilization, we propose that the file system monitors itself as it manipulates files. In order to accomplish this, we introduce the minimum, maximum, and desired utilization thresholds (expressed as actual disk space or utilization percentage). The *minimum* threshold specifies the smallest size that a file system can shrink to. The *maximum* threshold specifies the largest size to which a file system can be expanded to. Passing the maximum threshold will always raise an alert to a higher level system. The maximum threshold ensures that file systems don't grow out of control. The *desired* free utilization threshold specifies the amount of free space that should be maintained in the file system. When a write operation reduces the amount of available disk space below the desired threshold, the autonomic kernel module launches an asynchronous disk cleanup. If the disk cleanup fails to free enough space, then the file system will be expanded to accommodate the write operation, while also maintaining the desired free space.

When the file system usage exceeds the desired threshold, the autonomic kernel module tries to remediate the situation first by deleting the files that exceed a policy-specified age and are marked as not required, and second by expanding the file system. If that is not enough, other file systems are cleaned up, and an attempt is made to shrink them to their minimum and make space to allocate to the expanding file system. The autonomic component raises an alert if all remedies fail. If the utilization goes below the minimum threshold, the autonomic component shrinks the file system to reclaim disk space that can be used in the future to expand other file systems.

In order to determine the list of files to delete, the kernel module provides an interface to the applications and users, through which files, or folders (and their entire contents) can be assigned to the four categories described in section

III-A (required, debug, audit, and temporary). In addition, the autonomic kernel module complies during the disk cleanup process with all the system-wide policies in place, such as identifying folders or file groups that can be assigned to one category, or defining a global age when files of a given category can be removed.

3) Accidental Removal Protection: The third use case is triggered when a user or application tries to delete a file that was previously marked as *required*. Although this capability is already in place (a file that is marked as 'immutable' in Linux cannot be deleted unless the 'immutable' attribute is unset manually), it can be very easily implemented using our proposed framework. For example, our manager can set the 'immutable' flag on all the files that are declared as required by the applications or users.

C. Architecture

The autonomic kernel module shown in Figure 1 implements the core functionality described in the use cases above. Instead of modifying the file system to handle these use cases, we followed the concept of stacked file systems [14], and bundled this functionality into a kernel module. The autonomic kernel module is placed on top of the Linux Virtual File System (VFS), and overwrites a subset of VFS system calls. First, it checks if a condition that triggers a use case occurred. Next, it tries to take remedial actions if required. Finally, it passes the execution control to the original system call.

The autonomic kernel module also has an interface that allows the applications and users to specify their file management policies. This interface is implemented with the help of a virtual device, as described in section IV-B. Applications register their policies by sending control commands to this device, which are interpreted by the autonomic kernel module. We also provide a set of shell commands that use the aforementioned interface for users to register their policies.

D. Design Rationale

The main objective of our work is to demonstrate the benefits of pushing autonomic capabilities all the way down to the operating system. The main benefit of an autonomic file system is the ability to detect any issues right before they occur, take immediate actions to remediate them, and make these events transparent to the file system operations.

For example, consider a process that attempts to write a 2 GB file to a file system which has 1 GB space left. The file system manager should intercept the write operation, detect the insufficient disk space condition, and take the necessary actions (such as deleting/compressing files, or even expanding the file system) before passing the control back to the file system.

The design choices we considered were: writing a user-space program, changing the kernel, or implementing a kernel module. A user-space program needs to know about any file system changes before taking action. The only way to accomplish that is to subscribe to the iNotify kernel subsystem [15]. In this case, the file system events are captured after they occurred, and remedial actions cannot be taken transparently. For example, in the disk full scenario, the write system call will fail before the file system manager is given the opportunity to clean or expand the disk. That leaves us with the remaining two options, which both run in the kernel space. As the overhead of changing and rebuilding the entire kernel is much higher than that of developing a separate kernel module we chose the former option. However, for deployment in Production systems, a kernel enhanced with these functionalities would be a better option, indeed. As opposed to solutions based on periodic monitoring, our approach has the advantage of immediately detecting events and taking remediate actions. We demonstrate the effectiveness of taking these reactive actions in section V.

As discussed in section III-C, the autonomic kernel module is developed on top of VFS. This allows our module to work with a wide range of file systems, rather than restricting its usability to a specific file system implementation. The portability, however, comes with a performance cost, compared to a solution that is implemented in the file system.

Our design moves configuration decisions from the application users to the application developers. The rationale behind this is that the application developers have a better understanding of how the applications work and their resource requirements. Another important benefit of this approach is that the application development process goes through extensive testing, so the file system management configuration set by application developers is much less error-prone that a similar solution provided by a user.

IV. Implementation

To show the effectiveness of our approach, we have built a prototype of an autonomic file system manager, and implemented it as a loadable Linux kernel module. We chose to implement the prototype in a kernel module, rather than modifying the kernel itself, because a module can be rapidly developed, built, tested and experimented with. We chose Linux as the operating system because it is open source, and it exposes a set of low level routines to write loadable modules. In this section, we discuss the implementation of the prototype: its interfaces with the file system and the user-space programs, its policies and its storage subsystem.

A. File System Interface

The file system management process is activated upon detection of events that change file and directory permissions, or create, remove, or edit files. To achieve this behavior, we have modified the kernel system call table, to point to our modified implementation of each system call that triggers any of the events mentioned above.

We have modified the implementation of each system call that changes the state of the file system by adding a policy compliance check, and a set of remedies to be applied if the compliance check fails. After these additional steps are completed, the call proceeds with its normal execution. For example, when a *write* request is detected, the autonomic

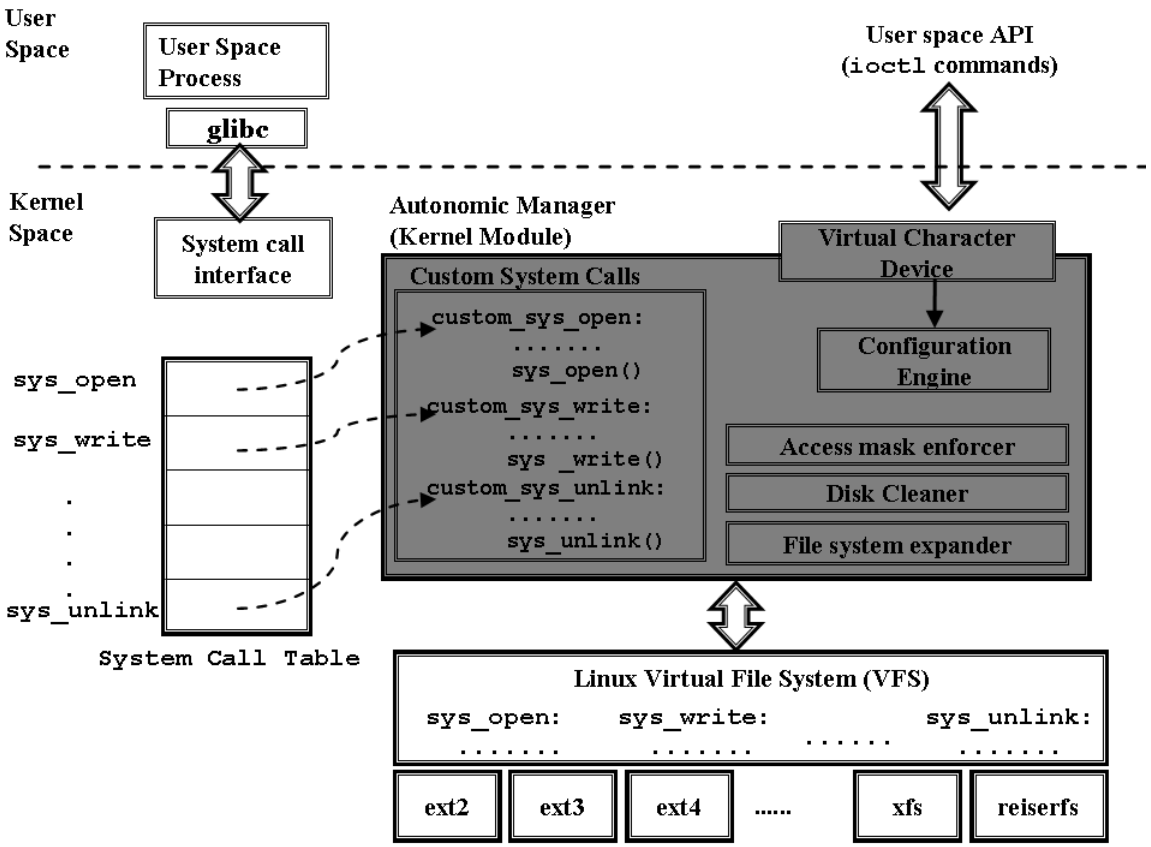


Fig. 1. System Architecture

manager retrieves the file corresponding to the file descriptor parameter, and the file system where this file is located. Then it checks if the utilization of the file system after the write is compliant with the usage policy (i.e. the amount of free space is still above the desired threshold). If the compliance check fails, the manager attempts to apply automated remedies (clean up or expand the file system). After the automated remedies complete successfully, the original *write* system call is invoked to write the file on the disk.

To accomplish this we modified the kernel's system call table during the module's bootstrapping. The entries corresponding to our desired system calls in the table were modified to point to our provided implementation. Once a system call is intercepted, the low level routines of VFS exported by the Linux kernel are used to read file system changes and error conditions are figured out.

Most of the actions to read the file system state and changes, as well as to remedy error conditions are performed through the VFS interface. Exceptions to this rule are the operations of expanding or shrinking a file system, which are performed using Logical Volume Management (LVM) tools. The LVM commands are spawned from the kernel module to perform modifications to the volumes on which the file system resides.

B. User-space Interface

The user-space interface enables the communication between the autonomic file system manager on one side, and applications and users on the other side. Users and applications use the interface to send to the autonomic manager the policies that define the file system usage requirements. The interface comes in two different flavors, as described below.

First, a configuration file stores both system-wide and directory / file specific policies. The autonomic kernel module loads this file during bootstrapping, and sets its internal state accordingly. During steady-state operation, the autonomic manager can detect any changes made to the file system. Upon detecting a change to the configuration file, the manager automatically reloads its policies. By editing this configuration file, a user can change the manager behavior on the fly, without restarting it.

Second, user-space applications can use an API to the autonomic kernel module to express their file system usage requirements. We implemented this API with the help of a virtual device, by exploiting the fact that a user process can send low level I/O commands to any device (through `ioctl` system call) along with their own parameters. Our autonomic kernel module registers a virtual device[1] with the

[1] a physically non-existent device

OS during bootstrapping. Any `ioctl` command issued by a user process to this virtual device is intercepted by the `ioctl` implementation provided by our autonomic kernel module, which interprets the commands as configuration commands for itself, and sets its configuration parameters accordingly. In our reference implementation the following configuration options are allowed from a user-space program:

- Specify permission mask for a file or directory
- Categorize a file or directory into one of the four proposed categories

C. Policies

Policy authoring, storage and management tools have been already extensively studied in various contexts [16]. Developing a full policy framework is outside the scope of this paper. Therefore, we assume the existence of a policy delivery platform that allows to propagate updates to all the managed servers. We also adopt a very simple policy specification format, JSON for specifying system-wide and user-defined policies in our reference implementation. These policies are persistently stored in the disk as configuration files and are loaded during bootstrapping.

V. Evaluation

A. Evaluation Setup

We have run our experiments in a Ubuntu 13.04 Linux Virtual Machine, with an ext3 file system, 2 Virtual CPUs, 3 GB of RAM, and 30 GB hard disk. The virtual machine is hosted on an NTFS file system. We have partitioned the hard disk into a 15 GB system partition, and a 15 GB partition allocated for the measurements. We have used Logical Volume Manager (LVM) to create file systems with various initial sizes on the experimental partition. To ensure fairness between different methods and between each run, the experimental file system was formatted prior to each run.

We have used FileBench [17] and Postmark [18] two of the most popular file system benchmarking utilities (as shown in [19]) to evaluate the load that the autonomic file system manager puts on the system, as well as the manager's effectiveness in keeping the file system utilization within bounds. During the evaluation process, we took into account the limitations of these tools. First, both FileBench and Postmark are micro-benchmarking tools that put a short-term load on the system to measure its I/O performance, but are not able to emulate long-term steady state behavior, a feature needed to perform a realistic study of disk cleanup and disk expansion processes. However, given their features (most notably the capability to generate various workloads, and their wider acceptance), we decided to use these tools, rather than developing our own custom test suite.

We used FileBench to measure the overhead that the autonomic manager imposes on the system when no disk cleanup or disk expansion procedures are triggered. In this case, the overhead still occurs, because the file system manager intercepts each write system call and performs checks on the file system. In fact, this will comprise the majority of the total introduced overhead, because it happens on a much faster time scale than the disk cleanup or expansion activities. Filebench provides the capability to generate workload following a number of predefined profiles. Each profile has different read / write ratios, and represents the I/O pattern of some real life application. For the purpose of our studies we have run the experiments using three predefined profiles – simulating the workload of a file server, and a web server.

We have used Postmark (which emulates the workload of a mail server) to measure the effectiveness of the autonomic file system manager in keeping the utilization within bounds. We created a scenario where the file system utilization grows on a fast tune scale, and observed the way in which the disk cleanup and expansion procedures operate. For these experiments, we have set an upper bound of 90% for the disk utilization.

We ran each experiment for 20 minutes with a 30-second statistics collection interval and took the best result for 3 runs. We configured the average file size, the number of files and the I/O size used parameters for both FileBench and Postmark according to the guidelines in [20].

B. Evaluation Metrics

To determine the overhead that the autonomic manager imposes on the system, we have measured the averages of throughput (specified in I/O operations per second) and CPU time per I/O operation, as calculated by FileBench at each statistics collection interval.

To measure the effectiveness of the autonomic file system manager, we captured the file system utilization and size over time in 5-second intervals.

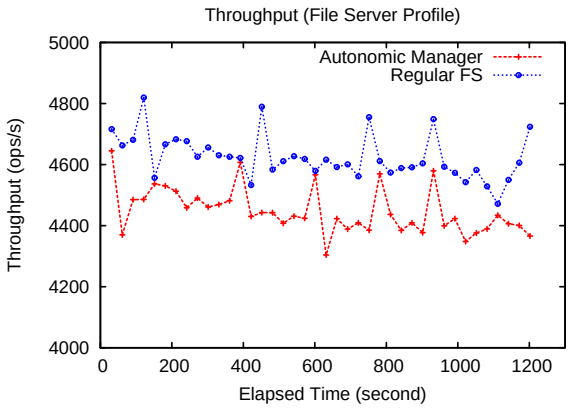
We performed both experiments with and without the autonomic manager activated, and compared the results.

C. Evaluation Scenarios

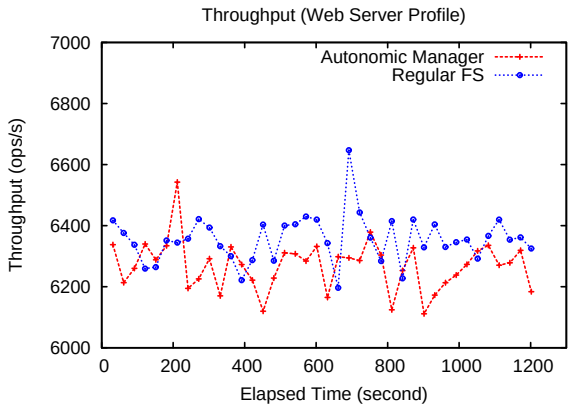
Two evaluation scenarios are presented. In the first scenario, we monitor the performance overhead of the file system manager while it is running in a sufficiently large file system, where no disk cleanup or disk expansion procedures are triggered. In the second scenario, we start with a smaller sized file system, and perform disk cleanup and disk expansion operations, as soon as the utilization crosses a threshold (90%). We examine the disk utilization against time to see how well disk cleanup and expansion work.

D. Results

From the figures 2 and 3 we estimate the overhead imposed by the autonomic manager at about 10%. This overhead comes from the write operations. We introduce an additional read operation in each write system call. If we estimate the overhead of this read I/O operation to be about 20% per write system call, we get an overhead of 10% when half of the I/O operations are reads and the other half are writes. We make no changes to the read operations, and therefore there is 0 additional overhead. We provide a similar explanation for the better performance of the autonomic file system manager for the web server workload. In the case of the file server

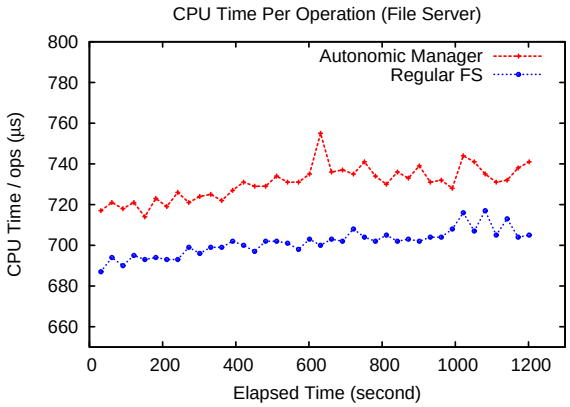


(a) File Server

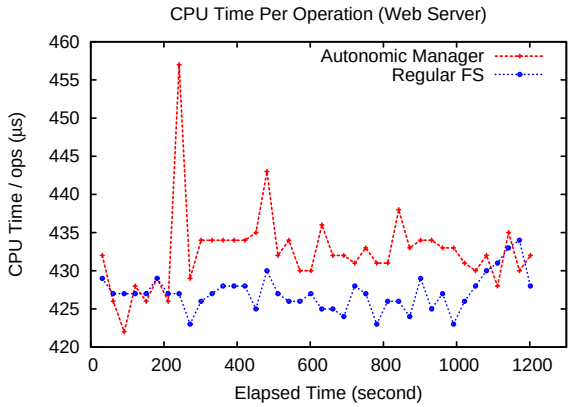


(b) Web Server

Fig. 2. Throughput



(a) File Server



(b) Web Server

Fig. 3. CPU Time

workload, about half of the I/O operations are reads and half are writes. For the web server workload, the ratio between reads and writes is about $10:1$ ([20]), and hence our system performs better in this case.

Figure 4 shows that the system behaves as expected in the case when the file system utilization increases over time. If we are given an elastic bound for expansion, the autonomic manager can accommodate writes without causing the processes to fail, while a regular file system fails as soon as it runs out of allocated space.

VI. Future Work

In this paper, we have investigated the benefits of placing the file system manager inside the operating system kernel. We have designed and implemented a solution that encapsulates the file system monitoring, troubleshooting and error remedial operations in a Linux kernel module. The main benefits of our approach are the capability to detect issues instantly when they occur, and fix these issues transparently, without the invoking applications being aware that they occurred. These capabilities are not present in external agent architectures, including contemporary configuration management systems, like Puppet, Chef, or CFEngine. We have investigated the performance and overhead of this solution.

In order to deploy this solution in a Production environment, we need to address a number of research challenges. First, we need to guarantee the soundness of the policy specification, as any holes in this specification can lead to a security breach. Second, we must be able to associate applications with policies, as well as with the resources to which these policies apply. For example, an application should not be able to mark files that do not belong to it as ”temporary”. Also,

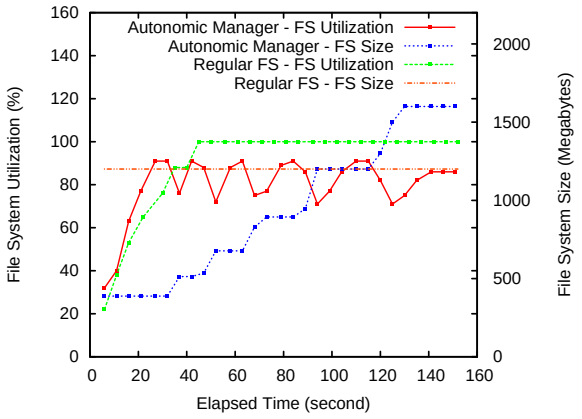


Fig. 4. File System Utilization

when an application is uninstalled, all the policies specified by that application should be revoked. Third, while our study thus far focused on the file system, it could be extended to manage other system entities, such as processes, memory, or CPU. Fourth, an autonomic operating system represents one layer in an automation architecture. Although the autonomic management capabilities within the OS ensure its smooth operation by tackling the error conditions by itself, a higher layer component in the automation architecture can look into the root cause of the errors and take preventive measures to stop these from occurring at the first place.

In conclusion, this work is a step in investigating how to design data center management processes that default to automation and only involve humans when everything else fails. We call this approach *Extreme Automation*. By adding autonomic management functions inside the operating system, we aim to implement Extreme Automation management processes that are *scalable* with the growth of the cloud, *continuously* monitor the state of the system to detect any issues, and *transparently* fix these issues before they turn into errors that perturb the system operation. While our initial work is focusing on making the file system self-managing, we think that the concept can be expanded to other parts of the operating system as well.

REFERENCES

[1] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing – degrees, models, and applications," *ACM Computing Surveys (CSUR)*, vol. 40, no. 3, p. 7, 2008.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: A framework for self-aware computing," MIT CSAIL Technical Report, MIT-CSAIL-TR-2011-046, 2010.

[4] D. B. Bartolini, R. Cattaneo, G. C. Durelli, M. Maggio, M. D. Santambrogio, and F. Sironi, "The autonomic operating system research project: achievements and future directions," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 77.

[5] "Self-healing NTFS in windows server 2008 and windows vista," http://blogs.technet.com/b/apawar/archive/2008/02/14/self-healing-ntfs-in-windows-server-2008-and-windows-vista.aspx.

[6] "NITIX - autonomic linux-based server operating system," http://www.iccci.com/images/Nitix_Letter_lo.pdf.

[7] "Oracle solaris ZFS administration guide," http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/.

[8] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reorganization for self-optimizing storage systems," in *FAST*, 2009, pp. 183–196.

[9] O. C. Leonard, J. Nieh, E. Zadok, A. Shater, J. Osborn, and C. P. Wright, "The design and implementation of elastic quotas: A system for flexible file system management," 2002.

[10] M. Devarakonda, D. Chess, I. Whalley, A. Segal, P. Goyal, A. Sachedina, K. Romanufa, E. Lassettre, W. Tetzlaff, and B. Arnold, "Policy-based autonomic storage allocation," in *Self-Managing Distributed Systems*. Springer, 2003, pp. 143–154.

[11] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: automatic management of data and computation in datacenters," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–8.

[12] H. V. Madhyastha, J. C. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat, "scc: cluster storage provisioning informed by application characteristics and slas," *FAST'12*, 2011.

[13] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease, "Polus: Growing storage qos management beyond a 4-year old kid," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX Association, 2004, pp. 31–44.

[14] J. S. Heidemann and G. J. Popek, "File–system development with stackable layers," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 1, pp. 58–89, 1994.

[15] "inotify - monitoring file system events," http://linux.die.net/man/7/inotify.

[16] R. Boutaba and I. Aib, "Policy-based management: A historical perspective," *Journal of Network and Systems Management*, vol. 15, no. 4, pp. 447–480, 2007.

[17] "Filebench: Filesystem benchmarking tool," http://sourceforge.net/projects/filebench/.

[18] J. Katcher, "Postmark: A new file system benchmark," Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, Tech. Rep., 1997.

[19] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, "Benchmarking file system benchmarking: It* is* rocket science," *HotOS XIII*, 2011.

[20] P. Sehgal, V. Tarasov, and E. Zadok, "Evaluating performance and energy in file system server workloads." in *FAST*, 2010, pp. 253–266.