NVIDIA Data Verification Testing

A Major Qualifying Project Report

submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____

Rhyland Klein

_____

Matthew Pedro

_____

Thomas Rybka

Date: 2 March 2006

Sponsor: NVIDIA, corp.
Liaison: Allen Martin

_____

Professor David Finkel, Advisor

Abstract

The aim of the NVIDIA Data Verification project was to develop a testing utility that would stress the drivers of NVIDIA's hard drive controllers. The tool created, nVerify, does so using highly configurable methods of interaction with the drivers. This creates a breadth as well as a depth of stress on the drivers. The tool is designed to be extensible and scriptable for automation making it easily introduced into NVIDIA's testing cycle.

Executive Summary

As computing technology continues to grow, the amount of data in the world is expanding at an even quicker pace. With this surge in data, storage solutions play a very large role in the progression of computing technology. Present storage solutions, such as hard disk drives, adapt to meet the needs of the computing world. These adaptations lead to more complexity in the disks themselves and the controllers that operate them.

NVIDIA is a growing developer of such hard drive controllers. The drivers that interface the controller in their chipsets and the operating system also play a very important role. Not only do they provide a layer of abstraction from the machine-level commands that are interpreted by the controller, but in some cases, the driver software can also be responsible for some of the features of the NVIDIA products.

This project was to design and implement a tool for testing NVIDIA's storage software. Our tool, nVerify, had to stress the drivers in many ways to expose errors in the software before being released. nVerify is a highly configurable tool capable of creating conditions that will attempt to stress the operation of hard drive device drivers. It has been designed to be extensible and modular so that it can be ported to other operating systems.

nVerify is capable of providing multithreaded operation and creating many outstanding requests for disk access. By configuring the number of threads and the number and type of I/O requests, a user can specify the various types of strain that it wants to place on the driver. The tool will output a detailed analysis of the failure including a data dump and the various parameters of the I/O request. This information is used to diagnose and fix problems in the driver software.

By performing data verification testing, drivers for hard drive controllers can be verified to perform the proper functionality in many different situations. Using nVerify to stress the drivers, developers and consumers can have greater confidence in the drivers running their hardware. Such confidence is necessary to be established with each change in the hardware itself or the software interface, as developers attempt to increase reliability and performance.

TABLE OF CONTENTS

ABSTRACT ..... **I**

EXECUTIVE SUMMARY ..... **II**

1 INTRODUCTION **1**

2 BACKGROUND **3**

2.1 PHYSICAL DISKS 3
2.1.1 Drive Geometry *3*
2.2 DISK ACCESS 5
2.2.1 Raw disk access through Operating System API *6*
2.3 THREADS 7
2.3.1 Threading in Windows 32-bit *8*
2.3.2 Mutex *9*
2.4 XML 11
2.5 CURRENT TOOLS AND THEIR DRAWBACKS 12
2.5.1 Hazard *13*
2.5.2 WinThrax *13*
2.5.3 IOmeter *14*

3 FEATURES **15**

3.1 EXHAUSTIVE FEATURES 15
3.2 MQP SCOPE 16
3.2.1 Alpha *16*
3.2.2 Beta *16*
3.2.3 Final Release *17*

4 PROCESS **18**

4.1 DESIGN 18
4.1.1 Major Data Structures *18*
4.1.2 Description of Modules *21*
4.2 FLOW CHART 26
4.3 IMPLEMENTATION 29
4.3.1 Incremental Development *29*
4.3.2 Managing the Workload *30*
4.3.3 Testing *31*

5 RESULTS **35**

5.1 TESTS 35
5.2 SAMPLE OUTPUT 36
5.2.1 Screen Output *37*
5.2.2 Log Output *37*
5.2.3 Debugger Output *39*
5.3 PERFORMANCE 39
5.3.1 Memory Usage *39*
5.3.2 Speed / Number of I/O's Performed *42*
5.4 BUG DETECTION 42
5.4.1 Generated Bugs *42*
5.4.2 Unexpected Bugs *43*

6 CONCLUSIONS AND FUTURE WORK **46**

6.1 GOALS SET AND ACCOMPLISHED 46
6.2 POSSIBLE IMPROVEMENTS/ FOLLOW-UP WORK 47

APPENDIX A: FEATURE DOCUMENT **51**

INTRODUCTION 51
EXHAUSTIVE FEATURE LIST 51
SCOPE OF WPI PROJECT 52
Alpha *53*
Beta *53*

APPENDIX B: API 54

IOCONTROL (USER ←→ APPLICATION) 54
DEVICE I/O 54

APPENDIX C : NVERIFY USER'S GUIDE 57

SETUP 57
The Configuration File *57*
The Pattern File *60*
EXECUTION 61
Running a Test *61*
Incorrect Parameters *62*
Log File *62*

APPENDIX D – NVERIFY DEVELOPER'S GUIDE 65

BRUTE THREAD 65
CATCHER 67
CONFIG PARSER 68
DEVICEIO 70
DPHANDLER 72
ERROR GENERATOR 73
INTEGRITY TESTER 75
IOCONTROL 76
IT MODULE 78
NALLOGATOR 79
PATTERN 80
SCRIBE 81
THREADMANAGER 83
THREADWRAP 84
TRIAGENURSE 85

TABLE OF FIGURES

Figure 2.1 Moving-head disk mechanism [1, pg. 37] 4
Figure 5.1: High Thread Count Stress Test 36
Figure 5.2: Sample Output 37
Figure 5.3: Sample Log File 38
Figure 5.4: Memory Usage and Disk Queue During Execution 40
Figure 5.5: Release Memory Usage 41
Figure 5.6: busTRACE Showing Driver Bug 44
Figure 6.1: A Sample Request 58
Figure 6.2: An Example Job 59
Figure 6.3: An Example Configuration File 60
Figure 6.4: An Example Pattern File 61
Figure 6.5: Log Entry on Failure 63
Figure 6.6 : Log of Succesful Run 64

1 Introduction

As computing technology continues to grow, the amount of data in the world expands with an even quicker pace. With this surge in data, storage solutions play a very large role in the progression of computing technology. Present storage solutions, such as hard disk drives, adapt to meet the needs of the computing world. These adaptations lead to more complexity in the disks themselves and the controllers which operate them, and a greater chance for something to go wrong.

A device driver forms the layer in which the operating system interacts with such hardware. It is the middle-man, so to speak. Developing device drivers requires an exhaustive knowledge of a given platform's hardware and software. A fault in a driver, operating at such a low level in a system, most often leads to system-wide failure. It is imperative to remove these errors prior to the driver software ever being used in a production environment. In the development phase of the drivers, testing is an integral step to increase the likelihood that the latest versions of their software have no bugs at all.

By performing data verification testing, drivers for hard drive controllers can be verified to perform the proper functionality in many different situations. By thoroughly exploring various command paths, and input/output operations, driver errors can be exposed and rectified. Confidence must be established with each change in the hardware itself or the software interface, as developers attempt to increase reliability and performance.

Our project was created to develop a new tool that NVIDIA’s software storage team could use to stress test their nForce drivers to check for errors. It will perform large

amounts of I/O operations on a hard disk and will be highly configurable to allow for the developers to customize runs to test certain parts of the driver.

2 Background

This section covers the material that we had to research to understand different aspects of our project. The topics include physical disk knowledge so we understood how the hard drive worked and was structured. We also had to learn some of the Win32 API calls that we would use in our project to do low level disk access and synchronization.

2.1 Physical Disks

We had to learn how a physical disk is organized so that when we did raw disk I/O (Input/Output) we would understand what the values we had to use meant and how they related to the physical disk itself.

2.1.1 Drive Geometry

Disk drives consist of a relatively simple mechanism. A read-write head moves across the surface of platters which are covered in a magnetic material. Generally there exist two heads per platter, one on each side. A disk arm moves all of these heads simultaneously over the platters of the disk. The geometry of a platter is what defines how the data is stored in a magnetic disk such as a hard disk drive. Conceptually, the difference between a hard disk drive and a floppy disk is the actual consistency of the platter used. [1, pg. 491]

The platters are divided into cylinders, which is the area of each platter which can be accessed without moving the heads. A cylinder is a cross section of a disk, consisting of a circular strip from each side of each platter. If you position the heads at a single radius from the center of the disk, they can access one cylinder by rotating the platters.

The circular strip on a single platter is the area which one head can access, and is called a track. Tracks are divided into a number of slices called sectors, which are the smallest parts of the disk which can be read or written at one time. [2] See Figure 2.1 for a visual representation.

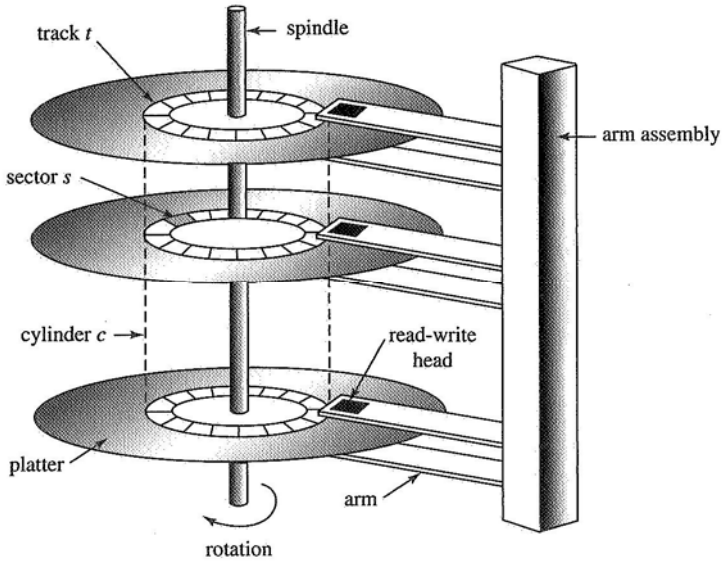

Figure 2.1 Moving-head disk mechanism [1, pg. 37]

Therefore, the geometry of the disk is specified as:

- The number of cylinders that the disk contains;
- The number of tracks per cylinder (same as the number of heads, or twice the number of platters);
- The number of sectors per track; and,
- The size of each sector (in bytes), usually 512 bytes. [2]

For an example, one of the test drives that we used reported the following information:

- Cylinders: 19458
- TracksPerCylinder: 255
- SectorsPerTrack: 63
- BytesPerSector: 512

Multiplying these out, 512 bytes/sector * 63 sectors/track * 255 tracks/cylinder * 19458 cylinders equals ~149.01 Gigabytes – the exact capacity of the disk. Please note that the

number of tracks per cylinder would imply approximately 128 platters in the disk drive. This is a logical geometry which most motherboards' BIOS can understand. There are usually only 3-4 platters in a disk drive, but the disk's controller does a translation to overcome limitations such as the numbers of sectors per track so that the BIOS can understand.

2.2 Disk Access

Disk drives can be abstracted as large one-dimensional arrays of logical blocks, the smallest unit of transfer. The size of such a block is usually the size of one sector on a disk, which is usually 512 bytes, and the array is mapped to sectors sequentially (i.e. the first item in this array is sector 0, the first sector on the first track on the outermost cylinder). [1, pg. 491-492]

A disk is generally low-level formatted at the factory, and the sector size can be chosen here, but some operating systems can only deal in sector sizes of 512 bytes, so that is the usual convention. To use a disk to hold files, the operating system must implement its own data structures on the disk. First, it partitions a disk into one or more groups of cylinders. As far as an OS is concerned, each group of cylinders can be treated as one disk. The next step is the creation of a file system, known as logical formatting. Here, the OS stores the file system data structures it will use, such as map of free and allocated spaces and an empty directory. The OS provides methods and APIs for creating files and reading, writing, and seeking within them. A file system may provide special services such as buffer cache, file locking, pre-fetching, space allocation, file names, and directories. These services can prove to be cumbersome for some applications, and the

ability to use the disk as the large sequential array that it inherently is can be more useful. [1, pg. 499]

Raw I/O is the ability to access the raw disk as the sequential array of sectors that it is comprised of. The block-device interface captures all the aspects necessary for accessing disk drives. The devices understand commands such as *read()*, *write()*, and if it is random-access, *seek()*, to specify which block to transfer next. Applications normally access such a device through the file-system interface. The OS itself, and special resource intensive applications, like database management systems, may prefer to access a drive as a simple linear array of blocks, and therefore perform direct disk access, or raw I/O. [1, pg. 469].

2.2.1 Raw disk access through Operating System API

Most modern operating systems provide some API or manner of accessing disks in a system via raw I/O. Different operating systems provide different abstractions and methods for accessing raw drives, and you must be familiar with their APIs in order to correctly store and retrieve data. The Windows XP operating system and other NT variants provide a layer of abstraction over the raw disk and its sequential array of blocks to make it appear as though it were one single file. With one file, a program can access any block on the disk by seeking to a position in the disk, and reading from or writing to it. This provides the same functionality as if you accessed the blocks on the disk by using an array with an index. [3]

Windows NT provides an API for accessing block-devices. It uses the *CreateFile()* function to provide a handle for any kind of block-device, in this case, a physical drive. With a file handle, the API provides the *ReadFile()* and *WriteFile()*

functions for reading and writing to a specific location. *SetFilePointer()* is then used to seek to a different location to perform I/O. The seek capability allows you to choose any particular byte to begin reading or writing to, however this can be problematic. When performing direct drive access, or raw I/O, you must be careful to seek, read, and write in multiples of the sector size supported by the drive, and the OS. The API provides the *DeviceIOControl()* function to query a drive and determine the number of bytes per sector, number of sectors, sectors per track, and so forth, so that you can compute the right location to seek to, as well as allocate a buffer which is a multiple of the sector size. [3]

2.3 Threads

A thread is defined as a "lightweight process, [it] is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack."[1, pg 129] A single process can spawn multiple threads associated with it and many processes do. A process, by comparison is "a program in execution." [1, pg 93] A thread does not maintain a large amount of resources like a process does and shares its memory locations with other threads created from the same process and running on the same sections of code, data sections as well as other resources like open files and sockets. This sharing of resources and the low overhead of creating threads is one of many benefits of using them over multiple processes.

2.3.1 Threading in Windows 32-bit

This section outlines the use of threading in 32-bit Windows that was specific to our project. We discuss how we created threads and what calls we used to do so. Understanding how threads are created and used is integral to understanding our project.

Including the windows.h header file in a project, one will have access to the Windows 32-bit functions that are used to create and manage threads. It is possibly to use a function in this library called *CreateThread()* to create a thread as many times as one may need. This function makes it possible to pass in a pointer to a data structure that the thread will use. Once some threads have been created it is possible to wait until they complete. The operations to do the previous are described below.

In Windows 32-bit, the function call to make a thread is called *CreateThread()*. When you create a thread with this function a handle is returned which can be used to check on the status of the thread. You can pass in NULL for LPSECURITY_ATTRIBUTES to use default security, and 0 for the stack size and creation flags specify the default. The thread method that is actually created and used is the “lpStartAddress” which is cast to (LPTHREAD_START_ROUTINE). The LPVOID lpParameter is the pointer to the data structure that you want the thread to have access to.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpsa,
  SIZE_T  cbStack,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpvThreadParam,
  DWORD fdwCreate,
  LPDWORD lpIDThread
); [4]
```

The data structure that the thread accesses is usually created with *HeapAlloc()*. It is important to remember when you use *HeapAlloc()* to make sure that it is deallocated with HeapFree(). If you do not do this there will be memory leaks.

An important aspect of threads is that you can wait for them to be completed. This is done using the *WaitForSingleObject()* function and *WaitForMultipleObjects()*. These two functions take in similar arguments. The *WaitForSingleObject()* is also used with mutex, for more information refer to section 2.3.2 Mutex.

```
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
); [4]
```

WaitForSingleObject() and *WaitForMultipleObjects()* both signal when their respective handle(s) signal that they are completed. Both functions have a great feature in that you can specify the wait timeout. If you specify “INFINITE” it will block until the thread(s) are complete. If you specify a number say 500, it will wait 500 milliseconds then timeout, should you need to do any type of polling.

2.3.2 Mutex

Mutex is short for mutual exclusion which is a type of protection used on code that is responsible for modifying data that will be shared between different threads or processes. Mutex allows only a single process or thread to enter the code and perform its operations to avoid conditions when a value might change when a thread time-slices in the middle of a segment of code. Unexpected changes, due to concurrent threads both modifying the same shared data structure, can cause unpredictable errors which are hard to catch.

Mutex works by setting a status value in a variable. Each thread or process that wants to access a mutexed segment of code has to first check the value of the mutex to see if it is already been changed. If it has, then that process/thread has to wait until the mutex is reset to the “free” value. When it is changed, all waiting threads/processes are signaled and then the first one to change it back to “busy” is then inside and the others have to continue waiting.

In Windows the call to create a mutex instance is:

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName
); [4]
```

The parameter of type LPSecurity_Attributes is a structure passed in to specify the permissions of the mutex, if NULL is passed in, then the default parameters are used. If the calling thread/process wishes to take ownership of the Mutex after creating it, then the bInitialOwner parameter should be true. Lastly, when creating a mutex, you give it a name so that it can be differentiated, no name is necessary and NULL is a valid parameter. *CreateMutex()* returns a handle to the mutex created.

WaitForSingleObject() is used when using a mutex to check if it is free and then reserving it if it is. When using this function, a timeout of INFINITE should be used to block all but 1 thread/process from accessing the mutex. For more information about *WaitForSingleObject()* refer to section 2.3.1 Threading in Windows 32-bit.

The function used to signal a mutex that a process/thread is finished with it is the:

```
BOOL ReleaseMutex(
  HANDLE hMutex
); [4]
```

This function takes in the handle of a mutex and sends a signal to all processes/threads waiting on this handle that it is ready to be re-allocated.

The final function used, when all of the threads/processes are finished accessing a mutex, is the *CloseHandle()* method. This function takes in the handle to a mutex and closes it preventing any further access to it.

```
HRESULT CloseHandle(
   HANDLE hObject
); [4]
```

2.4 XML

XML was created by the W3C. The W3C define XML as “Extensible Markup Language, abbreviated XML, [which] describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them.”[5] XML is similar in form to HTML because both use tags but each serves a different purpose. XML uses tags like HTML, but the tags are used to describe what the data contained in the file is, not how it should be presented. XML files must be written so that they are well-formed. Well-formed means that for every opening tag, there is a matching closing tag. XML tags are defined by the designer in a file called the XML Schema. The schema defines the tags used in XML documents and defines how they can relate to each other, such as which tags can contain other tags of a certain type, and how many of these tags it may contain. The classic XML example is the book example, which defines what a book might look like in XML tags.

```
<?xml version="1.0" encoding="UTF-8"?>
<book isbn="0836217462">
 <title>
  Being a Dog Is a Full-Time Job
 </title>
 <author>Charles M. Schulz</author>
 <character>
  <name>Snoopy</name>
  <friend-of>Peppermint Patty</friend-of>
  <since>1950-10-04</since>
  <qualification>
    extroverted beagle
  </qualification>
 </character>
 <character>
  <name>Peppermint Patty</name>
  <since>1966-08-22</since>
  <qualification>bold, brash and
tomboyish</qualification>
 </character>
</book>
```

Figure 2.2 : Example XML Schema
(*from http://www.xml.com/pub/a/2000/11/29/schemas/part1.html*)

One of the key features of XML is that it is used in applications to store data in a format that other programs can easily decipher and use. Since XML is a structured language with all tags defined in a separate XML Schema file, the XML files using that schema can be read in through an interpreter, parsed, and loaded into a program with relative ease.

2.5 Current Tools and Their Drawbacks

There are two primary tools NVIDIA currently uses for data verification testing of their hard drive controller drivers, and one tool they use for performance testing measurements. Hazard is one of the tools used for data verification, which is a proprietary tool created by Hewlett-Packard. WinThrax is another tool used for data verification. The performance tool used is called IOmeter and was developed by Intel. The purpose of our project was to replace these tools with a much more comprehensive tool, specialized to meet the needs of the developers who write and design the nForce chipset drivers.

2.5.1 Hazard

Hazard is a tool created by HP, and licensed under very strict terms. Allen Martin, in the nForce chipset division, described the terms of use as “temporary licensing for a small number of copies only after which HP found a problem with our chipsets.” Under these terms, Hazard is a tool which is ineffective for use for NVIDIA. Without knowing about the specific capabilities, the Hazard tools major drawback is its unavailability for testing while developing the drivers.

2.5.2 WinThrax

WinThrax was created by AT&T in 1992. It is a 16-bit Windows application which does not take advantage of current-day 32-bit standards, and by no means advanced technology such as 64-bit processors. In fact, the 16-bit technology that it was written for is a hindrance, because it must run in some degree of emulation in a Windows environment. WinThrax performs a basic copy and read test between two hard drives which the user specifies. Unfortunately, it provides no room for customization of the tests, nor does it describe what tests it does run. WinThrax provides a very minimal graphical interface which renders it simple and easy to use. Use entails merely selecting a drive to write data from and a drive to write data to. Selecting the *Start* menu item will begin the tests. In our test use of the program, we did not discover errors, and cannot speak of the error reporting it does. WinThrax has negligible error reporting. Allen Martin stated that he wanted extensive information gathered on a failure so it can be used to troubleshoot what went wrong. It grossly under-reports the free/total size in megabytes of the disks. During the test phase, it reports the elapsed time in days, hours and minutes. It also reports the number of units copied, and compared, and the status (working, paused,

etc.). While this tool provides a means for data verification, it is easy to see why it may not provide the stresses of the system, as well as the configuration capabilities to be able to fully test the nForce chipset drivers being developed.

2.5.3 IOmeter

IOmeter is actually two programs, one called IOmeter and the other dynamo. IOmeter is the graphical interface used to make the settings for dynamo which does all the performance work. The combined program performs tests to write to specific drive(s) and then reports on the performance statistics. It is interfaced so you can perform disk, network, and CPU performance tests. The major drawback of this program is it reports how good your driver speed is but you have no idea if it was writing the correct information. IOmeter is therefore useless as a data verification tool. It was useful to the project team because it is under the open source license and provided similar functionality and processes as to what we needed for our project. It was explained to us that a later version of our project may include performance statistics as well as the data verification.

3 Features

This section describes all the features of the data verification program we wrote and includes the features that NVIDIA would like to eventually see implemented in their finished tool. Keeping all these features in mind throughout the design helped keep the structure extensible. The features are broken up into an exhaustive feature list and two subsets of that list, the features of our Alpha release and the features of our Beta release.

3.1 Exhaustive Features

The WPI team project was a subset of a larger verification tool. NVIDIA had asked the team to come up with a complete feature list for the final program. The final program would support multiple OS's. The WPI Team only worked on one particular part of the final programs goal. NVIDIA thought it would a good idea to keep a feature list with the end-goal of their final program in mind. This feature list was to include support for many different operating systems as well as performance measuring. The Team was not expected to support all of the list. The complete list of features can be found in Appendix A. Some functionality that the final product could have that was out of scope of this project is:

- Collection of performance statistics
- GUI user interface
- Arbitrary commands
- CPU affinity

3.2 MQP Scope

The scope of the WPI's team project was less than that of the final program goal. The WPI team had to implement the program the most common operating system, Windows. To accomplish this goal we had three separate releases that are mentioned below. For a complete list of the exhaustive features, refer to Appendix A in the exhaustive feature list section.

3.2.1 Alpha

The features for the alpha-level release were minimal at best. The alpha version just had the structure and flow of control of the main program. It created and passed the necessary structures and data it needed between the different modules of the program. This provided for a solid framework to work off of later. For a list of complete functionality of the alpha version refer to the Appendix A in the Alpha section. A basic summarization of the key points of our alpha version is:

- Single OS support, win32
- Basic functionality with program flow
- Control and configurable data patterns
- Logging of results

3.2.2 Beta

The beta-level release was "close to final release except that the bugs may not be worked out." The beta level release had full functionality but some things were not working at the time. For a list of complete functionality of the beta version refer to

Appendix A in the Beta section.

- Multi-threaded
- Ranges of where it wrote on disk
- Randomness of requests and IO locations

3.2.3 Final Release

After the beta version was complete, there was two weeks left for debugging the beta to make it into the final release. The final release included an XML log file, the ability to specify different patterns (user-made or pre-canned), and the ability to set different configurations.

4 Process

The goal of our design was to make a modular, extensible system for nVerify that abstracts the system level and OS specific functionality. The modular design of our project made the code organized and structured in a meaningful way. This modular design also made the project easily extensible by modifying the internal pieces without having to modify the other modules. The abstraction was achieved by using replaceable wrapper classes and defining our own variable types to mimic OS-specific variable types that we needed. This abstraction is essential for porting nVerify to different operating systems, because all of the functionality and program flow will work flawlessly with the program interface of the replaced modules.

4.1 Design

One of the most important parts in the development of our project was the design. We had to do a lot of thinking, discussing and testing to determine what would be a good design and as with most software development, our design changed rapidly and consistently.

4.1.1 Major Data Structures

Our project uses data structures to pass relevant information between different objects. Each of these objects has a specific purpose and is used in specific places. These data structures range in use from storing the data pattern information to storing the results of a read/write pair.

4.1.1.1 Pattern

The pattern structure is used to store pattern information that is obtained from either a pattern file (.ptn) or a pre-made pattern that is specified by the user. This structure contains both the data for the pattern itself, the size of the pattern, and the name of the pattern. The data stored in a pattern is repeated as many times as needed to fill a given size buffer when an I/O is performed. This structure will be used everywhere except for the interface module, catcher module, and the device I/O. This structure will be created in the Data Pattern Handling module, and passed to the main DataVerification module. One pattern will exist in each Job structure.

4.1.1.2 Job

The job structure is used for storing the separate test information that the threads will use. There can be multiple jobs per test and all of the multiple jobs are stored together inside the test structure. Each thread is run using only one job, and each job contains a list of references to the requests it would like to use in its operation. With only one thread per job it is the threads responsibility to handle the requests inside the job. The job structure is used primarily in the Brute thread module, where the relevant information needed later for verification and analysis is stored into the result data structure. The job structures are created in the ConfigParser module when the data is read in from the configuration file.

4.1.1.3 Request

The result structure is used to specify specific I/O operations in different test configurations. A request specifies the range to write data to, the size of the buffer to

write, and the randomness of the addresses chosen to write to. They also specify if there is an address increment between writes. Each job will have one or more request associated with it.

4.1.1.4 Test

The test structure is used to encompass the entire configuration file, as well as command line options. There is only one test structure per execution of nVerify, with references to it being passed to the various modules for control flow. The test structure holds all of the job and request information, the runtime of the test, the log filename, the seed for random number generation, and the other command line options that are stored.

4.1.1.5 Result

The result data structure is used to store the outcome of a specific I/O operation. They are created in the Brute thread module and then sent to the Integrity Tester where it is copied. There is one result structure for each write and its associated reads. When the read(s) is/are completed for a write the result structure is filled with the I/O information as well as other statistics such as runtime and the number of I/O's completed at that time.

4.1.1.6 Log Entry

The log entry is used for storing the information that will be sent to the log for output to file. This structure is used in Integrity Testing and Triage mode and passed to the logging module to be logged to the xml file. There is one log entry for each job/thread.

4.1.1.7 Summary

The summary structure is used for storing the end results from the entire test. It is passed from the main module to the log file. It stores information about the test as a whole such as the runtime of the entire test, the files used for the log and the configuration. It also outputs information like the overall result of the test, whether it was a success or not, and the date at which the test was initiated.

4.1.2 Description of Modules

Each of the modules in our program performs a certain task. The modules range in purpose from getting input from the user to performing the disk I/O operations and sending the results to another module. Our modules were designed so that if a programmer wanted to change the functionality of a single part, the change can be consolidated into one module.

4.1.2.1 Main (DataVerification)

This controls the main functionality and program flow of nVerify. It is responsible for getting the parameters from the user and initiating the configuration file parser, loading the data patterns, initializing the catcher module, and starting the thread manager.

4.1.2.2 Interface (IOControl)

This provides operations for getting command parameters from the user and displaying information about the progress and results of nVerify. It is currently a Command Line Interface. This allows shell scripting in the primary Operating Systems it

will be running on. Later plans may include an extensive GUI, but this module presents the ability to display and receive the same information, regardless of implementation.

4.1.2.3 Configuration File Parser (ConfigParser)

This module reads in a configuration file and creates a test structure which encapsulates all of the information. The pattern structure inside each job structure returned from the ConfigParser contains only a message in which the name/filename, of the pattern to be used, is stored. This will tell the DPHandler which pattern to use to correctly fill in the information. The ConfigParser also contains helper functions for conversions, such as converting a string to an integer and converting a string into its hexadecimal value.

4.1.2.4 Data Pattern Handler (DPHandler)

The Data Pattern Handler is responsible for the data patterns used in data verification. It has the ability to generate patterns, such as a ramping pattern which contains 256 bytes where each byte is a number from 0x00 to 0xFF and is incremented by 1 each location. It also has some hard coded patterns in it, such as a pattern where the bytes are all 0xFF, or 0xAA, chosen for their bit representations. It also supports inputting custom data patterns external to the utility, stored in pattern files with an extension of .ptn.

An example could look like this:

```
size:16
00 01 02 04 08 10 20 40 80 40 20 10 08 04 02 01
```

The primary information is the size of the data chunk to be repeated, and the actual data. The DPHandler is responsible for creating a buffer in memory on request. It stores one iteration the generated pattern inside the pattern's data member.

4.1.2.5 Thread Manager

This module spawns new worker threads, and sends them off to perform the I/O. Each thread is given job-related information such as the I/O queue depth and which request(s) to use. It is also given information about which pattern to use and what size IOs to do.

4.1.2.6 Operating System Thread Support (ThreadWrap)

The ThreadWrapper module is an OS specific thread implementation code. It handles the OS specific code for spawning threads and makes calls to the actual worker thread code which does the brunt of the work of nVerify. Thread implementation is OS specific and therefore this module will need to be replaced and recompiled with a different source code file for each operating system.

4.1.2.7 Worker Thread (BruteThread)

The worker thread is responsible for executing specific I/O operations on the disk. It contains most of the logic and code for the threaded operation inside our program. It issues calls to through the Device I/O layer to the disk. All information is sent to the Integrity Tester module after a write/read combination has completed from the queue. The worker thread is responsible for maintaining the queue depth and choosing the amount of data to be written and the address to be written to based on the job and request

structures passed in. It handles the logic of splitting the read operation into smaller chunked reads based on parameters from the configuration file. See our Features section for more information on chunk reads.

4.1.2.8 Device I/O

This wraps the OS-specific I/O API calls within generic method calls. This allows a new Device I/O module to be written for a different operating system, and recompiled without modification to any other components of nVerify. This module also includes helper functionality to query information about the disk drive that nVerify is operating on.

4.1.2.9 Integrity Tester

This is the first module which performs data verification. Integrity tester operates as its own thread performing data verification. Integrity Tester objects exist on a one-to-one basis with the worker threads – for each worker thread performing I/O, there exists one Integrity Tester verifying the information. It maintains a queue where it stores results from the I/O operations to be verified. It compares the data read in to the data written, and on the first indication of error it goes into Triage mode where further error analysis takes place. In the case of failure it flags the global status as failure, and finishes.

4.1.2.10 Triage

In the case of an error, the triage module performs error analysis. It begins to re-read the data from the disk where an error occurred, and calculates and logs the error information, such as the offset into the file, or logical block address of the disk, and the

difference between the expected pattern and given pattern. It also logs extra information such as elapsed time, and the number of I/Os which have occurred currently. It creates a hex dump of the expected and actual data, which goes into the log file, and in debug mode, is printed to an external debugger. All of the information that is logged is done so through the Scribe module.

4.1.2.11 Scribe

This module converts the information that it receives into the correct format and logs it into the file. Currently, this means writing the information passed in into an XML file. The XML log file can then be parsed by an external tool to be developed in the future which can present the data to a user of nVerify in a clean way with no need for prior experience reading XML.

4.1.2.12 Dictionary

The dictionary defines all the structures used within the program including all the data structures for patterns, tests, jobs, and requests as well as some enumerated type definitions. Refer to the section on Data Structures in our design for more information.

4.1.2.13 Error Generator

This module is responsible for injecting errors into our program. This error generator will be a separate layer residing between the worker thread and the device IO layer. It performs many different types of error generation including changing the address to write data or read data from, changing the data written and read, and modifying the size of the data.

4.1.2.14 Disk Address Allocation (nAllogator)

This module is responsible for managing the allocation of blocks of a hard drive, and abstract offsets into a file to the separate worker threads. It maintains a list of the known used blocks and provides the address generation for the worker threads to find which blocks within the given range are free to be used by that thread.

4.1.2.15 Catcher

This module is responsible for wrapping some OS specific calls in an abstract way to be used by the application at a much higher level than the disk IO. This module defines the actions to be taken in the case of specific keyboard events. The code that handles this event is windows specific so this module will need to be replaced when compiling for a different operating system.

4.2 Flow chart

To understand the flow of execution in our program, it is best to refer to a diagram, such as Figure 4.1. The program starts out in the *DataVerification* module, as is specified by the color coded blue box in the top middle of the diagram.

Then nVerify goes into the *configParser* module on the right side of the diagram and reads in the specified configuration file. Then the main uses the *DPHandler* module to create the pattern identified in the configuration file in each job. The *DPHandler* will either generate the pattern specified or read it in from a pattern file and store it in a pattern file, returning the pattern structure to *DataVerification.* After the *DataVerification* module is done, it sends the created Test structure it received from the *configParser* with the Patterns generated by the *DPHandler* to the *ThreadManager*.

The *ThreadManager* reads in the Test structure and separates each Job into a separate thread. Then the *ThreadManager* waits for all the threads to complete and when they all return, it signals the *DataVerification* module. *ThreadManager* uses the *ThreadWrap* module to create threads and *ThreadWrap* makes a call to the *BruteThread* module for each thread.

The *BruteThread* module takes care of whether or not to use *ErrorGenerator* modules or *DeviceIO* modules depending on whether or not errors are supposed to be generated. The *ErrorGenerator* modules each use *DeviceIO* modules to manage the IO. The BruteThread uses the *nAllogator* module to identify and allocate disk memory locations for reads and writes. The *BruteThread* passes Result structures to the *IntegrityTester* when a read returns to have its data verified.

The *IntegrityTester* checks the data read back to see if it is what was expected. If it is not correct, then the *IntegrityTester* sends the Result structure to the *TriageNurse* to analyize the results. Otherwise it deletes the structure and moves onto the next result to check.

The *TriageNurse* analyzes the data and logs the error to the log file using the *Scribe* module. The *Scribe* module creates the XML log of the test results.

When all the threads are complete, from either running past the run time specified, or from stopping on the event of an error, the threads return to the *ThreadManager* and that returns the *DataVerification* module.

The *DataVerification* module then logs the global status whether it is a success or a failure and notifies the user of the end result. Then the program exits.

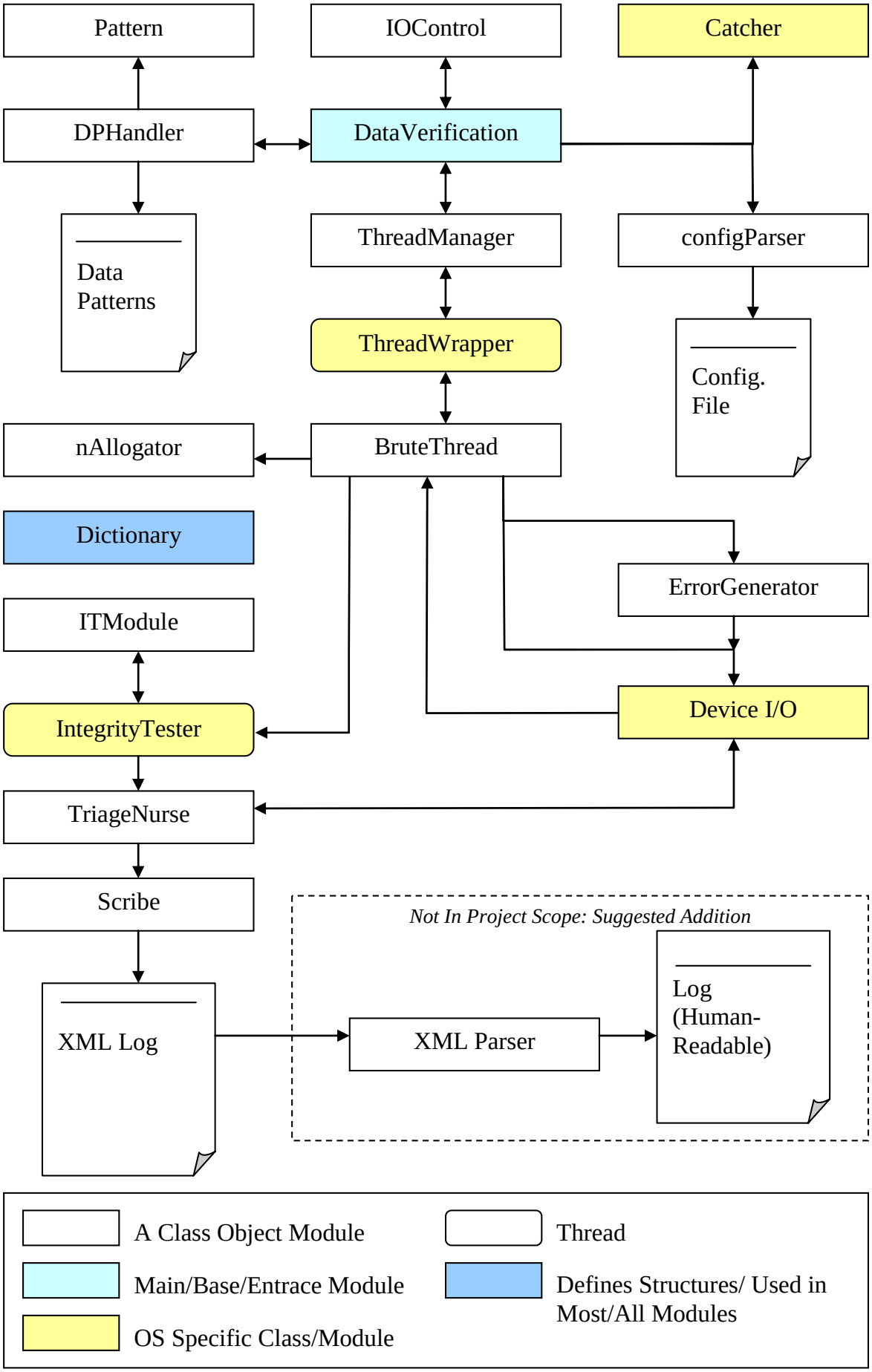

Figure 4.1: Modular Flow Chart of nVerify

4.3 Implementation

This section is going to describe the process by which we developed our project. We first used an incremental development strategy, assisted by our liaison Allen Martin. We also divided up the workload so that the whole group could simultaneously work on the project.

4.3.1 Incremental Development

The project was developed in a series of phases. The project had several phases consisting of an alpha phase, a beta phase, and a release phase. In each of the phases more functionality and features were added and the complexity of the overall program rose. The incremental development allowed a deadline to be established for each phase forcing a reasonable version to be completed at that time.

4.3.1.1 Alpha Version

The alpha phase consisted of having a program that had all of the data structures needed by the program defined and passed correctly between modules. It outlined the basic flow of the program. Not all functionality was included in the alpha version. Basic program functionality of a write and a read was performed as a proof of concept. Our alpha version was single threaded and performed synchronous un-buffered I/O.

4.3.1.2 Beta Version

The beta version had some major differences from the alpha. The beta was designed to be a complete version of the program with all functionality included but not working.

This means that all features were supposed to be present just not fully tested for this iteration. The notion of supporting different I/O request types was added to the beta version. It was multi-threaded and performed asynchronous I/O. The features that did not make it into the beta were chunk reads, triage mode, external debugger hooks, and error generation.

4.3.1.3 Release Version

The release version is a complete working version of the program. All features are included and it is fully functional within the scope of this project. In this version there are no memory leaks, and it is a stable build. There was time left after the release version's presentation to clean up the code and debug any issues that were left.

4.3.2 Managing the Workload

For our project we made extensive use of Perforce, a source and version control, to help us parallelize the workload of our project. Perforce is a tool used by developers for backing up code as well as sharing it between multiple people here at NVIDIA. The tool is run off a server and keeps a copy of all prior versions of the code. As a developer modifies a file, he or she updates the version on the server and then everyone can download it to stay up to date.

We used Perforce as a means to easily divide the parts of the project that we all worked on. While one person was working on one module, the other two were working on two separate modules and when someone finished a change, they updated it on the Perforce server. This allowed us to maintain a fast pace because we rarely encountered a situation when one person had to wait until someone else was done to progress.

4.3.3 Testing

Testing is an integral part of the development process. Unless a piece of software or utility is proven to work, it cannot be trusted to perform its purpose. With our liaison, we developed a thorough test plan with which to test our software at key points in the development process, as well as with each modification to the project. The test plan is meant to provide not only exhaustive testing of new features, but to provide regression testing to prove that our utility does not have new bugs introduced as others are tackled.

Our testing was used to stress nVerify to attempt to utilize all the control paths, and unearth any bugs that normal use might miss. We attempt to stress our utility in many ways in order to seek out unexpected behavior, such as a memory leak. Memory leaks happen when programmers do not free memory that they have allocated for use. Any users of this program need to have an output for any likely or even somewhat unlikely combinations of inputs that nVerify receives so that they do not need to be exposed to the inner workings of technical nuisances of the program. This makes testing a very high priority.

There were very strict requirements on the behavior of nVerify which we had to guarantee. We have to ensure that every test runs to completion. If it is configured to exit on an error which it detects, it has to do so cleanly, and report its findings first. We have to ensure that we catch errors which could occur which are out of our control. Some of these include checking that certain OS operations return successfully, and handling the error codes that they return to us. For example, an asynchronous I/O operation always returns a specific error reporting that it is in progress; however it does not affect the operation of our utility, so we may discard it. If an error occurs which is linked to data

verification, we must set the global status of the program execution to failure, and log all relevant data.

Another aspect was the ability to run nVerify for long periods of time. We wanted to ensure that nVerify did not have any memory leaks which would lead to degradation of performance over time. We must therefore guarantee that we do not leak or keep consuming new memory. We also had to handle bad inputs such as data patterns and configuration files. Our program output in the form of a log file must also be well formed, and match the XML standards.

Our utility, nVerify, has to be able to detect errors and inconsistencies in the operation of hard drive storage drivers. To do so, it must stress the drivers to a great degree. To make sure that nVerify catches errors of many different types, we have to test our program by programmatically causing errors to occur. In essence, we must test our testing tool. There are two good ways to test a testing tool. The first is to make sure that when no errors exist, that it does not find any, and the second is to introduce errors and make sure the tool finds them. We ran many tests to stress the program flow of nVerify under many configurations and settings. See Appendix B for a listing of the test configurations which nVerify had to be able to run to meet most of our expectations. All of these configurations acted as the inputs to our program, which we then checked the output of. Usually, if the program exited normally, and generated a log-file, we could be confident that the test completed. We ran batch scripts with the various configuration files to ensure in all iterations that the program would run to completion with no crashes, and would output successfully.

Since nVerify is multi-threaded, an important part of the testing was to ensure that it behaved the same way whether we configured it for multiple threads, or a single-thread. We also utilized third party software which aided in detecting our memory leaks during testing and development. In addition to this, a built-in utility for Windows, called *perfmon* was used to monitor many statistics of our program. It allowed us to visually see the behavior of our program, such as the disk I/O queue being filled. We used this to also show our memory usage over time, by comparing the starting memory with the memory in use right before close, and at the termination of the program.

The second main testing included error injection. Since using known buggy drivers was not in the scope of our project, we had to assume that we would not find bugs in the drivers on our own. Therefore, to stress the parts of our program that handled finding errors, we had to inject them on our own. For this, we developed a transparent error generation layer. We designed it using a popular object-oriented design pattern known as delegation. If you enable error injection, the utility creates an Error Injection module which processes all the data which a Device I/O module might otherwise process. It injects errors by corrupting the data which it receives to process, and then delegates the processing back to an internal Device I/O module. It acts as the sole interface to this module, and can therefore peek and fiddle with the data which goes through it. Error injection was designed to inject a multitude of errors, which we then tested, and tried to detect failure. If it detected a failure, then the test passed, if it did not, then it failed, or if it crashed the program, it failed. The types of errors which we injected were data corruption, address corruption, and size corruption. The first, and most complex was designed to force a lack of data integrity by changing either the data written to disk, or

the data read back from disk. The data is corrupted in many ways. Some examples include swapping one or many bytes, words, or double-words, flipping one or many bits, shifting the whole buffer left or right by some number of bytes, or writing random data into the buffer. In the case of a write you can increase the size to be written, and it can write extra data to the disk, or you can decrease, and only write some of the buffer to disk. Also, you can change the logical block address, LBA to write somewhere else. These are all errors which the driver could conceivably be guilty of itself, and therefore we wanted to ensure that if any of these things happen, that we detect a lack of data integrity.

For each of the major iterations, we ran many of these tests with and without error injection. We were able to find some bugs and holes in our code for situations that we did not account for. For the more minor code changes we tried to run tests which were geared towards what we changed. For example, if we fixed bugs in the Triage error analysis, we would primarily test with the error injection turned on. This greatly helped improve our coverage of various situational inputs and outcomes.

5 Results

This section discusses the results of our project. The final release of our program has all of the required features and more. This section also shows some of the sample input and output from our program and also discusses the errors generated and found in our program.

5.1 Tests

To test our final version, we designed many different configuration files, each stressing a certain aspect of our design. We also used the error generating layer in our code to simulate driver errors that might occur. Aside from the test configurations we created, Allen also created several more for us. The simplest test to run is a test with a single thread performing disk I/O and using a single request structure while the most complicated test is a multithreaded test using multiple requests with some of the tests being clones.

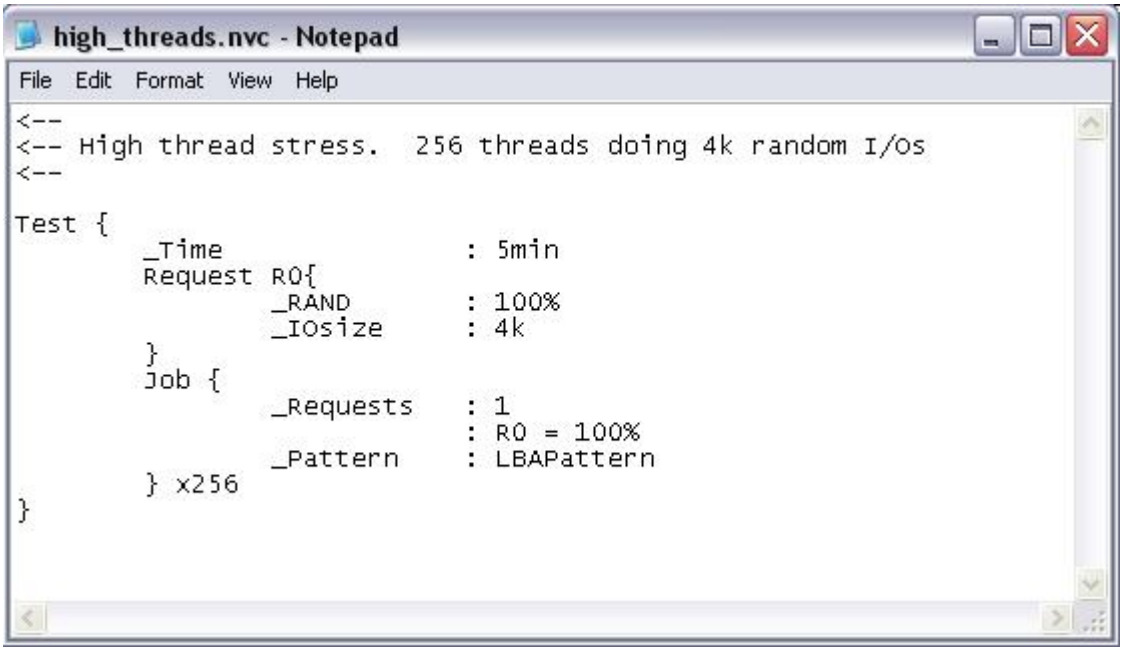

Figure 5.1: High Thread Count Stress Test

The most stressful tests are tests that have extremely large queue depths and ones that use an extremely large number of either threads or read splits where the number of individual I/O calls are very large. **Error! Reference source not found.** shows an example of a test running 256 clones of a single thread, all using a single request. This test stresses the overlapping and allocation of memory addresses because all the threads share the same range and therefore attempt to obtain and use the same memory addresses.

5.2 Sample Output

This section describes what the user to our program might expect to see. It covers what is displayed on the screen as well as what is printed to both the log file and the debugger.

5.2.1 Screen Output

During the run of our program there is very little outputted to the console for the user. Only the most basic and important information is displayed to the user. This information includes the requesting of certain required values that were not specified on the command line and outputting the results of the complete test. If all parameters that are needed are passed into the program via the command line, then there are no prompts to the user and the program simply tells the user that the test has been initiated. Figure 5.2 shows an example of the output that might occur when a user runs the program using some regular command line options.

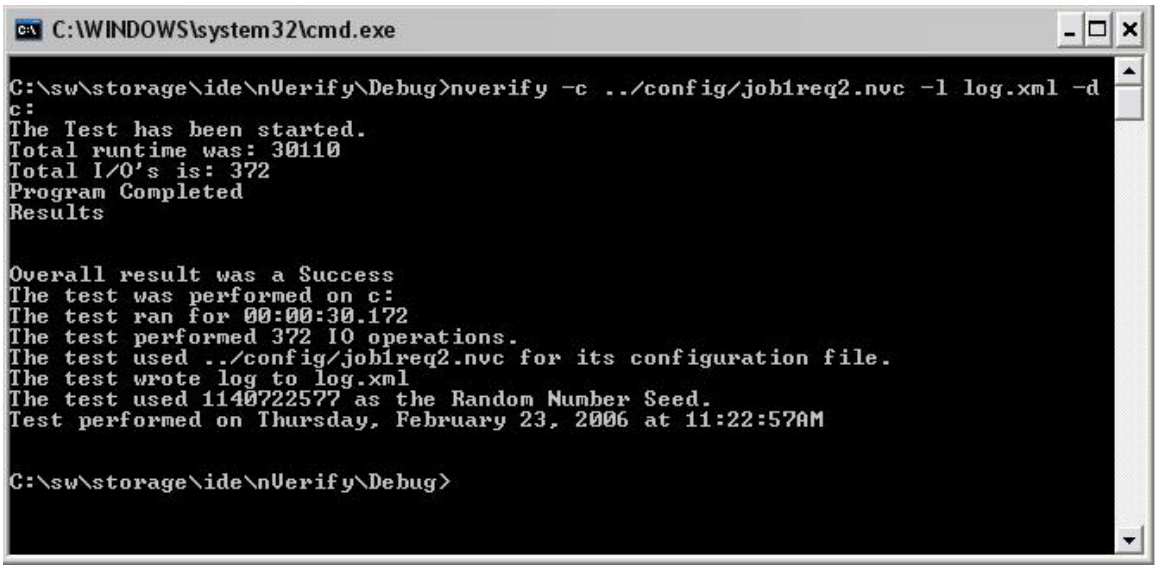

Figure 5.2: Sample Output

5.2.2 Log Output

The log file is generated and filled during the execution of the program. If there are no errors detected, the log file looks much like the output on the screen does; it simply outputs the overall results of the test. In the event of an error, the log file becomes a valuable resource. When nVerify detects an error, it will output all possible relevant

information about the error from the Triage module. The information that is outputted includes the information about which thread was responsible for the I/O, the request it was using at the time, and a hex dump of the bytes in error. This hex dump is valuable because it allows developers to see the entire range of bytes that were in error, the 16 bytes before and after the error, as well as the bytes that they should have been. This information helps enormously when attempting to debug an error. Figure 5.3 shows an example of what a log file with an error might look like.

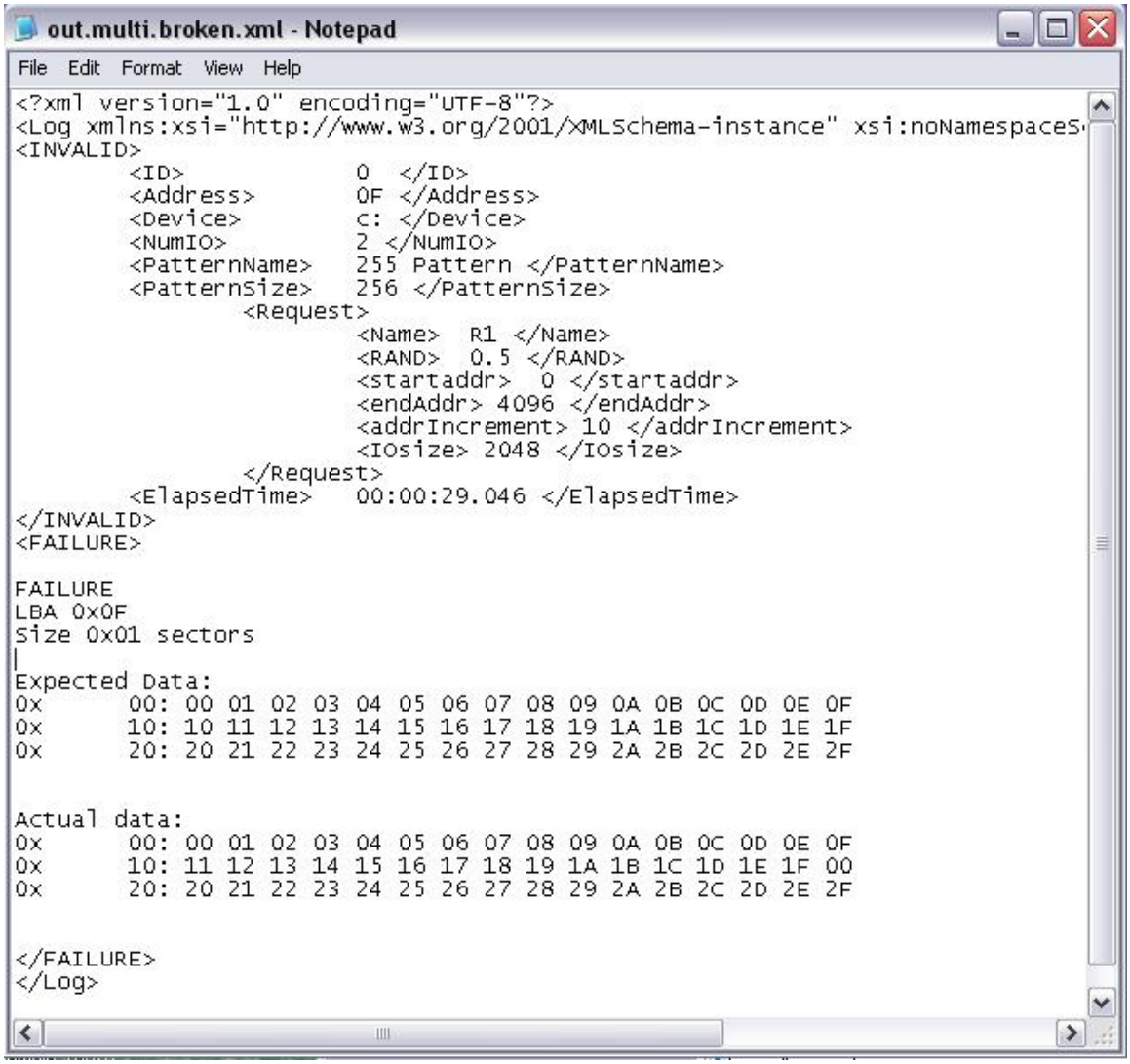

Figure 5.3: Sample Log File

5.2.3 Debugger Output

One of the command line options is '-debug.' This option enables the program to output everything it writes to the log file, to the debugger as well, and break into the debugger when an error is found. This enables developers to use remote debuggers to monitor the application. The kernel debugger will allow the developers to see what the state of the program was when the error occurred and therefore easily find the error. The output to the debugger looks much like the above output (Figure 5.3), with only the XML tags removed.

5.3 Performance

This section talks about how well our program performs, from its memory consumption to how well it keeps the disk's outstanding IO queue full. This section also discusses the speed and number of I/O's that our program can perform in a given time period.

5.3.1 Memory Usage

After the Beta release of our program, we realized, through the use of Perfmon, that our program was allocating large amounts of memory and not deallocating it, so as the program ran, it kept using up more and more memory. Figure 4 shows what the output of this monitoring program looked like. The dark blue line is the memory consumption and the red line is the current physical disk I/O queue depth. The blue line climbed fast when running our older program on a RAW disk with error generating on.

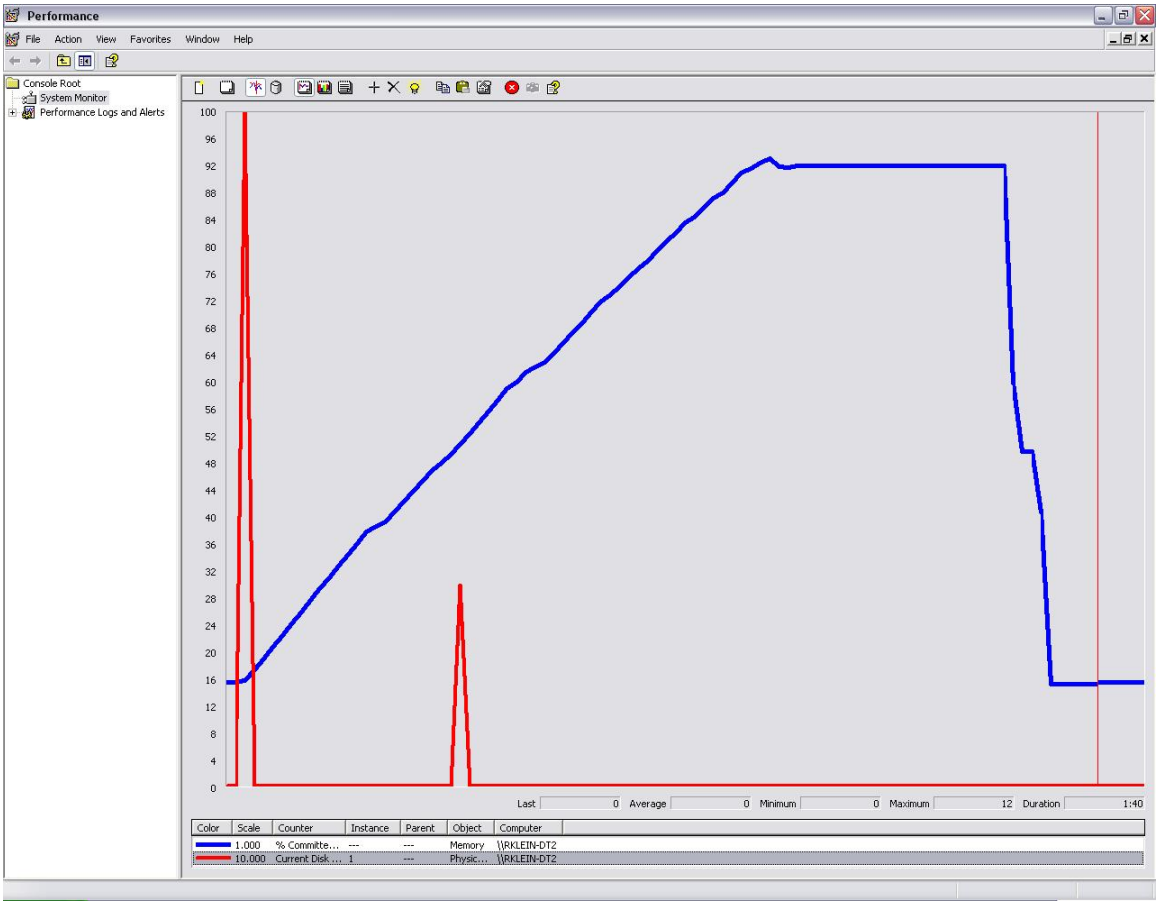

Figure 5.4: Memory Usage and Disk Queue During Execution

When we saw how bad our memory management was, we realized it was time to start freeing up unused memory. During our work, we managed to find all the places where we could minimize the amount of memory currently allocated and when possible free it up so that other processes could use it. After a few days of aggressive memory management and several dozen runs later, we finally got our memory management under control. The graph of our current memory usage is shown in Figure 5.5.

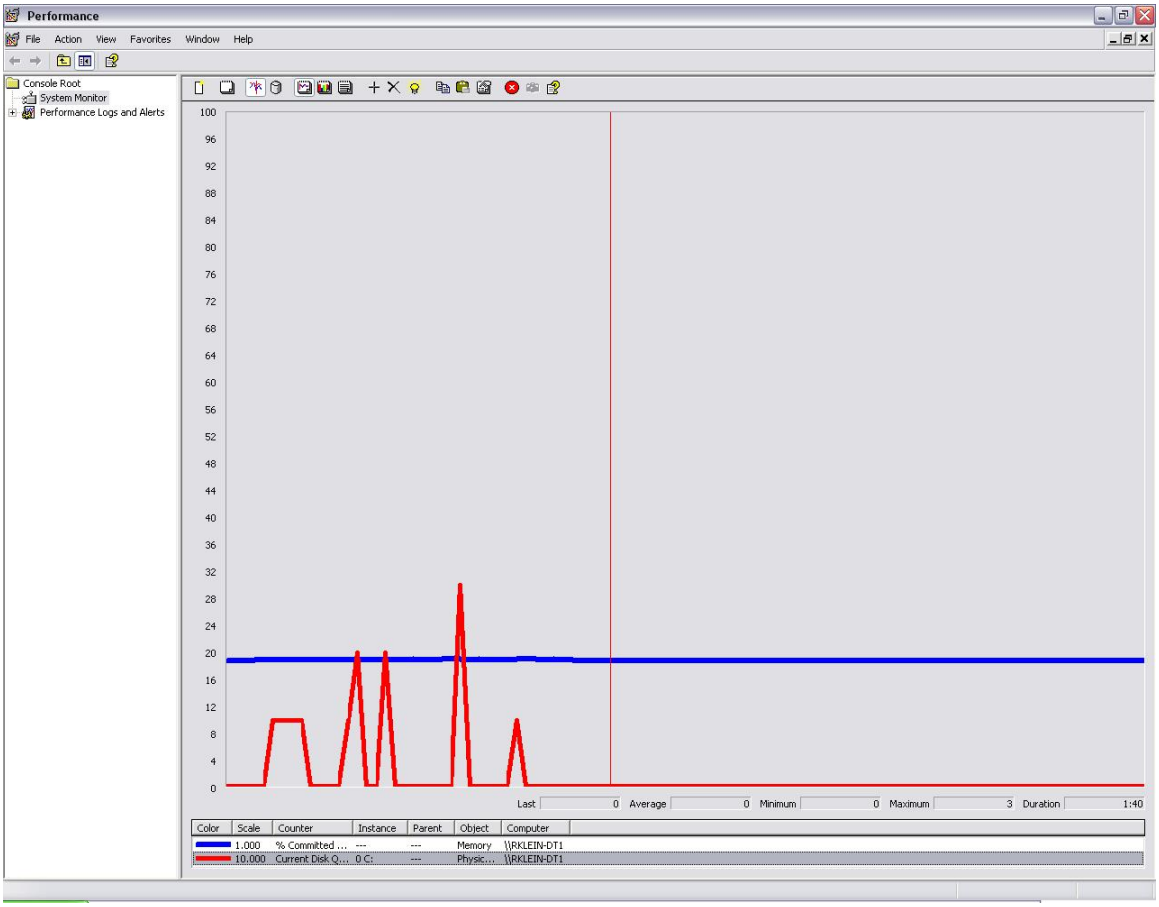

Figure 5.5: Release Memory Usage

Through use of the libraries that Allen pointed us to, we are able to monitor the number of memory leaks and after thorough testing, we finally got this number down to zero. This is why you can see the blue line level off as the program runs.

Also you can see in both graphs that the red line, the current disk I/O queue, is being emptied and refilled during the course of operation of our program. The spikes represent calls to perform I/O operations by our program while the dips represent the driver completing these requests and reporting back that they are complete.

5.3.2 Speed / Number of I/O's Performed

One of the other problems we encountered is that while we were debugging, we were doing several things that greatly slowed down the processor and limited the number of I/O's each thread could perform.

The first limiting factor was the number of print out statements we had. Print out statements slow the processor greatly, changing the number of possible I/O's performed from somewhere like 600 to approximately 340.

The next limiting factor which we encountered was the way we were checking and handling errors. First, in order to find the problem, we were searching through the array of data byte by byte and comparing it to the expected value. We did this same operation again inside the Triage module when trying to find out what parts of the data were different. We instead now use *memcmp()* to compare the data in 16 byte chunks, saving the number of comparisons we have to perform and thus speeding up the process.

The last limitation was the file system itself. We found that when performing writes on a RAW disk, our program could perform up to 100 times the number of I/O's per minute than the same test run on a disk with a file system.

5.4 Bug Detection

This section describes our use of generated bugs to test our software and also unexpected bugs that arose during development that we were forced to handle.

5.4.1 Generated Bugs

Our program was required to generate many different types of errors on both read and write commands. While developing, these introduced errors were crucial in the

design and implementation of our system. Using the string generated we were able to test our integrity testing and triage handling modules. Having run them against these known errors of all types, it increases the likelihood that nVerify will catch similar errors that might actually occur.

5.4.2 Unexpected Bugs

During the course of developing our program, we ran into several difficult problems. The first was when we were freeing up memory before it was done being used, causing somewhat random memory errors where pointers would be deleted while they were still be accessed. This error took a while to track down because we had to backtrack through our code to find the deallocate statement, which was often in an entirely different module, that was causing the bad pointer.

The other problem we were finding is that while performing thousands of I/O operations on a RAW disk, every now and then a comparison would fail. The results of this comparison showed that where one data pattern was expected to be written, the read call returned the data from another data pattern. After meticulously going over the code, we came to an interesting conclusion, with the help of our liaison. We found, using busTRACE, that the read operations were being completed before the corresponding writes were, a clear violation of what the driver was supposed to be guaranteeing. busTRACE is a program that will monitor all the activity of a drive or drives and reports what is done to it. Below is a screenshot of busTRACE showing the order of the write and read and the order that they finished in.

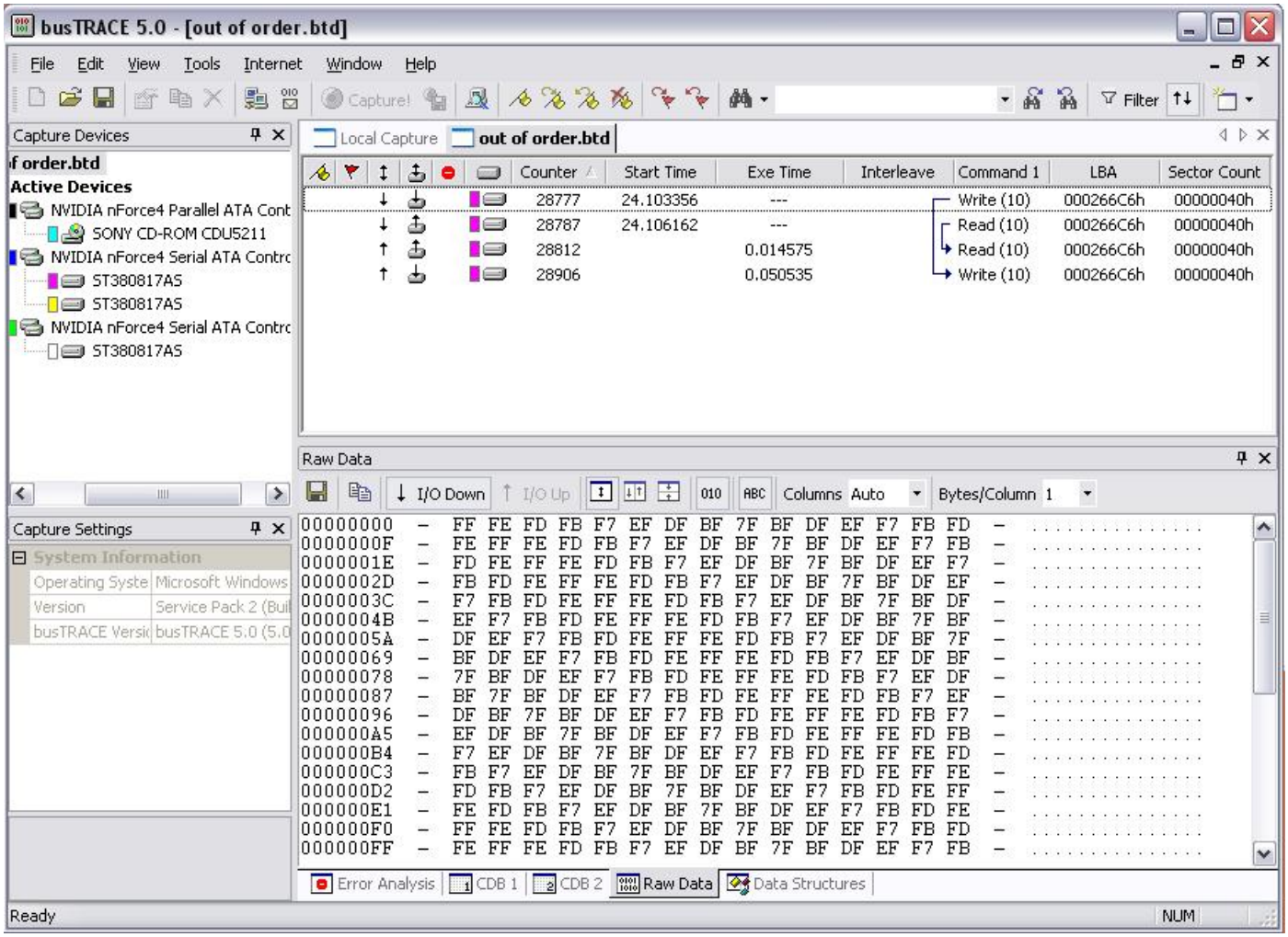

Figure 5.6: busTRACE Showing Driver Bug

In the top right pane of the screen shot above, there are 4 lines that have arrows and icons of hard drives in them. The first two of these lines are a write and a read command being issued. If you follow the arrows in the “Interleave” column, you can see that the read’s arrow ends before the write’s arrow ends. By looking at the “Start Time” and “Exe Time” columns, you can verify that the read was started after the write and the read executed before the write. Therefore, the data read back by the read is stale data from the previous write command which could be from an entirely different data pattern. We only found this problem when the program was operating on a RAW disk and performing

many thousands of operations. This shows that while our tool was only recently developed, it has already caught a bug in their drivers that their previous tools have not. This bug is common in different versions of the driver meaning that it is common code to all of them.

6 Conclusions and Future work

The following discusses the goals that our project was set to accomplish and which ones we accomplished. It discusses who will use our software. It also discusses what we accomplished for the sponsor as well as what improvements could be made in the future with our software, that section also includes improvements that were outside of the scope of the project.

6.1 Goals Set and Accomplished

The data verification project had many important goals which it needed to accomplish. The most important goal of the project was to stress NVIDIA's hard disk device drivers. We successfully created a tool which stresses the drivers. Our tool creates many threads and many I/O's which place a great deal of stress on the drivers. The stress we placed on the drivers was demonstrated through our finding of an unknown bug in their current drivers.

Another important goal of our project was to make sure that our program would be easily portable to other operating systems. This was important because NVIDIA produces drivers for multiple operating systems and having a tool that can test their all their drivers is important. Our project was designed to be easily portable through the use of modules abstracting and wrapping the OS specific calls. This way our project is able to be ported by replacing the few classes that are OS specific.

The ability of our program to be highly configurable was another goal that we achieved. This is done through the command line parameters and the highly customizable configuration files. The command line parameters allow a user to specify which drive

they want to test, which configuration file to use, the name of the file the log should be stored in, and whether they want to stop or continue on errors. Having these options for the command line allows a user to create batch files that are specific to their needs. Having all these customizable features allows the user to create test cases which will suite their needs and hopefully diagnose problems with the drivers.

An important goal for our sponsors was to find bugs in their driver software because having bug-free drivers is important for a large well-known company like NVIDIA. Our program met this goal by finding a previously unknown bug in the current drivers. By using our software in the future we believe that it will assist NVIDIA to diagnose problems earlier in the development cycle to ensure that their drivers are stable.

An exciting accomplishment is that the Quality Assurance (QA) team at NVIDIA will be using our tool in their test plans for checking for bugs in their drivers in the future. With this new tool it may be possible to catch errors and bugs that the other testing tools can not. Once an error is detected by QA, the engineers would then use our tool to try to narrow down what the bug could be.

6.2 Possible Improvements/ Follow-up Work

While many of the goals of nVerify were met, there were other goals outside the scope of this project. NVIDIA's storage software group and quality assurance group has plans for future development of nVerify. They plan to not only extend nVerify in usage in the company, but also to extend the functionality and features of the tool beyond what we were able to do in 9 weeks.

The major add-on to the project at NVIDIA is a version of the tool which will run on Linux. One of our goals was to make our program modular in part for this future goal.

In theory, only changes to specifically the Device I/O modules as well as the threaded modules will need to be made. Practically speaking, there are always other hurdles. The specific behavior of supported function calls can differ between Operating Systems, but we accounted for these as best as possible.

One area of testing the functionality of our tool that was deemed future work was running our tool against drivers with known issues, and seeing if it could expose them. This is planned to be done by NVIDIA at some time. This kind of testing would be crucial to increasing trust and reliability in nVerify.

Since this tool is going to be rolled out by QA into their testing cycle, there are some improvements which would aid them in their work. Automation is a big part of the future of nVerify. The command line interface of nVerify makes it easily scriptable in any OS which supports such scripting. This way one script could run a multitude of test configurations on various drives and for a large number of times. Another component of the automation of nVerify is drive detection. Currently, a user specifies which drive to have nVerify operate on. They plan to have a small program that can detect which hard drives to run nVerify on, and delegate the commands to nVerify by creating a new instance of it with the specified hard drive parameter.

The user interface has a lot of room for improvement. While the command line interface provides for easy scripting, some less experienced users will find it cumbersome to use. A GUI has been planned further down the road, which would also eliminate the need for a configuration file, as parameters could be easily specified at runtime. We tried to design the Command Line interface with an API which would make the core program require no modification if they wanted to replace it with a Graphical UI.

One feature requested by the software group at NVIDIA was the ability to call arbitrary commands to a hard drive via the controller chip, and therefore via their driver. This type of testing was not in the scope of our project. The major modifications which would be needed would be additions to the Device I/O module API, as well as additions to the core functionality of the program – deciding where you would call various commands, and how you would verify the results.

More major modification suggested was performance statistics gathering. We currently only gather information on the elapsed time a given I/O might take. NVIDIA thought it might be useful to have more statistics gathered such as time spent at various points on the stack, and so on. This may be added at a later time. More configuration modifications were also suggested. In the future, the group might want to be able to specify CPU affinity – meaning being able to specify which processor a given thread should run on. This would be useful on multi-processor systems to balance the load manually to attempt to reproduce an error. Also, they suggested changing the configuration to specify the range of addresses in which to perform I/O to be a part of the job, and therefore specific to thread, as opposed to specific per I/O request. The last addition in the foreseeable future of nVerify is an external tool. Since we output all of our information to an XML log, an XML log parser utility may be of use to be able to present the relevant data to a user who does not wish to sift through XML tags.

References

1.) Abraham Silberschatz, Peter Baer Galvin & Greg Gagne, Operating Systems Concepts, 6th Edition. John Wiley & Sons, Inc: New York, NY. 2003.

2.) Welsh, Matt. *Hard Drive Geometry.* http://grouchy.cs.indiana.edu/usr/local/www/linux/gs/subsection2.6.5.1.html, Last Accessed: February 7th, 2006.

3.) Microsoft Knowledge Base. *INFO: Direct Drive Access Under Win32.* http://support.microsoft.com/kb/q100027/, Last Modified: May 6th, 2003.

4.) MSDN Microsoft Corporation 2005.

5.) W3C : Extensible Markup Language (XML) 1.0 (Third Edition), Recommendation 04 February 2004, http://www.w3.org/TR/REC-xml/ , Last Accessed: February 9, 2006.

Appendix A: Feature Document

This document describes in detail the different features that NVIDIA wanted from our project. The first section is a complete list of all the functionality that NVIDIA said they would want from a final finished version. The sections after that describe what features they thought we could accomplish in our time period.

Introduction

The Data Verification Tool will provide developers with an easy to use tool that can stress the NVIDIA nForce hard disk controller drivers. The tool will execute reads and writes to a single disk or disks in a RAID configuration and will perform verification to make sure that the data written is the correct test data and that it is in the expected location on disk. The tool will be modular so that its functionality can be extended in the future. This tool will ultimately provide many customizable features to allow developers and testers alike to create unique tests to fit their needs and desires.

Exhaustive Feature List

- Perform I/O reads and writes on a hard disk via a file system, or as a raw device
- Will not differentiate between single disk or RAID configuration
- Will be able to control the following:
    - Size of data or range of sizes of data
    - Number of worker threads performing I/O
    - Randomness of I/O location
    - Writing to streams or across surfaces of disk
    - Range of addresses
    - Devices it operates on
    - Arbitrary commands
    - CPU affinity
    - I/O queue depth per thread
    - Splitting Reads into multiple smaller reads
    - Stacked I/O's (multiple writes/reads to same location in 1 thread in a row)

- Triage mode
- Collect performance statistics (e.g. time spent in various times at stack.)
- Control and configure data patterns
    - Use pre-configured data patterns (e.g. subsequent blocks being different to detect “off-by-one”)
    - Allow user to provide custom data patterns
- Error injection
- Logging
    - Start time
    - End time
    - # of I/Os
    - Breakdown of reads/writes
    - Provide one end status (Pass/Fail) for QA
- Logging on failure – detailed analysis
    - Disk address of current write
    - I/O pattern size
    - Memory address
    - Previous I/O pattern
    - Offset into I/O
    - Expected result
    - Difference between the expected, and obtained result
- Hooks
    - Programmatic hook to stop the test
        - Break into the debugger on stop
- Modular design to allow for further extension
- API documentation
- Plug-in to allow integration with other tools
- Log – XML format
- User interface
    - GUI
    - Command-line
        - Allow shell scripting
- Performance mode
    - Allows IO’s without data integrity testing
- Inline integrity testing
    - Allow for integrity testing in separate thread or inline.

Scope of WPI Project

As our project was only going to span approximately 9 weeks, NVIDIA felt that the scope of our project could not be the entire list of features that they would want because it would take too much time to develop them. They instead decided to give us a large

subset of features that would create the design and layout of the program leaving it open to add the last remaining features. Our features we broken into 3 groups based on the phases of our project.

Alpha

- Perform I/O reads and writes on a hard disk via a file system, or as a raw device
- Will not differentiate between single disk or RAID configuration
- Size of I/O or range of sizes (for alpha two sizes: small, large)
- Arbitrary commands
- Control device operated on
- Control and configure data patterns
- Repeatable patterns (plus some pre-canned) (Alpha only one or two pre-canned)
- Logging
    - Start time
    - End time
    - # of I/Os
    - Breakdown of reads/writes
    - One status for QA at end of run
- Modular

Beta

- Multiple threads with ability to specify the # of threads
- Randomness of I/O (location of where) streams or across surfaces of disk
- Range of addresses
- Splitting Reads into multiple smaller reads
- Stacked I/O's (multiple writes/reads to same location in 1 thread in a row)
- Triage mode
- Error injection (testing)
- Log may be in XML
- Logging on failure
    - Disk address
    - I/O size
    - Memory address
    - Previous I/O
    - Miscompare
    - Offset into I/O
    - Expected result
- Hooks
    - Hook to stop the test (break in debugger)
- API document

Appendix B: API

This is the API (Application Programming Interface) for our software. It describes how input is read into the program as well as how it is outputted from the program. Output comes in two ways from the program, messages intent to be sent to the log file and messages intent to be sent to the user.

IOControl (User ←→ Application)

Output:

- nPuts(Char *)
- nPuts(Pattern *)
- nPuts(Result *)
- nPuts(Request *)
- nPuts(Job *)
- nPuts(Test *)
- nPuts(LogEntry *)

These functions are to map the particular structure specified from the application to the user with the specified parameters. They take as parameters char *, pattern *, result *, job *, test *, logentry *.

Input:

- Char * nGets(FILE)

This is the function that is used to get the information from the user. The file descriptor could be used for stdin/stdout, socket, or file descriptor.

Device I/O

Write:

- bool nWrite(char* dataBuffer, DWORD bufSize, BIG_INT lba);

This function will take in a data buffer of bytes to write. It also takes as its parameters the size of said buffer, and an lba to define the address. It performs the I/O unbuffered, and asynchronously. Returns true immediately if there were no errors.

Read:

- bool nRead(char* dataBuffer, DWORD bufSize, DWORD *readBytes, BIG_INT lba);

This function will take in a data buffer to fill with read bytes. It also takes as its parameters the size of said buffer, an lba to define the address, and a pointer to provide the number of bytes that were actually read, as well. It performs the I/O unbuffered, and asynchronously. Returns true immediately if there were no errors.

Get Result:

- bool GetResult();

This function will return the status of the DeviceIO object. If the I/O performed with the object has completed, then it will return true, else it will return false.

Get Bytes Transferred:

- bool GetBytesWritten(DWORD* size);
- bool GetBytesRead(DWORD* size);

This function will set the number of bytes either written or read into the passed in double-word variable. It will return true if the query succeeded, else it will return false.

Get Sector Size:

- static long GetSectorSize(string device);

This function will return the number of bytes in a logical sector for the device represented b y the string parameter.

Get Last Address:

- static BIG_INT GetLastAddress(string device, bool isRaw);

This function will return the LBA of the last address of the disk. It takes a Boolean which determines if the device we are dealing with is raw or contains a file system. It then calculates the last address either using the free space on a file system, or the total disk capacity on a raw disk.

Appendix C : nVerify User's Guide

This document is meant to give the reader knowledge of how nVerify is configured, how to run the program, and how to make sense of the data after a test is complete.

Setup

This section will go through the setup of a test. It will take you through an example configuration file and a pattern file. This program can be run from either a command prompt with arguments passed in which specify all the parameters for the test or it can be run in an interactive mode where it will prompt you for the parameters it requires.

The Configuration File

The configuration file defines a test the program will perform. The main components of a test file are the *Requests* and the *Jobs*. Comments inside a configuration file are specified using '<--'at the beginning of the line.

Request

Each Request has several fields which can be specified for it. A Request contains the following fields:

Member Name	What is Means	Type	Example	Required
name	Name	Word	R1	Yes
RAND	Percent of I/O's random	Number	80%	Yes
startAddr	Starting Address	Number	0x1000	No
endAddr	Ending Address	Number	10345	No
addrIncrement	Address Increment	Number	0x10	No
IOsize	IO size	Number	32K	Yes

The *Request's* name is supplied right after the word 'Request' separated by a space. The values for *startAddr*, *endAddr* and *addrIncrement* can all be inputted as either hexadecimal values, by preceding the value with '0x', or decimal values. The IO size in bytes is specified with a decimal value followed by an optional size indicator. There are three valid size indicators recognized by the current version of nVerify. The first is K which signifies kilobytes, M for megabytes, and G for gigabytes. All the numbers entered for a request are entered as whole numbers. Here is an example of a Request as it would look in a configuration file:

```
Request R1{
     _RAND       : 50%
     _startAddr  : 0x0
     _endAddr    : 0x1000
     _IOsize     : 2K
     _Increment  : 10
}
```

Figure 1: A Sample Request

Job

The next major element of the configuration file is a *Job*. A *Job* represents a single thread of execution in the program. A single thread (*Job*) can be configured to use up to 100 different requests each with its own parameters. A *Job* has the following parameters:

Member Name	What is Means	Type	Example	Required
Request	The Requests to use	Request	Figure 1	Yes
QueueDepth	# of outstanding writes	Number	20	No
Pattern	Pattern to write	Word	0xFFpattern	Yes
ChunkReads	# of chunks to split read into	Number	2	No

The Request field is defined in two parts. The first is the number of requests the job is going to use and the second part consists of defining how often to use each of the *Requests* for the *Job*. An example is shown below. The *QueueDepth* is the amount of outstanding concurrent writes that are going to be performed. The Pattern is the data that is going to be written to the hard drive and the *ChunkReads* is the number of times to split up each read. Both the *QueueDepth* and *ChunkReads* are entered as whole numbers.

```
Job {
   _Requests    : 3
                : R1 = 30%
                : R2 = 35%
                : R3 = 35%
   _QueueDepth  : 15
   _Pattern     : 0xAApattern
}
```

Figure 2: An Example Job

Outside of the Requests and the Jobs, there are only two other fields that can be set. The first is the most important, the *Time*. The *Time* specifies the runtime for the entire test. After the amount of time passed in, nVerify will stop issuing new reads or writes. There are several indicators that can be used to show units of time. The first is ‘sec’ which means the number passed in is in seconds; then there is ‘min’ for minutes and ‘hrs’ for hours. The last field that can be entered by is not required, is the *RANDseed* value. *RANDseed* specifies the number you can use to seed the random number generator used in nVerify. If not specified, the current time in seconds will be used. Only one indentifier should be used in a test, so a valid use is “90sec” not “1min30sec.”

Below is an example of a simple configuration file:

```
<-- Test Configuration File

Test {
    _Time              : 15sec
         _RANDseed     : 15454552
    Request R2{
         _RAND         : 20%
         _startAddr    : 0x20000
         _endAddr      : 0x30100
         _IOsize       : 32K
         _Increment    : 0x10
    }
    Request R3{
         _RAND         : 100%
         _startAddr    : 0x1000
         _endAddr      : 0x1AB80
         _IOsize       : 64K
         _Increment    : 100
    }
    Job {
         _Requests     : 2
                       : R3 = 50%
                       : R2 = 50%
         _QueueDepth   : 10
         _Pattern      : walkingOnes.ptn
         _ChunkReads   : 4
    }
}
```

Figure 3: An Example Configuration File

The Pattern File

There are two different types of patterns, the generated ones, and the ones loaded from pattern files. Generated ones include: pattern255, 0xAApattern, 0xFFpattern, Fibonaccipattern, and LBAPattern. Pattern files are defined as hex files ending in '.ptn'. The first line of the file consists of 'size:X' where X is the size in bytes of the pattern. After that the hex file is arranged in rows of 16 pairs of hexadecimal digits separated by a space. Figure 4 is an example pattern file.

```
size:16
00 01 02 04 08 10 20 40 80 40 20 10 08 04 02 01
```

Figure 4: An Example Pattern File

Execution

Running a Test

There are two different modes that nVerify can run in. The first method is by passing all the parameters for the test in the command line to the program. The program requires 3 parameters and has several other optional ones. The three required parameters are:

Parameter	**Example:**
Name of the configuration file	-c job3req3.nvc
Name of the log file	-l output.xml
Device to operate on	-d PhysicalDrive1

In order for the program to know which field is being specified, the field is preceded by an identifier. The configuration file identifier is '-c', the log file identifier is '-l', and the device identifier is '-d'.

There are several option commands that can be passed in from the command line. The first is '-help' which if passed in will force the program to output a help screen that lists all the possible command line arguments and their identifiers. The next optional command line argument is '-p' which lets the program know that when it encounters and error, it should stop processing new commands and exit after it has completed analyzing and logging the error. The last option command line argument is '-debug' which tells the program that you are going to be monitoring the program from an external debugger and

signifies that you want it to output all its information on errors to the debugger and to pause and break into the debugger after an error occurs.

The second way of running nVerify is through an interactive approach. To run it in this mode, simply start nVerify with no command line arguments passed in. It will prompt for the 3 required fields and then run the test. If nVerify is run in an interactive mode and you wish to specify to either pause or break into the debugger, then pass the optional command line arguments in through the command line anyway.

If any of the required command line arguments is not passed in nVerify will prompt the user interactively for the required field again.

Incorrect Parameters

If an incorrect parameter is passed into the command line, such as ‘c;’ instead of ‘c:’ the program will terminate outputting a message indicating the error. The only exception is if you specify a configuration file that does not exist or is not found. nVerify will prompt for file name to be reentered. In the case of a malformed configuration file, the program will output an error message indicating the type of error and then terminate.

Log File

The log file generated by nVerify is very small in the case of a success. In the case of a success the log file contains only the most basic information about the test including the runtime, the information about which test was used, what device it operated on, and how many I/O operations were performed. This information along with the overall test result, a success or failure, is logged regardless of the status of the test.

In the case of a failure, the log file also logs all information it knows about the error to the log file. It logs which thread was responsible, where the write/read was being done, the data pattern used, the request structure being used, and the mismatched data. The mismatched data is represented in the form of a hex dump. This hex dump shows the data that was expected to be there, and then it prints out the data that was read back from there.

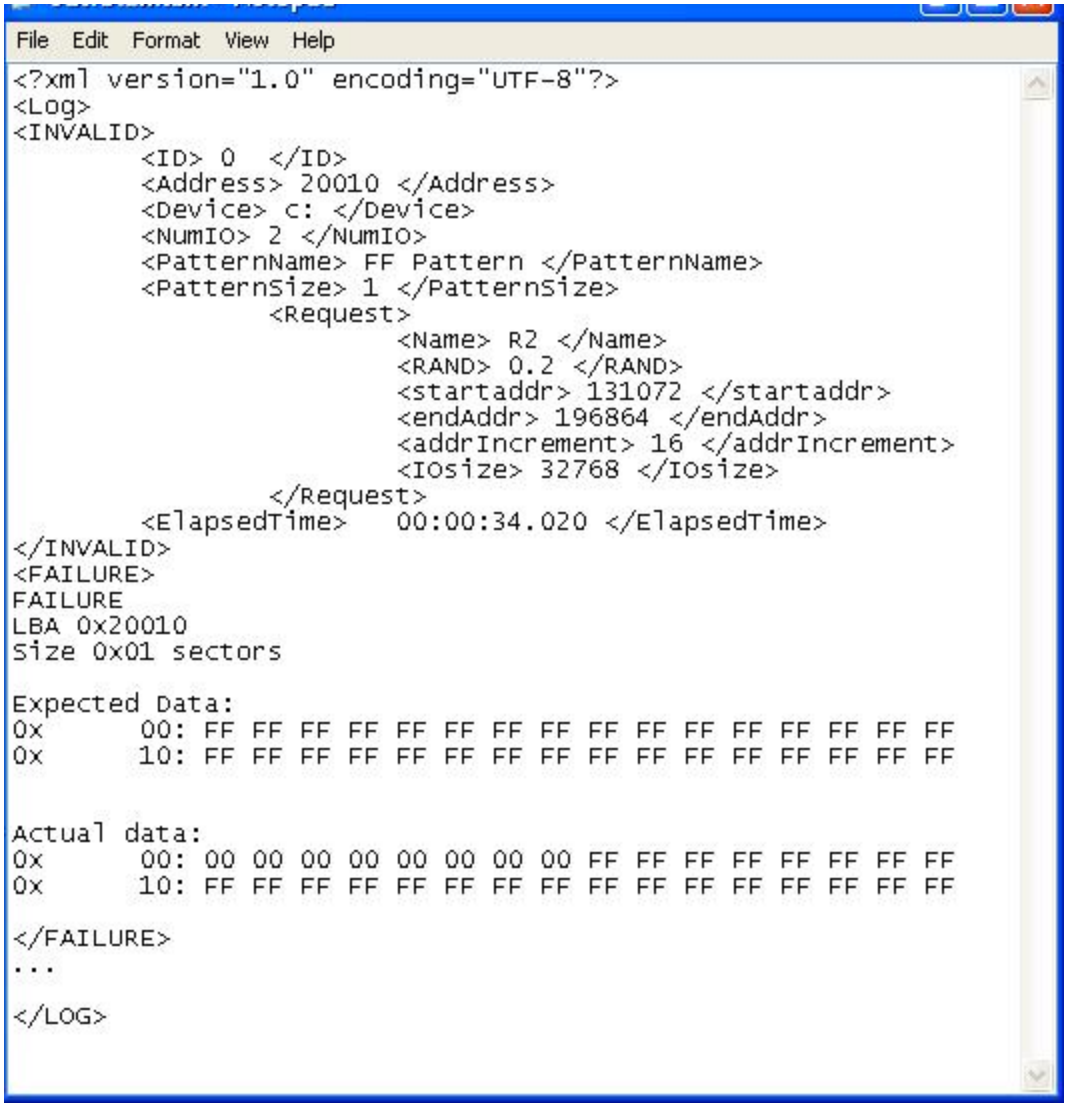

Figure 5: Log Entry on Failure

The first information logged on an error is all the relevant information about the test configuration at the time that the I/O was issued. This includes the job information, the information about the request used, and the elapsed time at the time of the error. The next section is one of the most important sections, it is the hex dump. The hex dump starts 16 bytes before and goes to 16 bytes after the broken section. NVerify prints out the hex dump line by line with the starting offset of the line from the address the read was performed on and then prints out the 16 bytes of data in hex values.

The log file on a success is much shorter. It displays only the overall test results. This information includes the elapsed time, the number of disk I/O's performed, configuration file used, and the global result of success. Figure 6 shows a log file of a successful run.

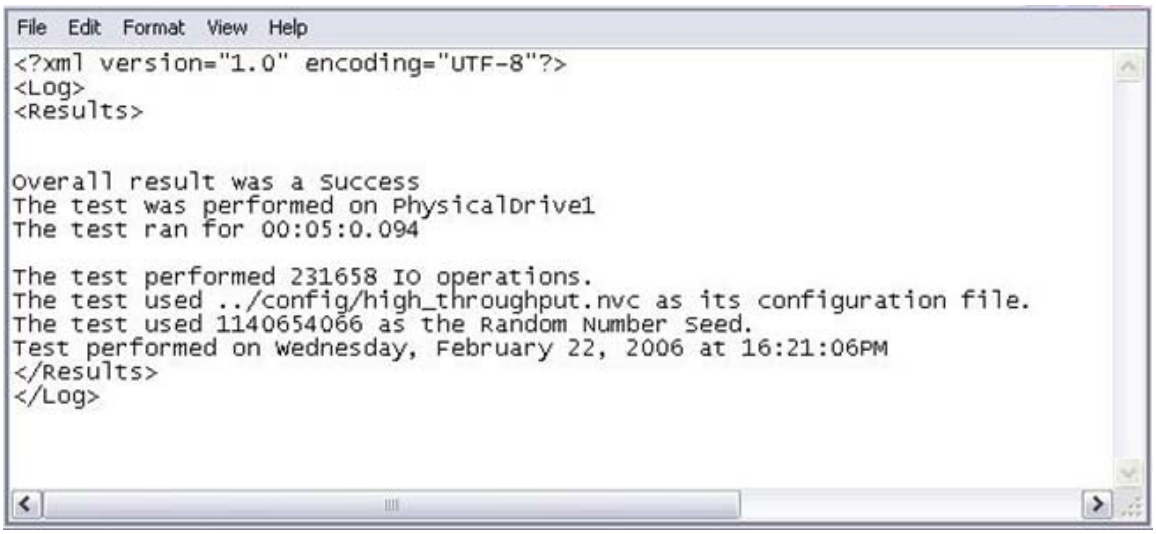

Figure 6 : Log of Succesful Run

Appendix D – nVerify Developer's Guide

This Manual is for the nVerify tool created in March 2006.

Brute Thread

`void Go(Job *theJob, long Seed, nAllogator* DiskAllocator)`
This function is the main worker of the program, it keeps the queue filled and determines randomly which request to use and issues the writes and reads.

`double * Requestparse(string RequestPercentages, int NumRequests)`
This function takes in a string that has the request percentages stored in it and parses them into an array of doubles. The input string will be of the format "30:35:35" then the output of the function would be the numbers stored.

`ReqSize *RequestSeperation(double *percentVals, int NumRequests, Request **myReqs)`
This function makes an associated array with the request percentages. For example if you the two requests split evenly like 50:50 it would get turned into an associated array like (50->R1, 100->R2)

`ReqSize * makeRequest(string RequestPercentages, int NumRequests, Request **myReqs)`
This function calls `RequestParse()` then `RequestSeperation()` which creates the associated array of requests with their percentages.

`char *getDeviceName(bool isRaw, const char* Device)`
This function creates the device name whether for filesystem (c:\\filename.txt) or raw (\\\\.\\PhysicalDrive1)

`bool randLocation(double randNum)`
This function chooses to see based on a percentage passed in whether or not the write should be to a random location.

`ULONGLONG *getChunkReads(long IOSize, ULONGLONG srtaddr, long Chunks, string device)`
This function decides upon the random addresses to where the reads should happen in a split read.

`ULONGLONG *DupCheckChunk(ULONGLONG *allChunks, long IOSize, ULONGLONG srtaddr, long Chunks)`
This function ensures that when performing a split read that it does not try to read from the same location.

`ULONGLONG *SortChunks(ULONGLONG* TheChunks, long Chunks)`
This function sorts the addresses where the splits should occur with regards to the split reads.

`long GetBegOfReadChunk(long RPI, long QD, long numChunks)`
This function returns the index of the read split that is supposed to be next, meaning if there are three split reads and they start at 1,4,7 and the index passed in is 2 then it will return the index 4 as the next read to check.

Catcher

`int closeNhandle(nHANDLE thehandle)`
This is a wrapper function that will close the windows specific file handle.

`nHANDLE nMutex(bool acquire,char *name)`
This is a wrapper function that creates a mutex and returns the handle to it.

`DWORD WaitFSO(nHANDLE hObject,DWORD timeOut)`
This is a wrapper function that waits for a single handle and for specified timeout.

`bool nRelease(nHANDLE hObject)`
This is a wrapper function used to release the mutex.

`static void PrintDebug(const char *)`
This is a wrapper function ensuring that printing is done to the debugger.

`static void BreakDebug(void)`
This is a wrapper function ensuring that the program will break into the debugger when called.

Config Parser

`Test Parse(string filename, string device)`
This function opens the configuration file and parses all the data in it and puts the data into their appropriate structures. This then returns the Test structure that holds all the information

`static long getValueNumber(string str, size_t a, size_t z)`
This function takes a string and converts the string to the corresponding number returning a long.

`static double StringToDouble(string str)`
This function takes a string and converts it to a double which it then returns.

`static int StringToHex(const char *str, int len)`
This function takes a char * and a length which are in hexadecimal and returns the integer value of it.

`static string LongToString(long num)`
This function takes a long and converts it into a string which is returned.

`static string nLargeIntToString(BIG_INT li)`
This function converts a BIG_INT's high part and low part into a string.

`static string nLargeIntToString(ULONGLONG li)`
This function converts a Ulonglong into a string.

`static ULONGLONG StringToHexLL(const char *str, int len)`
This function takes in a length and a char* in hexadecimal format and converts it into an Ulonglong.

`static string HexToString(ULONGLONG hex)`
This function takes in a Ulonglong and converts it into a hexadecimal number contained in a string.

`static string ULLToBinary(ULONGLONG bin)`
This function takes a Ulonglong and converts it to a binary string.

`static char *BinaryToHex(string bin)`
This function takes a string of binary numbers and converts it to hexadecimal contained in a char *.

`static string DecimalToBinary(string dec)`
This takes a string of decimal numbers and converts it to a string of binary numbers.

`static string IncrementDecString(string dec,long amount)`

This function takes in a string and adds the corresponding long and then returns the new string created.

`static ULONGLONG StringToULLDec(string num)`
This function takes in a string and converts it to decimal and stores it in an Ulonglong.

`void ZeroOutTest(Test *t)`
This function resets the `Test` that is passed in to default values.

`void ZeroOutJob(Job *j)`
This function resets the `Job` that is passed in to default values.

`void ZeroOutRequest(Request *r)`
This function resets the `Request` that is passed in to default values.

DeviceIO

`DeviceIO(bool isRaw, char* device)`
This constructor takes two arguments. The Boolean, isRaw, determines whether this `Device I/O` object is going to perform requests on a raw drive or file system drive. The string is the name of the drive / file to read from or write to.

`bool nWrite(char* dataBuffer, DWORD bufSize, BIG_INT lba)`
This method takes in a data buffer to write to disk, the size of the buffer, and the lba for raw I/O and offset for file system. It returns true if there were no errors, and false if there was an error. If DEBUG_PRINT is set, then it will print out the error code if there was an error.

`bool nRead(char* dataBuffer, DWORD bufSize, DWORD *readBytes, BIG_INT lba)`
This method takes in a data buffer to read data from disk into, the size of the buffer, and the lba for raw I/O and offset for file system to read from. It returns true if there were no errors, and false if there was an error. If DEBUG_PRINT is set, then it will print out the error code if there was an error.

`nHANDLE GetEventHandle()`
This method returns the Event Handle which when used in Windows will signal when the I/O has completed.

`static unsigned char *AllocateBuffer(string device, long size)`
This allocates a sector-aligned buffer of length size. It also takes the device so it can determine what the proper size of a sector is on the given system.

`bool GetResult()`
This returns true if the I/O has completed.

`bool GetBytesRead(DWORD* size)`
This returns true if the method succeeds. It fills the size variable with the number of bytes read, assuming the I/O completed.

`bool GetBytesWritten(DWORD* size)`
This returns true if the method succeeds. It fills the size variable with the number of bytes write, assuming the I/O completed.

`static long GetSectorSize(string device)`
Returns the sector size of the drive named in the string device.

`static BIG_INT GetLastAddress(string device, bool isRaw)`
Returns the last possible LBA or file offset, based on whether it is raw I/O or not, and some information queried from the disk.

`HANDLE access`

Internal file handle used for Windows API calls `ReadFile` and `WriteFile`.

`OVERLAPPED *overlap`
Internal overlap data structure used for Windows API calls `ReadFile` and `WriteFile`.

DPHandler

`static Pattern getPattern(string name, string device)`
This function creates the pattern either from a pre-specified pattern that is generated at runtime or from a pattern file. The pattern is then returned.

`static Pattern inputFilePatt(string fileName)`
This function inputs the pattern from the filename that is specified.

`static Pattern inputFilePattHex(string fileName, string dev)`
This function inputs the pattern from the filename that is specified in the proper hexadecimal format.

Error Generator

`ErrorGenerator(bool isRaw, char* device)`
This constructor takes two arguments. The Boolean, isRaw, determines whether this `Device I/O` object is going to perform requests on a raw drive or file system drive. The string is the name of the drive / file to read from or write to.

`void Seed(long threadID, long NumIO)`
This method sets the seed for randomly choosing error generation types.

`bool nWrite(char* dataBuffer, DWORD bufSize, BIG_INT lba)`
This method takes in a data buffer to write to disk, the size of the buffer, and the lba for raw I/O and offset for file system. Internally, it calls CorruptData with the size and buffer parameter, and may randomly choose to perform address or I/O size errors. See *ERRORTYPE thisErrorGen* member for more info. It then delegates the nWrite call with the potentially modified buffer to the internal DeviceIO member. It returns true if there were no errors, and false if there was an error. If DEBUG_PRINT is set, then it will print out the error code if there was an error.

`bool nRead(char* dataBuffer, DWORD bufSize, DWORD *readBytes, BIG_INT lba)`
This method takes in a data buffer to read data from disk into, the size of the buffer, and the LBA for raw I/O and offset for file system to read from. Internally, it delegates the nRead call with the unmodified buffer and may randomly choose to perform address or I/O size errors. See *ERRORTYPE thisErrorGen* member for more info. It calls CorruptData with the size and buffer parameter when the read has returned. It returns true if there were no errors, and false if there was an error. If DEBUG_PRINT is set, then it will print out the error code if there was an error.

`void CorruptData(char* dataBuffer, DWORD bufSize)`
This method takes in a data buffer, and the size of the buffer. It then randomly selects a type of data corruption to perform (see *DATACORRUPT thisDataCor* member). It then corrupts the data (assuming that NOERR was not selected), and returns.

`DeviceIO* delegate`
This member is the delegate DeviceIO which handles all of the underlying I/O functionality of the Error Generation module.

`ERRORTYPE thisErrorGen`
This could be one of:

- ADDRESSERR
    - This will create an error in the Address to transfer data to/from. It will modify the LBA/Offset in some way similar to the errors caused in the actual data.
- IOSIZE

    - This will create an error in the Size parameter for data to transfer. It will decrease read size, or increase/decrease write size. For an increased write size, will zero fill the increase of the buffer to allow more to be transferred.
- RDATA
    - This will create an error in the data buffer after being read. See the *DATACORRUPT thisDataCor* member for more information.
- WDATA
    - This will create an error in the data buffer before being written. See the *DATACORRUPT thisDataCor* member for more information.
- NOERRGEN
    - As its name implies, this will cause no error generation to occur.

`DATACORRUPT thisDataCor`

This could be one of:

- NOERR
    - As its name implies, this will cause no data corruption to occur.
- GARBAGE
    - This will replace the entire buffer with garbage characters that have no relation to the prior contents.
- FLIPBITS
    - This will flip the bits entirely (0x00 becomes 0xFF, etc.) of one randomly selected byte.
- SWAP2BYTE
    - This will swap two randomly chosen bytes.
- SWAP2WORD
    - This will swap two randomly chosen words.
- SWAP2DWRD
    - This will swap two randomly chosen double-words.
- SHLBYTE
    - This will shift the entire buffer left by some number of bytes.
- SHLWORD
    - This will shift the entire buffer left by some number of words.
- SHLDWRD
    - This will shift the entire buffer left by some number of double-words.
- SHRBYTE
    - This will shift the entire buffer right by some number of bytes.
- SHRWORD
    - This will shift the entire buffer right by some number of words.
- SHRDWRD
    - This will shift the entire buffer right by some number of double-words.

Integrity Tester

`nHANDLE ITGo(ITModule* itMod)`
This returns a handle to the new IntegrityTester thread that is spawned. It is passed an ITModule which is the queue of items to be verified which Integrity Tester pulls from, and from which the BruteThread adds results to.

`static void Test(Result *r)`
The Integrity Tester thread will constantly take items from the ITModule queue, and call Test() on them. This will do minor data verification, and on error, it will change the Outcome to FAILURE and call the Triage Mode.

`static TYPE FinalOutcome()`
This returns the FinalOutcome shared among all the Integrity Testers. If one finds a failure, the whole test is a failure.

`static TYPE Outcome`
The internal data member that stores the final Outcome of the test – Success or Failure.

IOControl

`static void nPuts(string str)`
This function outputs a string with a newline.

`static void nPuts(Pattern *p)`
This function outputs the `Pattern` structure in the form of:
Pattern : DATA
Message : DATA

`static void nPuts(Result *r)`
This function outputs the `Result` structure in the form of:
THE REQUEST:
THE PATTERN:
Result: DATA
Device: DATA
ID: DATA
NumIO: DATA
QueueDepth: DATA
TimeElapsed: DATA

`static void nPuts(LogEntry *le)`
This function outputs the `LogEntry` structure in the form of:
ERROR
ID = DATA
Address = DATA
Device = DATA
NumIO = DATA
PatternName = DATA
PatternSize = DATA
THE REQUEST:
ElapsedTime = DATA

`static void nPuts(Summary *s)`
This function outputs the `Summary` structure in the form of:
Results
Overall result was a DATA
The test was performed on DATA
The test ran for DATA
The test performed DATA
The test used DATA
The test wrote log to DATA
The test used DATA as the Random Number Seed.
Test performed on DATA

`static void nPuts(Test *t)`
This function outputs the `Test` structure in the form of:
RunTime : DATA
LogFile : DATA
JOB DATA…

`static void nPuts(Job *j)`
This function outputs the `Job` structure in the form of:
PATTERN DATA
REQUEST DATA
Device : DATA
NumRequests : DATA
Runtime : DATA
QueueDepth : DATA
ID : DATA
ReqPercentages : DATA
Repetitions : DATA
isRaw : DATA

`static void nPuts(Request *r)`
This function outputs the `Request` structure in the form of:
name : DATA
startAddr : 0x DATA
endAddr : 0xDATA
RAND : DATA
Add Increment : 0x DATA
IOsize : DATA

`static string nGets()`
This function inputs a string with `cin` and returns it.

IT Module

`ITModule(int)`
Proper constructor. Parameter is the depth of the queue of Result structures to verify.

`int getQueueSize()`
Returns the size of the queue.

`bool addQueueItem(Result *x)`
Adds an item to the queue to be verified. Increments the curDepth member. The queue is implemented as a linked list.

`Result* getNextQueueItem()`
This removes an item from the queue and returns it, decrementing the curDepth member.

`bool getStatus()`
Returns the status of whether or not the queue should be quitting. This is set from BruteThread when it wants to communicate to Integrity Tester to stop.

`void setFlag(bool status)`
BruteThread uses this to inform Integrity Tester to stop.

`Node* head`
The head of the linked list used for the queue.

`Node* tail`
The tail of the linked list used for the queue.

`int queueDepth`
The max size of the queue.

`int curDepth`
The current size of the queue.

`bool dontQuit`
The current status as to whether or not Integrity Tester should quit. It checks this status while removing items from the queue.

Nallogator

`nAllogator(int size)`
This is the constructor of `nAllogator` that takes in the number of `Jobs` (the number of threads) in the `Test` and creates the `nAllogator` object.

`ULONGLONG nAlloc(ULONGLONG begin,ULONGLONG limit, ULONGLONG length, long threadID, long serial, bool isRAND, ULONGLONG lastLocation, ULONGLONG AddIncrement)`
This function returns start the address of allocated section with < 0 indicating an error.

`int deNAlloc(ULONGLONG start, ULONGLONG length, long threadID, long serial)`
This function will de-allocate the space that was allocated to that particular request. It returns 0 on success, -1 on not found, and -2 if there is an unknown error.

`void PrintOut()`
This function prints out the hash table of all the used blocks.

`void RemoveThread(long threadID)`
This function effectively removes a thread from the hash.

`void ResetHighLow(long threadID)`
This function resets the entries highest and lowest variables for the particular thread ID.

`ULONGLONG FindChunk(long threadID,ULONGLONG start,ULONGLONG end)`
This function checks if the memory group chosen is free to be used.

`void DestroyLL(node *n)`
This function destroys the hash table that is passed in.

`HKey **hash`
This is the member variable that contains the hash table.

`int numThreads`
This is the member variable that contains how many threads there currently is.

`nHANDLE m`
This is the member variable of the file handle to the mutex.

`catcher mut`
This is the mutex object that is used to lock the `nAllogator` class.

Pattern

`Pattern(string dev)`
This is the constructor that takes in the device name and sets up all the appropriate default values for the `pattern` class.

`void setSize(long size)`
This function sets the size of the private data member `mySize`.

`long getSize()`
This function returns the value of the `mySize` private data member.

`char *getNextSegment(long size)`
This function returns the next segment of pattern with the given length passed in as the parameter.

`char *getNextSegment(int start,long size)`
This function returns the next segment of pattern with the given length passed in and the start location of the pattern as the parameters.

`char *getLBASegment(string start,long IOsize,long serial,string device)`
This function generates an LBA pattern with the iosize and serial number specified.

`char *makeLBAPattern(string lba,long serial, long offset)`
This function makes the LBA pattern with the specified parameters.

`int getIndex()`
This function returns the value of the private data member `lastLocation`.

`int lastLocation`
This private data member stores the last location in the pattern that was generated.

`int mySize`
This private data member stores the size of the pattern that is to be generated.

`string device`
This private data member stores the device to operate on.

Scribe

`Scribe(string filename)`
This function calls `PrepareLog` and creates the scribe `mutex` object.

`void Log(LogEntry *le)`
This function locks the mutex then outputs the log entry to the log which includes the invalid tag and all information included (request, runtime, pattern information, device, address, ID, etc.) with an invalid tag and it also outputs to the debugger if that option is specified.

`void PrepareLog(string filename)`
This function adds the opening XML tags needed to the log file.

`void Log(Test *t)`
This function locks the mutex then outputs the Test tag and all appropriate information with that (all the jobs, runtime, logfile). It also outputs this same information to the debugger.

`void Log(Summary *s)`
This function locks the mutex then outputs the success attributes and information to the log (overall results, device, elapsed time, total number of IOs, configuration, log-file, seeed, and date). It also outputs this same information to the debugger.

`void Log(const char *str, long length)`
This function locks the mutex then outputs char by char to the log-file the distance specified by the length that is passed in as an argument.

`void Close()`
This function adds the closing </log> tag and close the log-file. It also closes the mutex as well.

`void Log(Pattern *p)`
This function locks the mutex then outputs the pattern tag and the pattern message. It also outputs this same information to the debugger.

`void Log(Result *r)`
This function locks the mutex then outputs the result tag and all appropriate information with that (address, time elapsed, total number of IOs, device, ID, and queue depth). It also outputs this same information to the debugger.

`void Log(Job *j)`
This function locks the mutex then outputs the Job tag and all appropriate information with that (the requests, device, number of requests, runtime, queue depth, ID, request percentages, and repetitions). It also outputs this same information to the debugger.

`void Log(Request *q)`

This function locks the mutex then outputs the Request tag and all appropriate information with that (name, randomness of addresses, start address, end address, address increment, and IO size). It also outputs this same information to the debugger.

`nHANDLE m`
This is the private data member that is the handle to the mutex object.
`catcher mut`
This is the private data member that holds the mutex object.

ThreadManager

`void SetTest(Test *t)`
This function sets the `Test` passed in for the `ThreadManager` to use.

`nHANDLE Go()`
This function launches the thread for the `ThreadManager` and returns the handle to it.

`void Stop()`
This function sets the global kill status for `ThreadManager` to TRUE.

`string GetLogFile()`
This function returns the name of the log-file that is to be written to.

`static bool Kill_Status`
This member variable holds the global kill status for all the threads under the `ThreadManager`.

`static long Finished_Threads`
This member variable is a counter that stores all the threads that have finished.

`static long TotalIOs`
This member variable is a counter that stores the total number of I/Os that were done.

`static long RSeed`
This member variable stores the random seed value.

`static bool PauseTest`
This member variable stores whether the program should stop on errors or continue.

`static bool Debug`
This member variable stores whether the debugger option is on or off.

`static Test myTest`
This member variable holds the `Test` structure that is used by the `ThreadManager` class.

ThreadWrap

`nHANDLE managerGo(Test *t)`
This is a win32 specific function that sends the `Test` structure to the `ManagerThread`. A handle is returned that is the handle to thread that is the `ManagerThread`.

`static long WINAPI wrkThread(LPVOID lParam)`
This win32 function calls the `BruteThread` and will perform the main work of the program.

`static long WINAPI ManagerThread(LPVOID lParam)`
This is the win32 thread declaration for the `ManagerThread.` This thread breaks the jobs up and sends them to their own `wrkThreads.`

TriageNurse

`TriageNurse(string log)`
This is the constructor that sets the private data member `logFile`.

`void Admit(Result *r,char* testPatt)`
This function is where the actual error analysis is done when there is a mis-compare. The data that is passed in through the `result` structure is re-read a number of times to determine what has gone wrong. This function calls the `Compare2` function.

`void ZeroOutLogEntry(LogEntry *l)`
This function clears the `LogEntry` data structure that is passed in.

`static void ConvertTime(double dtime, char *time)`
This function converts the time into an output that is of the following format: Hrs:mins:secs

`void LogMisCompare(Result *r, char *dataRead, char* testPatt, long mis_start, long mis_end)`
This function is the function that outputs to the log on a mis-compare, it stores the pertinent information about the mismatched data into a string and then sends that to be logged. The expected data and the actual data is logged here.

`bool Compare2(bool logIt,Result *r, char *latest, char* testPatt, DWORD BytesRead)`
This function checks to see if the # of bytes read is equal to what is written then it outputs what extra it has read. It also calls the function `LogMisCompare`.

`string logFile`
This is the private data member that stores the name of the log-file.