

# Robustness Hinting for Improving End-to-End Dependability

Michael W. Bigrigg

**Abstract**—File systems make unreasonable attempts to provide data to the point that they will block an application instead of passing the error on to the application to handle. Transient problems such as network congestion or outages and heavily loaded systems or denial of service attacks can lead to failure-like situations. Alternative mechanisms have been developed for the file system to trade performance for robustness in an attempt to always guarantee full availability of data. These mechanisms may not be necessary, as the application programmer may have already accounted for such situations. By hinting to the file system the application's ability to handle errors it is possible for the file system to make better resource allocation decisions and improve end-to-end dependability.

**Index Terms**—Reliability, Filesystems, Program Analysis.

## I. INTRODUCTION

WE will show that for I/O intensive applications on a potentially failure-prone environment:

(1) Some I/O applications can handle errors on their own and do not need additional support. Some applications do not handle I/O errors and will silently process garbage and produce an incorrect result.

(2) There are techniques to increase the chance that data storage is available. However, these techniques are resource intensive and have a negative impact on other applications running on the system.

(3) By hinting to the file system the application's ability to handle I/O errors it is possible for the file system to make better resource allocation decisions. This will reduce the impact on other applications when it needs to be avoided and improve an application's end-to-end dependability.

As a motivating example, consider a data intensive program such as a data mining application. After performing several hours of data processing, the remote file server becomes unavailable and no further data can be read. Provided the file

system does not try indefinitely to retrieve data thereby blocking the application, the application could store its partial result locally in order to resume processing later.

## II. THE FILE SYSTEM DEPENDABILITY WALL

File systems make unreasonable attempts to provide data by blocking an application from continuing when it may not be necessary. Transient problems such as network congestion or outages and heavily loaded systems can lead to failure-like situations when it is not possible to perform the entire requested operation, yet the problem is only transient and could still provide a partial result. The semantics of application calls into a file system actually provide for cases where not all data is available and where failure conditions do not exist. A true failure condition for a file system is a request for data where no data is available such as a read past the end of a file. This is different from a situation where data would be available, but is currently not accessible such as when using a mobile device.

The C I/O library `fread` call will return up to but no more than the number of bytes that have been requested [10]. A return value of 0 does not signify an error condition, just that no data is currently available such as at the end of the file. It is a negative return value that signifies an error. This information is to be passed to the calling code to be handled. It would then be the calling code's responsibility to handle the problem.

But the distributed file system such as the Network File System (NFS) or Andrew File System (AFS) does not trust that the application will handle the error and instead will block until all data is available. From the file system's perspective, dependability is maintained, as it does not return an error. The file system has not explicitly caused the application to crash, but through negligence forces the user to terminate the program ungracefully. From the end-to-end perspective, we have a very undependable system.

If we are attempting to measure an application's performance and it crashes moments after it starts, we do not consider the application to be very fast. Similarly, if we are attempting to measure an application's dependability and it waits forever, we should not consider the application to be very dependable.

## III. APPLICATION DEPENDABILITY

In our FlakyIO project [1], we built a test harness to allow us to selectively manipulate the behavior of the I/O function calls of a program to simulate file system failures using fault

This work was supported in part by DARPA/ATO's Organically Assured and Survivable Information Systems (OASIS) program (Air Force contract number F30602-99-2-0539-AFRL) and by the Pennsylvania Infrastructure Technology Alliance.

Michael W. Bigrigg is with the Institute for Complex Engineered Systems at Carnegie Mellon University, Pittsburgh, PA 15213 USA (telephone: 412-268-76680, e-mail: bigrigg@cmu.edu).

Text Utilities	fread()	fopen()	fclose()	fwrite()	read()	close()	putchar()	ungetc()	printf()	fprintf()	vfprintf()
cksum	S	HC	HI						S	S2	S2
comm		HC	HC	S						S2	S2
csplit		HC	HC	S	HC	HC				S	
cut		HC	HI				S	S		S2	S2
expand		HC	HC				S			S2	S2
fmt		HC	HI				S			S2	
fold		HC	HC	S						S2	
head		HC	HC	HC	HI	HI				S2	S2
join		HC	HI	S			S			S2	
md5sum	S	HC	HC				S		S	S2	S2
nl		HC	HC	S					S	S2	S2
paste		HC	HC	S						S2	S2
pr		HC	HI				S			S/S2	S2
split		HC			HI	HI				S2	S2
sum		HC	HC				S			S/S2	S2
tr		HC		HC	HC					S2	S2
tsort		HC	HC						S	S2	S2
unexpand		HC	HC				S			S2	S2
uniq		HC	HC	S						S2	S2

### GNU Text Utilities I/O Error Handling

injection. We applied the testing to a specific class of applications that are I/O based, use the C standard library, and have been around for years, the GNU textutils. Our testing harness performed source code modification to insert a fault injection layer into the application. Faults were injected into a single execution of the application based on the specific function call. To fail the fread call, a -1 was passed as the return value to the calling application and it is then up to the calling application to check the return value to determine if an error has occurred or if the file is at its end. If the data is unavailable and the application does not check its return value, the application will process whatever garbage data was in memory that was set aside for the fread result.

We ran the applications on a failure free environment and then again inserting faults and observed their behavior. The results fell into four groups: handles correctly (HC), handles incorrectly (HI), silent failure (S), and silent error (S2). Many applications handled error conditions correctly, the HC group. For instance, all correctly handled the failure of an fopen call. Some applications acknowledged that an error did occur but misreported the error, the HI group. This was due to the engineering of the testing system that emulated transient failures instead of catastrophic failures. A transient failure would only fail for a specific instance of a function call but would subsequently not fail for the next instance of an I/O function call. An application acknowledged an fread failure but used an feof function call to determine if the file is at its end. The read call will fail but the feof call would not. An additional coupling of function calls is for silent errors, the S2 group. Most applications used a print in order to report an error and did not check to see if the error had been correctly reported. It may seem as if printing an error message is the

only recourse, the applications in the S2 group did not return an error code to the operating system to report that the application had failed. The last group, S, silently processed the result as if it were correct and did not report any failure. While in our previous paper we focused on the failures, here we would like to focus on the successes.

Looking across the row of the application's error handling capability, its success or failure to handle I/O errors would contribute greatly to its dependability measurement. While not substantial for a quantitative metric, a qualitative analysis would conclude that the tr application which correctly handles all potential I/O errors would at least make it more dependable than an application such as cksum which results in a silent failure when faced with an fread error. The tr application, while possessing the capability to be more dependable than cksum, will be forced to have the same dependability rating (when measuring its ability to handle I/O errors) because most file systems will not allow the cksum to handle the error.

Looking at the fwrite column there are nine applications that make use of the fwrite call. While seven of the nine applications suffered silent failures when an error is propagated from the file system, there are two applications that would successfully be able to handle an error. These two applications, head and tr, should be allowed to handle the error.

However when error checking is absent, it is then necessary for the system to adapt itself during a system error, trading computational or other resources for robustness. The file system has several alternatives such as re-requesting data again, assuming that if a transient failure has occurred it may be able to retrieve the data in a second attempt. There may also be other replicas of the data on the network or other

means of obtaining the data, so it may be possible to contact a different location. Several projects attempt to be adaptive to changing availability of data such as survivable storage systems [2][5][13] and storage systems for mobile users [11][12]. All of these alternatives that demand additional resource consumption should be used appropriately and not when code provided by the programmer exists expressly to handle the case of limited data availability.

#### IV. APPLICATION DEPENDABILITY ANALYSIS

A hinting mechanism similar to what has been done for file prefetching [9] would pass information from the application itself to the file system. A robustness hint would be passed for each I/O function call that had the correct error handling code. While optimally automated hinting is desired, preliminary hinting would use the manual insertion of hints into the application code. The argument against this is that if it is possible to know where the problem lies and the source code is available, it is possible to fix the code. However the insertion of hinting code, which only requires the programmer to acknowledge the problem, would be far easier than the modification of the program logic. In addition, the hinting is added only where there is handling of error conditions allowing the file system to default to its current behavior of blocking should no hint be given.

It is true that programmers often overlook error checking, as they believe that the various errors are just inconceivable and do not need to be checked or are overwhelmed with the task of identifying all possible error situations. Unhandled error conditions would lead to potential software failures when the underlying system cannot satisfy our requests. Programmers alone cannot thoroughly account for all error conditions, as it is a non-trivial task with a lack of tool support. But as shown in the previous section while error checking is difficult, it is not always missing.

There are two major approaches used for the detection of software faults. These systems are typically aimed at providing information to the programmer in order to modify the program source to eliminate software faults. This information would be used by the programmer to insert hints into the code.

The first group of fault analysis techniques, similar to what was using for our FlakyIO system, was to run the entire program or a subset of the program to observe its behavior. Errors are introduced to examine how the software behaves. This approach makes use of external faults being passed into the application that will cause it to fail. Such approaches include: fault injection through random memory corruption, passing values typically known to cause exceptions into an individual software module through its software interface [8], or the creation and use of a comprehensive test suite. In particular it is possible to identify the type of input or condition that has led to the fault, but it does not typically identify any remedial action that should be taken by the application making it an ideal candidate for providing hint information.

The second approach for the detection of software faults is compile-time analysis. It attempts to identify program features

that would cause a program to behave improperly. This analysis focuses on a particular characteristic that is typically the base cause of faults such as portability [7]. Other approaches use programmer-defined models of acceptable behavior to drive the analysis [3][4]. These analysis techniques report the results to the programmer for correction presupposing that the programmer has the time and capability of rewriting the offending code to correct it also a good candidate for hinting information. The analysis is done using the static analysis that is a part of any optimizing compiler. The problem is that there are a lot of code constructs that defies analysis such as pointer-to-function. The default behavior of optimizing compilers is to not perform the optimization and for us the same holds true.

If it is not possible to determine if the code properly handles the error, no hint is given to the file system and its default behavior of blocking until the file system is back online holds.

#### V. CONCLUSION

Distributed file systems currently take the worst-case situation when confronted with a lack of data availability and block until data is available. File systems should only take exceptional measures to ensure robustness when robustness measures are not handled by the application. We showed that some I/O applications can handle errors on their own and do not need additional support. Some applications, on the other hand, do not properly handle I/O errors and will silently process garbage and produce an incorrect result therefore need the file systems additional support. By allowing the application to hint its ability to handle I/O errors it is possible to increase the end-to-end dependability.

#### ACKNOWLEDGMENT

We thank the members and companies of the Parallel Data Consortium (including EMC, HP, Hitachi, IBM, Infineon, Intel, LSI Logic, Lucent, MTI, Network Appliance, Novell, PANASAS, Platys, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support.

#### REFERENCES

- [1] Michael W. Bigrigg and Joseph Slember. Testing the Portability of Desktop Applications to a Networked Embedded System. Workshop on Reliable Embedded Systems, October 2001.
- [2] Aaron Brown, David Oppeneimer, Kimberly Keeton, Randi Thomas, John Kubiawicz, and David A. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. Proceedings of the 7<sup>th</sup> Workshop on Hot Topics in Operating Systems (HOTOS-VII), March 1999.
- [3] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. Proceedings of the 2000 OSDI Conference, October 2000.
- [4] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. SIGSOFT Symposium on the Foundations of Software Engineering, December 1994.
- [5] Armondo Fox and Eric A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. Proceedings of the 7<sup>th</sup> Workshop on Hot Topics in Operating Systems (HOTOS-VII), March 1999.

- [6] Samuel P. Harbison and Guy L. Steele, Jr. *C A Reference Manual*, Prentice Hall, 1987.
- [7] S. C. Johnson, lint, a C Program Checker, Computer Science Technical Report Number 65, 1978.
- [8] Philip Koopman. Towards a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security. Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA'98), November 1998.
- [9] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Proceedings of the 15<sup>th</sup> Symposium on Operating Systems Principles, December 1995.
- [10] P.J. Plauger. *The Standard C Library*. Prentice Hall, 1992.
- [11] M. Satyanarayanan. Coda: A Highly Available File System for Distributed Workstation Environment. Proceedings of the Second IEEE Workshop on Workstation Operating Systems, September 1989.
- [12] D. B. Terry, M. M. Theimer, K. Peterson, A J. Demers, M. J. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. Proceedings of the 15<sup>th</sup> Symposium on Operating System Principles (SOSP-15), December 1995.
- [13] Jay Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, Pradeep K. Khosla. Survivable Information Storage Systems. IEEE Computer, August 2000.