Measuring Software Dependability by Robustness Benchmarking

Arup Mukherjee Daniel P. Siewiorek

May 1994

CMU-CS-94-148

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Inability to identify weaknesses or to quantify advancements in software system robustness frequently hinders the development of robust software systems. Efforts have been made to develop benchmarks of software robustness to address this problem, but they all suffer from significant shortcomings. This paper presents the various features that are desirable in a benchmark of system robustness, and evaluates some existing benchmarks according to these features. A new hierarchically structured approach to building robustness benchmarks, which overcomes many deficiencies of past efforts, is also presented. This approach has been applied to building a hierarchically structured benchmark that tests part of the Unix file and virtual memory systems. The resultant benchmark has successfully been used to identify new response class stuctures that were not detected in a similar situation by other less organized techniques.

This research was supported in part by the Computer Sciences Corporation under contract number G.S.09K90BHD0001LFP and by the Office of Naval Research under contract number N00014-91-J-4139. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of CSC or ONR.

Keywords: system reliability, software dependability, robustness benchmarking, test suite organization, object-oriented benchmarks, software validation, extensible benchmarks.

1. Introduction

Given the current scarcity of tools to measure the robustness of a software system, operating system developers lack the means to focus their attention on issues affecting system robustness. System developers have long used suites of performance tests to aid in the development of high performance machines and application programs; we believe that a suite of robustness tests would be similarly useful in gauging the development of robust systems, and in providing a means to compare robustness among various systems. Throughout this paper many examples are presented in the context of evaluating the robustness of an operating system. However, most of the issues examined arise in evaluating the robustness of any complex software system.

A robustness benchmark is a suite of robustness tests, or *stimuli*. The benchmark should address issues that are general enough to apply to a wide range of systems yet specific enough to provide a basis for differentiation according to system robustness. Essentially, a robustness benchmark aims to stimulate the system in ways that are likely to trigger internal errors, and thereby to expose design errors in the error detection or recovery mechanisms. Differentiation amongst systems should reflect the number of such errors uncovered.

In attempting to design a useful benchmark with the most general applicability, several issues must be considered. For example, if a benchmark simulates memory faults via fault injection into the supervisor code of an operating system (as in [Kanawati92] or [Kao93]), it is not likely to be easily portable between operating systems, perhaps not even between operating systems that are very similar from an application's point of view – similar operating system interfaces are often backed by very different bodies of code. This makes it very difficult to inject faults into supervisor code in such a way that the results can meaningfully be compared across systems. This paper documents several goals that a benchmark of robustness should strive to achieve. Those goals are considered in light of the constraints presented by Unix-like operating systems. We then present the choices we have made in our initial efforts to develop a suite of robustness tests for machines running a Unix-like operating system.

After Section 2 describes the motivation for robustness benchmarking, Section 3 presents several characteristics that a benchmark of robustness should attempt to achieve on any computer system. Section 4 subsequently examines constraints and opportunites that arise in the design of a benchmark to be used solely in evaluating the robustness of Unix-like systems. Later document initial efforts into the development of robustness benchmarks and evaluate them as examples of some of the design philosophies presented herein.

2. Background and Motivation

The development of computer systems has traditionally been motivated by the desire to achieve higher performance. The need to measure progress towards this goal prompted the development of performance benchmarks, which have grown in complexity and sophistication since their inception. The original performance measures of a computer system reflected attempts to compute the average instruction execution time of that system. Later, focus shifted to attempts to measure overall system performance in scenarios designed to reflect common uses of the system. The latter approach led first to synthetic benchmarks, such as Whetstone[Curnow76] and Dhrystone[Weicker84], and then to applications oriented benchmarks such as the SPEC[SPEC90] suite, which measures performance under prototypical workloads built from a collection of real applications. Similarly, the advent of reliable computing systems is spurring the development of robustness benchmarks to quantify improvements in system reliability. As robustness benchmarking grows, its focus is also shifting from simple measures of hardware characteristics to measures that reflect the overall reliability of a computing environment (i.e. the hardware together with its supporting software).

To date, much of the effort in building robust systems has been devoted to building robust hardware. Efforts to evaluate the robustness of software systems have become common only recently, and are exemplified by such studies as [Miller90] and [Suh93]. These studies both concentrate, like all robustness benchmarks, on studying the behavior produced when a system is subjected to unusual (rather than common-case) stimuli. Both studies perform their evaluations via a collection of isolated tests, and draw conclusions from the collected results. Unfortunately, it is often difficult to evaluate the relative significance of each of the individual results collected through such test suites.

For example, [Miller90] examines the behavior of Unix utilities when they are supplied with randomly generated input data. The crash of any single utility must be taken as seriously as the crash of any other utility, even though this weighting may not reflect reality, because the benchmark lacks knowledge of the underlying system's structure (so it cannot know if two utilities are related in any way). Thus, if several utilities crash due to a bug in an underlying shared system library, the robustness of the system being measured might be perceived to be unduly low — the robustness benchmark is affected by a lack of knowledge of the system's structure. [Miller90] and other such studies are similar to synthetic benchmarks in the performance arena — The validity of these benchmarks depends on the accuracy with which they are constructed to emulate the normal workload of the system. In the case of robustness tests, emulating the "normal workload" refers to maintaining the same frequency distribution of exception conditions that occurs in the system's normal use. This distribution must be maintained to obtain an accurate assessment of the system's robustness when in normal use.

In order to allow more accurate evaluations of overall system robustness, robustness benchmarks must evolve towards a structure that embodies the dependency hierarchy of the system. Benchmarks should allow testing at multiple levels of abstraction, in order to facilitate the isolation of sources of failures, and to help evaluate the severity of any failures encountered. Robustness benchmarks should also be easy to adapt to any new facilities added to an existing system. For example, if a module is added to a software system and is conceptually similar to an older module, test procedures developed for the older module should be easily adaptable to the new module to ensure that such expertise is not lost. Present suites of robustness tests do not address these issues. In this paper we consider how these goals may be achieved without precluding any of the desirable properties found in existing benchmarks of robustness.

3. General Design Issues

Several issues must be addressed by the designer of any suite of robustness benchmarks. This section presents many desirable properties and mentions tradeoffs which may be necessary to attain them.

Portability: First and foremost, it must be possible to use the robustness benchmarks to compare different operating environments and computer systems, and thus the benchmarks should be portable across platforms. This goal often restricts the range of tests that can be performed – notably those that require knowledge specific to one environment, such as is often the case with fault-injection based tests.

Coverage: Ideally, a benchmark should test all possible uses of every system module being tested. Often, however, the space of such stimuli is too large to permit exhaustive testing. A completely deterministic benchmark may choose to test only the most frequent uses of a module, but experience shows that the common case uses are most often those that are properly debugged. Such a benchmark functions solely as a verification suite. Alternatively, a deterministic benchmark may focus solely on unusual uses of a module, thereby providing a better assessment of the robustness of that module. However, problems usually occur at the intersections of rarely occurring events which, taken together, produce an unexpected state. The set of possible event intersections is often too large to explore systematically. Thus, such a benchmark remains limited in its coverage, and is likely to be useful mainly as an aide to debugging the uncommon cases.

A more realistic estimate of the robustness can be obtained by the use of randomized stimulation.

Randomness: Randomized stimulation attempts to uniformly cover the space of possible uses of a module — randomized tests usually have a higher *serendipity* (i.e. ability to uncover previously unknown errors) than their deterministic counterparts. Note, however, that the nondeterminism introduced by randomized stimulation may lead to loss of repeatability. Sometimes randomized stimulation is required in order to adequately emulate a system's computational model, as is explained in the next section. Some degree of randomness without loss of repeatability is one motivation for an extensible set of benchmarks.

Extensibility: An extensible benchmark provides a means to extend its set of stimuli in a *consistent* manner, i.e. stimuli can be added to the benchmark in such a way that produces results that are directly comparable with results generated prior to the addition. Extensibility is necessary not only to allow for the addition of stimuli of a different nature to be used in testing an existing system module, but also to allow for an existing benchmark to be extended to apply to new system modules. Extensibility ensures that a benchmark is a consistent measure of progress, rather than simply a verification suite for an isolated module.

A simple form of extensibility can be achieved through the use of parameterized stimuli. For example, a benchmark suite might consist of a group of stimuli whose behavior is completely determined by the input of a string of random numbers. New sets of tests can be generated (thereby increasing coverage) by varying the input string, whereas every test set generated maintains repeatability. The extensibility of such a benchmark is rather restricted, however, as only limited variation can be achieved in tests by varying only input parameters. Greater extensibility requires the ability to add completely new testing code while maintaining the consistency of the result processing.

In general, the extensibilty of a benchmark is determined by the degree to which the test control structure of the benchmark can be extended without affecting the result processing. Hierarchically structured benchmarks provide a general means of achieving extensibility.

Hierarchy of Complexity: The tests within a set of benchmarks should be organized in order of increasing complexity, where complexity is inversely proportional to the number of modules that can be exercised by a test. The simplest tests are often applicable across multiple system modules (e.g., there are tests of proper resource allocation and deallocation that are applicable to any system module that manages a resource), whereas the most complicated tests are usually highly specific to a particular module or combination of modules. This organization may be reflected in the tests themselves, such that the most complicated tests can assume that all simpler tests have been passed. In designing a benchmark suite that tests several system modules, it may be desirable to develop a hierarchical interface to those system modules, organized to reflect the hierarchy of their functions – simple tests can then be written in an object-oriented fashion without requiring code duplication (possibly leading to divergence) on a per-module basis. An example of this approach to benchmarking is documented in Section 5.4.

Note that an arrangement of tests in a hierarchy of complexity may lead to a higher *initial implementation cost* — Some effort is required to define the structure of such a benchmark, unlike the construction of a benchmark composed of a collection of ad hoc tests. However, this initial investment is usually worthwhile due to the desirable properties of a hierarchical structure such as the resultant ease of code reuse when the benchmark is extended.

Reporting of Results: Several characteristics are desirable in the results reported by each test of robustness. As mentioned before, each test result should be repeatable. The results should also be amenable to comparisons between different machines – indeed, rather than simply indicating whether each test of robustness was passed or not, it is desirable to report on failed tests using a scale that reflects the severity of the failure. A closely related issue is the amount of *localization of triggering events* [Sullivan91] that is reflected in the reported results — i.e. the extent to which the results pinpoint the error(s) that were detected, and their possible causes. Good localization is especially valuable to system designers trying to focus on improving the depenability of the operating system.

Note that detection of a "failure" (i.e. failed test) in itself raises a number of difficulties. As many stimuli exercise the system in ways that may not have been anticipated by its creators, a "golden standard" for correctness is often absent. The developers of a robustness benchmark might choose to overcome this problem by defining a standard of correctness, or even a measure of incorrectness. For example, a scale ranking errors in terms of their severity, ranging from unanticipated error code returns to complete system crashes, could serve as a yardstick of incorrectness. Alternatively, the benchmark itself could be augmented with the ability to learn and record a "correct" result. For example, the developers of a benchmark may define the correct result to be the result most commonly produced in a particular abnormal scenario[1] — The benchmark itself can then be used to determine the most common results. A third possible approach involves defining only the possible "incorrect" results, and assuming that anything else is correct. (For example, the effects of a system call invoked with garbage parameters could be defined as being correct as long as the operating system does not crash, the file system is left intact, and other executing processes are unaffected.)

A benchmark suite may also elect to compute an "index of robustness" from the individual test results. Such a result serves to provide a very high level means of comparing two machines, and is generally a weighted average of the individual test results. The set of weights used generally

[1] Such an approach measures behavioral consistency of modules across the domain of the test.

reflects the perceived severity of each of the errors detected, and may depend upon the number of system modules affected by the error, its likelihood of occurrence in daily operation, the range of applications affected, the specific type of error involved, and whether the system was only able to detect the error, or whether it detected and corrected the error.

4. Benchmarks for Unix-like Systems

Unix-like operating systems are available on a wide spectrum of hardware platforms ranging from personal computers to supercomputers. As all of these operating systems attempt to provide similar interfaces and functionality, benchmarks can be written for the purposes of comparing their robustness. This section describes opportunities and limitations that constrain the development of a benchmark suite designed specifically to test Unix-like operating systems. As such, most of the issues raised are in addition to those presented in the previous section. Note that many other multi-user multitasking timeshared operating systems present similar constraints; we attempt to point out features of Unix that are relevant to robust benchmarking efforts, although most do not restrict the applicability of the work to other non-Unix operating systems.

4.1. Goal of a Unix Benchmark

Unix-like operating systems are primarily used to support applications that require support for multiple processes and multiple users. Even in "single user" setups, processes owned by at least two different users are usually present on the system (processes owned by the system administrator, and those owned by one or more users). The function of the operating system is to manage access to hardware resources, and to ensure that the running processes do not affect each other adversely. This suggests that the robustness of an operating system should reflect the system's ability to successfully contain fault conditions generated by any one process (i.e. reflect the ability of the operating system to prevent those faults from affecting other processes). Thus, a system crash is considered to be the extreme case of failure to localize a fault, as it means that all other processes are affected. A Unix benchmark suite should thus attempt to measure the ability of each module to contain errors, on a module by module basis for each of the several testable modules and interfaces existing on most systems.

4.2. Benchmark Structure

4.2.1. Gross Structure

Unix systems provide at least the following modules together with their interfaces: file system, virtual memory, process management, and signal handling. Most also provide network support and window management. A simple benchmark suite might consist of a series of independent test programs, each of which exercises one module. However, such a test does not accurately reflect the fact that under normal use, each system module must support simultaneous interaction with several programs. Thus multithreading, or support for running and monitoring several simple programs simultaneously, is required for a representative test of fault handling scenarios that may arise in regular use. Note that a multithreaded benchmark is also able to test the system's ability to handle the propagation of multiple faults simultaneously occurring in distinct modules. Thus, a robustness benchmark for Unix-like systems should have multiple threads — a feature that also proves convenient in measuring the extent to which a fault propagates (described later).

There is another disadvantage in designing a benchmark suite made up of one test program per system module. A system may have several modules, and the incremental cost of adding a new, widely applicable test to the test suite is high — the new test must be implemented once for every module involved. However, if the new test simply manipulates modules in a manner that can be abstracted beyond a standardized module interface, the test need only be coded once; module-specific code below the interface handles all interaction with the system modules and is unchanged. This is the motivation for hierarchical structuring of benchmarks. Unix systems support enough modules to justify implementing a hierarchically structured system interface to allow tests to be be coded in an object-oriented manner, without code duplication. (For an example of such an implementation approach, see Section 5.4.) Note also that a hierarchical approach enforces consistency of result reporting simply by eliminating multiple copies of functionally similar testing and reporting code — each test is implemented exactly once at an abstract level, thereby guaranteeing compatibility of test reports across modules.

Finally, it should be noted that portability considerations often require that the code of a benchmark suite be limited to a user-level implementation. Unix-like kernels often differ substantially in their implementation. Had this not been the case there would presumably be little or no difference in robustness among the various flavors of Unix. Thus any benchmark that requires kernel-level support is not likely to be easily portable across a wide range of Unix platforms.

4.2.2. The Measurement of Faults

The following criteria might be used to evaluate the seriousness of a fault condition (in increasing order of severity):

1. Does the fault affect the process causing it?[2]
2. Does the fault affect other executing processes?
3. Does the fault crash the operating system?
4. Does the fault crash the operating system microkernel?[3]

An uncontained fault may affect another executing process in one of several ways. It may cause the other process to crash (without having crashed the entire operating system), to produce incorrect output, or to simply to execute more slowly than it otherwise would. Note that the process causing the fault may be affected in a similar manner; however, if the causing process is the benchmark itself, the benchmark will not be able to detect the fault without the aid of an external monitoring agent, such as a watchdog started before beginning the tests.

[2] In practice, this measure is limited by a benchmark's ability to detect changes made to its own state.

[3] Of course, this applies only to systems that are built on top of a microkernel.

Barton	Cristian	Suh
Response too late	Timing (early/late)	Timeout (late response)
Invalid Output	Response (value/state)	Failure (incorrect answer)
Crash	Crash (partial/total amnesia, pause, halt)	Crash
Task stop (process crash)		Abort (crash w/ error message)

Table 1: A comparison of failure classifications.

The possible effects of a fault may be classified according to any of several taxonomies, such as those of [Barton90], [Cristian91], or [Suh93], which are summarized in Table 1. All of these taxonomies necessitate a means of measuring the effect on processes other than those owned by the benchmark itself. This may be done by observing a "sacrificial program" which is executed concurrently with the benchmark suite, and checking to see how it is affected by the faults generated by the benchmark suite. Note that almost all of the possible resultant states of the sacrificial program (as enumerated by Table 1) can be detected mechanically by a "watchdog" program that has been previously calibrated to the expected behavior of the sacrificial program. If the fault should produce a complete system crash, however, the watchdog may be unable to observe this fact, unless it is executing on a separate processor that is isolated from the one used to execute the tests. If a separate processor is not available human intervention may be required in the event of a system crash.

A sacrificial program should, of course, make widespread use of the system to increase the probability that it will reflect any effects of uncontained faults. A robustness benchmark suite might elect to provide a synthetic program to serve as a sacrificial program, or it may choose to make use of any of several performance benchmark suites, such as the SPECmarks, which have the advantage of widespread availability. If a system is being evaluated for its ability to run one particular application without failure, that application itself might well serve as the sacrificial program.

It should be pointed out that the use of a sacrificial program has a marked effect on some of the properties of the benchmark. In particular, any robustness benchmark employing a sacrificial workload is a multithreaded benchmark and as such suffers all the disadvantages of being multithreaded (e.g. the resultant benchmark is likely to lose both determinism and repeatability). However, as mentioned earlier, multithreading is also more representative of the application computing model supported by the operating system. The use of multithreading is discussed further in a later section.

4.2.3. Recording of Results

Most Unix-like systems provide no stable data repository, so a robustness benchmark must implement a means of recording results in the face of adverse conditions produced by the testing. As the possible effects of a test are often unknown, this can be rather difficult to implement. Any buffered output channel is susceptible to data loss in an operating system crash; most of the output channels available to a user-level process fall into this category. A simple but non-automated way to overcome this problem involves printing results to an unbuffered CRT or printer port monitored by a human. A similar effect might also be achieved through the use of an unbuffered serial line output to communicate results to a second "watchdog" processor. If a serial line is not available, the same arrangement, with a greater chance of losing a small amount of data, can be approximated by communicating results to a second computer over a local area network. In the last case, logging with variable granularity (i.e. if a test causes a crash, the test is repeated while synchronously writing the log to disk more frequently than before) may help to reduce the amount of data loss, although it cannot completely overcome the effects of data buffering by the kernel.

4.2.4. Randomness and Extensibility

In order to provide good coverage of the test space, a robustness benchmark may opt to use tests whose behavior is dependent upon the output of a random number generator. However, such a test may not execute exactly the same actions on different systems if the output of the random number generator changes. Most Unix systems provide a random number generator interface (`random()`) that is identical across implementations, but which is not guaranteed to produce exactly the same stream of random numbers from machine to machine[4]. This brings into question the value of comparing randomized runs made on two different machines, a problem that can be resolved through "controlled randomness" whereby streams of random numbers are pregenerated and stored in a file in advance. These pregenerated numbers are then fed to the benchmark at run time, thereby ensuring that benchmark runs on two different machines behave in a manner determined by identical sets of random numbers. Note that the Unix random number generator produces the same stream of random numbers from run to run on any given machine (for a given seed) and thus does not affect repeatability of results on a single machine.

Nevertheless, repeatability is often difficult to achieve in any realistic robustness benchmark of a Unix system because Unix systems do not usually provide deterministic process schedulers. Thus, any test that presents the system with a multithreaded workload (in order to evaluate the system's ability to handle multiple simultaneous requests) will introduce some randomness into the result of the robustness test. Consequently, the results of any such test may not always be repeatable, as the scheduling order is likely to vary from run to run. This problem cannot usually be resolved in a portable manner if a robustness suite wants to test a multitasking environment. Unfortunately, Unix schedulers do not normally provide hooks to allow repeatable or deterministic process scheduling. A multithreaded benchmark scheduling its own threads may reduce the severity of this problem, but does not provide a complete solution because the benchmark as a whole remains subject to the scheduling actions of the system-wide scheduler.

4.3. Summary

Unix environments provide a level of programming support adequate for benchmarks ranging from simple tests based on the perturbation of an input string to more complex hierarchical tests which are extensible at both abstract and module-specific levels. When the requirements of robustness benchmarks are better understood, Unix may well provide a testbed for the development of a

[4]In practice, most implementations do produce identical streams, but a system vendor might choose to change this.

	Portability	Coverage	Extensibility	Consistency of results	Initial implementation cost	Localization of triggering events	Repeatability
Randomness	$\times$	$\Uparrow$	$\times$	$\times$,$\Downarrow$	$\times$	$\times$,$\Downarrow$	$\Downarrow$
Multithreading	$\times$	$\Uparrow$	$\times$	$\times$,$\Downarrow$	$\times$	$\times$,$\Downarrow$	$\Downarrow$
Hierarchical structure	$\times$	$\times$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\times$	$\times$
Logging	$\times$	$\times$	$\times$	$\times$	$\Uparrow$	$\Uparrow$	$\times$
Kernel fault injection	$\Downarrow$	$\Uparrow$	$\times$	$\times$	$\Uparrow$	$\times$	$\times$

Table 2: The effects of implementation choices on benchmark characteristics. Postive correlation, inverse correlation, and independence are indicated by $\Uparrow$, $\Downarrow$, and $\times$respectively. Where more than one relationship is indicated, the actual relationship depends on the specific implementation.

	`crashme`	`CMU crashme`	Modular	Hierarchical
Portability	Unix	Unix	Similar modules (similar implementations only)	Similar modules (similar abstractions)
Coverage	High	System calls	Variable/Local to module	Variable
Serendipity	High	High/Limited to syscalls	Variable/Local to module	Variable
Extensibility	None	None	Difficult	Easy
Localization	None	Possible (via sentries)	High	High
Repeatability	Low	Low	Variable	Variable

Table 3: The properties of various example benchmarks. Properties marked "variable" are unconstrained by the design of the benchmark, and vary between the individual tests in the benchmark.

language in which robustness tests can be expressed with minimal effort. At present, Unix environments provide modular interfaces which are can be organized into a hierarchy. This permits hierarchically organized benchmarks, from which new benchmarks can be derived with only a small amount of effort via inheritance. An initial approach to such an implementation is described in Section 5.4.

Table 2 summarizes the relationships between the properties of a benchmark and the implementation choices made while constructing it. As can be seen from the table, these implementation choices affect most of the important characteristics of any robustness benchmark. The next section provides several examples of benchmarks exhibiting the tabulated relationships.

5. Some Examples of Robustness Benchmarks

This section presents initial efforts in producing benchmarks of system robustness. Each example is evaluated with respect to the design issues presented in Section 3. The properties of all the

Hardware	Operating System	Time to Crash (approx)
IBM/RT	Mach 2.5	3 sec
IBM/RT	Mach 2.6	60 sec
i486	Mach 2.5	5 sec
i486	Mach 3.0 (MK76)	4 sec
i486	Mach 3.0 (MK82)	50 sec

Table 4: Time taken by `crashme` to crash some machines.

benchmarks are summarized in Table 3 and are described in more detail below.

5.1. "Crashme"

`Crashme` is a simple, publicly available test of robustness for Unix systems. This program allocates an array, and fills this array with random data. Subsequently it spawns off several child processes that all try to execute this array of data as if it were code. The parent crashme process observes its children and spawns replacements for the children as they take exceptions and die. When crashme is run, the Unix system is subjected to a large number of varied exception conditions in a short period of time — as a result both the error detection and handling capabilities of the operating system are severely tested. Crashme succeeds in crashing a large number of Unix systems. Albeit from a very small sample of machines, we observed that the amount of time an operating system stayed up under crashme appears to be correlated to some degree with our observed reliability of that operating system in day to day use. (Refer to Table 4.)

`Crashme` is a very good test of a system's ability to handle a high error rate, but in many ways it is not a good general purpose benchmark of robustness. Although it is portable and has good coverage of the stimulus space (and a correspondingly high serendipity), the results from `crashme` are very limited — either the system crashes, or it does not crash. If the system crashes, it is very difficult to determine the cause of the crash (and any such determination could only be a result of error logging external to crashme itself, such as error logging provided by the operating system). `Crashme` also lacks repeatability — a high degree of randomness is introduced due to the scheduling of a large number of child processes created by the program. If the test is run twice, and the system crashes both times, it cannot be determined whether or not both crashes shared a common cause. Due to the difficulty of interpreting its results, `crashme` is only of limited usefulness as a measure of progress in building a robust system.

5.2. CMU Crashme

Having observed the aforementioned problems with `crashme`, we attempted to remedy them by restricting the coverage of the test, hoping to gain repeatability and better localization of triggering events. The spawned child processes were constrained to exercising only a single well-

Hardware	Operating System	Time to Crash (approx)
IBM/RT	Mach 2.5	30 sec
IBM/RT	Mach 2.6	no crash
i486	Mach 2.5	no crash
i486	Mach 3.0 (MK76)	10 sec
i486	Mach 3.0 (MK82)	no crash*

* became unusably sluggish after 30 secs

Table 5: Time taken by `CMU crashme` to crash some machines

defined system interface — namely, the Unix system calls. It was anticipated that the error checking on parameters passed to system calls would be sufficient to guarantee that system calls made with randomly generated parameters would not be able to crash the operating system. Much to our surprise, many of the systems tested were vulnerable to this limited test. (Refer to Table 5.)

Although this modified version of `crashme` still exhibits a high degree of nondeterminism, it offers better localization of triggering events than the original version. This localization can be further improved by restricting the tests to only a subset of the Unix system calls. Such restriction, together with the monitoring of system calls via the sentry mechanism described in [Russinovich92], has been used successfully to identify some errors in the Mach 3.0 Unix server.

5.3. Modular Benchmarks

Another approach is that of *modular benchmarking.* Modular benchmarks are separate tests of individual system modules. These benchmarks are constructed by regarding the system as a collection of isolated modules, and writing one or more tests to exercise each module independently. One example of a modular benchmark is documented in [Suh93]. Another example is a set of robustness benchmarks that was recently constructed at CMU to test the robustness of the Advanced Spaceborne Computer Module (ASCM)[Dingman93]. Although this is an embedded system running a real-time operating system not related to Unix, it can be regarded as a collection of system modules in much the same way as any other operating system. The ASCM test suite consisted of distinct tests to exercise the various system modules (file system, memory system, external communication, locking support and multi-program operations) together with a watchdog program (similar to the parent `crashme` process) to monitor and collect the results of the tests.

An example of a modular benchmark

The file module benchmark from the ASCM test suite serves as a good example of a modular test. This benchmark stresses the seven calls of the file module (`create_file`, `open_file`, `close_file`, `delete_file`, `read_file`, `write_file` and `move_file_pointer`) systematically, constructing tests

File Handles	Buffer Addresses	Number of Words
Closed	Start of 16 byte buffer	1
Opened read-only	Start of 256 byte buffer	16
Open read-write	Middle of 256 byte buffer	256
Deleted	End of 16 byte buffer	1024
Altered	Beyond end of 16 byte buffer	4090
Allocated memory	Null pointer	4100
	Address of deleted buffer	

(6 file handles $\times$ 7 data buffers $\times$ 6 sizes = 252 tests)

Table 6: The various possibilities for input parameters in the `read_file` and `write_file` tests.

Operation Tested	Number of Tests	Result Class					
		`correct`	`unexpected error`	`bad success`	`terminated`	`warm restart`	`cold restart`
read_file	252	175	77	0	0	0	0
write_file	252	178	42	24	0	0	0

Table 7: The results of running the `read_file` and `write_file` tests on an ASCM system [Dingman93]. Each test case is classed as having produced the correct result (`correct`), having returned an unexpected error code (`unexpected error`), having indicated success in spite of having been given invalid input parameters (`bad success`), having caused the operating system to terminate the benchmark (`terminated`), having caused a warm restart of the system (`warm restart`), or having necessitated a cold restart of the system (`cold restart`).

for each call from the interface definition of that call. For example, the `read_file` call takes 3 parameters: a file handle, the starting address of a buffer into which data is to be read, and the number of words to be read from the file. The benchmark chooses a value for each of the parameters from a set of values that are based on the parameter's type. For example, a file handle might point at a valid file that is closed, at a valid file that was opened in read-only mode, or at a deleted file, among other possibilities. By choosing all possible test input combinations for all of the parameters of the `read_file` call, the benchmark generates 252 test cases, as shown in Table 6.

The results of the 252 tests are then divided into six groups in increasing order of severity: those that produced the expected result (`correct`); those that returned with an error code, but not one of those that would be expected given the input parameters (`unexpected error`); those that returned indicating success in spite of having been given invalid input parameters (`bad success`); those that caused the operating system to terminate the benchmark (`terminated`); those that caused a warm restart of the system (`warm restart`); and those that caused a cold restart of the system (`cold restart`). The results of these 252 tests on the `read_file` and `write_file` calls, which take identical sets of inputs (and thus generate the same test parameter combinations), are shown in Table 7.

The advantages of the modular benchmarking approach include relatively low complexity of the individual tests (because inter-module interactions are usually not considered), and the ability to guarantee determinism. Unfortunately, modular benchmarks also have several disadvantages.

Although the modular benchmark approach applies well to hardware, where system components are manufactured separately, and are often designed for independent testability, the approach does not scale well to large bodies of operating systems software, whose modules are often closely intertwined, making independent testing difficult. Here, this testing paradigm is not matched well to the system being evaluated. A different problem may occur when a modularly written system, seemingly well suited to modular testing, is evaluated. In this case, the modular decomposition of the benchmark suite restricts the coverage of individual tests, and eliminates any possibility of stimulating interactions between system modules. In addition, modular benchmarks are also unable to take advantage of the similarities between system modules. While it is quite likely that some similarities will exist between the many modules in a system, modular benchmarking requires that any similar tests applicable to multiple modules be coded once for each applicable module in the appropriate test. Thus, similarities between modules are hidden within the individual tests — a significant loss, because such similarities are the key to extensibility, as explained in the next section.

Modular benchmarks offer no guarantee that the results of similar tests, or even of two different revisions of the same test, are comparable with each other. In effect, while all comparisons are external to the benchmark (i.e. they are done by the human, or by an automated postprocessor, who collects the results of the individual tests), all information determining their comparability (the abstract nature of the tests) is completely hidden from the evaluator. This problem limits the usefulness, as a measure of progress in the development of a system, of any modular benchmark that goes beyond simple interface verification. For example, improvements to a system module may render a modular benchmark incompatible with its test simply because implementation details changed. The original test may still apply at an abstract level, but modular benchmarks enforce no separation between the abstract test they apply and the interfacing of this test to the module being tested. Consequently, a modular test often needs to be adapted in response to any significant change in a module. When such an adaptation is made, great care must be taken if the results of the adapted test are to remain directly comparable to results from the original version.

In order to illustrate the last point, consider the changes to a benchmark that might be necessitated by an incompatible upgrade of a system module (e.g., the change from version 10 to version 11 of the X window system). The old and new versions of the module provide the same functionality, and thus any tests of the old system are applicable to the new module. However, part of any benchmark code must be rewritten to accomodate the interface change. This rewriting involves "module interface code" which interfaces the system module being tested to the benchmark's test routines. If results of the updated benchmark are to remain comparable to those of the original benchmark, care must be taken not to modify code other than the module interface code (e.g., code for purposes such as result gathering and processing should not be changed). As modular benchmarks do not require separation of the module interface code and testing code, the necessary modifications may be complex, and must be performed with great care.

5.4. A Hierarchical Approach

Given the shortcomings of modular benchmarks, it would seem that decomposition of a system into multiple unrelated modules is not the best approach to organizing a suite of robustness tests. Consider the following set of identical tests, taken from the ASCM benchmarking suite. These tests appear in the benchmarks of both the file system and of the memory buffer system. `<object>` is used to denote either a file or a memory buffer in the list:

- Reference `<object>` before it has been created.
- Reference `<object>` after is has been deleted.
- Delete an active `<object>`.
- Write past the end of `<object>`.
- Read before the beginning of `<object>`.
- Allocate `<object>`s until resources are exhausted.

These tests represent just a few examples of stimuli that are applicable to multiple modules, but are hidden within individual modular benchmarks. As a result, similar tests are being performed, and their results are potentially comparable to one another, but this may not be apparent in the results.

One way to remedy this problem is to abstract the tests and their associated result processing, separating them from the implementation details of the various modules by a clearly defined interface layer. However, this one-step decomposition is not sufficient, because it does not delineate the range of applicability of any given test. Some tests may be applicable to all modules, while others might only apply to a subset, etc. We believe that the correct way to decompose a software system in order to test it is through the use of a *class hierarchy*. Once all of a system's features are organized into a hierarchy of classes, a test can be specified to apply to one or more particular classes. One possible class hierarchy that might be used to organize the testing of a Unix system is shown in Figure 1.

Note that the modular ASCM benchmarks described earlier actually represent an example of a simple hierarchy with only one level of abstraction. The ASCM benchmark generates tests for a module by looking at the *interface* to that module — sets of test input parameters, selected so as to be of the correct data types, are chosen for each call implemented by the module. For any input parameter of a given type, there is a predetermined list of inputs that may be used to instantiate that parameter. Thus, one can think of the interface to a call in a module as being a class inheriting from a set of base classes, where each base class corresponds to one input parameter type. The class corresponding to a particular call inherits from all of the input parameter types that describe its arguments, and the test applicable to such a class is composition of tests applicable to each of its base classes.

5.4.1. A proposed hierarchy

As the management of various resources is a primary function of any operating system, Figure 1 is one possible starting point in the construction of a benchmark of robustness. Note that the

Figure 1: Part of one possible hierarchy. Abstract classes, representing abstractions, are in boldface, while ordinary (nonabstract) classes, representing system modules or data types, are not.

hierarchy contains many abstract classes (e.g., `Storage Object`) which do not correspond to any single system module, but rather serve as the fundamental means of grouping modules by similarity. This explicit grouping is the basis for the organization of a hierarchically structured benchmark suite. It also provides the mechanism for orderly extension of the benchmark suite.

In order to construct a hierarchical benchmark, a hierarchically structured interface to the operating system must first be implemented. Unix does not provide such an interface, so a hierarchical interface library must be developed to support the subsequent construction of robustness tests. The construction of such an interface proceeds as follows. Having decided upon the hierarchy to be implemented, the designer must choose an appropriate set of methods to be defined in the interface to each class. For example, the methods `allocate` and `deallocate` might be defined for the top-level class `Resource`. `Shared_Resource`, which inherits from `Resource` might add the methods `lock` and `unlock` to those already defined by `Resource`. Great care must be taken to define these methods with sufficient generality to apply to all modules that will be their descendants. An instance of a particular class (e.g. `File`) is required to implement all methods from all abstract classes that are its ancestors. Whenever a method can be implemented in a completely module independent way, it should be defined in the most general abstract class to which it applies. Subclasses and instances of this class may chose to redefine the default implementation with a more specific one.

Note that operating systems of the future are likely to provide object-oriented hierarchical interfaces to their facilities and will thereby eliminate the need to construct the interface library. Note also that hierarchical interfaces and benchmarks are most easily written in a language that supports object-oriented programming (such as C++), but can also be written in more traditional languages (such as C) with some extra effort on the part of the programmer. The experimental hierarchy has been written in C++.

5.4.2. Using a hierarchy to build a benchmark

Once a hierarchical interface to the operating system has been built, any operating system test not requiring module-specific knowledge can be written in an abstract manner. The test should be implemented at the most abstract possible level in the class hierarchy to which it applies. For example, a test of resource exhaustion by repeated allocation (mentioned in Section 5.4) requires only an `allocate` method from the module that it is testing. As such, it should be coded to take a parameter of class `Resource`, and to use the `allocate` method provided by all `Resource`s. Once the test has been coded for a particular class, it can automatically be applied to any of the descendants of that class. In the case of the resource overallocation test, the test can be applied to all of the system modules at the base of the hierarchy (including files and memory buffers). In contrast, consider a test that checks for correct system behavior upon writing past the end of an object. This test requires the notion that the object be able to store information, and that it have a beginning and an end. In this case, the test applies to the class `Storage_Object` and its subclasses.

Once a test has been encoded at the appropriate level of abstraction, any result processing associated with the test should be encoded at the same level. Thus the result processing is allowed to have knowledge of the specificity of the test (e.g., it might want to use such knowledge to scale the results of the test according to some weight assigned to the severity of a failure in that class). Most importantly, however, both the test and the result processing for the test are coded exactly *once*, in an abstract way. This guarantees comparability of the results obtained from applying the test to multiple modules. Because exactly one copy of the test is used for its multiple applications, there is no way for inconsistencies to arise, as might occur when multiple copies are present (e.g., with modular benchmarks).

Hierarchical benchmarks are also easily extensible in a consistent manner due to their organization. If the benchmark suite is to be extended to a include a new module, the interface to that module is encoded as a subclass of the appropriate class within the hierarchical interface to the operating system. If the new module is related in any way to existing modules, its placement reflects these relationships. Any tests that have been developed for the existing modules can be applied to the new module immediately. Again there is no possibility of duplicating test code if the new module is positioned correctly in the hierarchy, and consistency in testing and result processing is guaranteed to have been maintained across the extension.

Finally, note that while hierarchical structuring offers superior extensibility, reduction of coding costs (through code reuse) and the potential for better organization of result reporting, it is not antagonistic to achieving any of the other goals desirable in a benchmark (portability, coverage, localization of triggering events, etc). Thus hierarchical benchmarks offer some special benefits without sacrificing the desirable qualities offered by other benchmark styles.

5.4.3. Hierarchical testing using C++

Hierarchically structured benchmarks are most easily implemented in a language that provides support for inheritance. This section presents some trivial tests that demonstrate the use of C++ in building a hierarchy, and in constructing tests using that hierarchy. The tests presented are intended solely to illustrate the programming paradigm. Readers who are familiar with C++ may wish to skip to the next section, which presents a more complicated example of a hierarchical test which was actually implemented.

Given the hierarchy from Figure 1, the significant parts of a declaration of class `Resource` might look as follows:

```
class Resource {
  public:
    virtual int  allocate   (int n) =0;
    virtual int  deallocate (int id, int n) =0;
};
```

`Resource` is an abstract class (i.e. it does not correspond to any single system module), so the implementations of methods `allocate` and `deallocate` are not provided, and are hence marked "`=0`." The `Storage_Object` abstract class *inherits* from `Resource`, and supplies some additional methods:

```
class Storage_Object : public Resource {
  public:
    virtual void set_mode (object_mode mode);
    virtual object_mode get_mode();
    virtual int read_data  (char *buf, int len) =0;
    virtual int write_data (char *buf, int len) =0;

  protected:
    enum object_mode mode_spec; // mode of object
};
```

The `Storage_Object` class provides two more methods (`read_data` and `write_data`) that will be implemented by classes inheriting from it. Default implementations are also provided of two methods that manage the mode of the storage object. Classes inheriting from `Storage_Object` may override the definitions of `set_mode` and `get_mode` if they so desire, as these methods are declared as being `virtual`. Finally, the `file_object` class is declared as follows:

```
class file_object: public Storage_Object {
  public:
    // Resource methods
    int  allocate   (int n);
    int  deallocate (int id, int n);

    // Storage_Object methods.
    void set_mode   (object_mode mode);
    int  read_data  (char *buf, int length);
    int  write_data (char *buf, int length);

    (...)
};
```

The `file_object` class must implement all of the unimplemented methods that it has inherited from its ancestors. It also chooses to override the default implementation of `set_mode`, replacing it with an enhanced version that manipulates the mode bits of the underlying file on disk, and then calls the original default implementation.

Having declared and implemented all the methods of the hierarchical interface, a simple test, applicable to any `Resource` might be implemented as follows:

```
void resource_test(Resource *r)
{
    for (;;) {
        // keep allocating forever
        r->allocate(1);
    }
}
```

This test attempts to exhaust the supply of a `Resource` in the hopes of stimulating anomalous behavior. Similarly, a more specific test, applicable only to `Storage_Object`s, might attempt to stimulate error conditions by writing unusually large segments of data:

```
void storage_obj_test(Storage_Object *s)
{
    // write out a big data block
    s->write_data(buf, 99*1024*1024);
}
```

Finally, the two tests can then be applied to objects of the appropriate types:

```
file_object f;
process_object p;

main()
{
    resource_test (&p);
    resource_test (&f);
    storage_obj_test (&f);
}
```

The resource test can be applied to both the `file_object` and the `process_object`, but the storage object test is only applicable to the `file_object`, because the `process_object` is not a `Storage_object`.

5.4.4. An example hierarchical test

As a simple illustration of a hierarchical benchmark, the hierarchical analog to part of the ASCM benchmark was constructed for a Unix system. Observe that the `read_file` and `write_file` tests described in Section 5.3 require only those capabilities that are defined by the class `Storage_Object`. The hierarchical Unix benchmark implements these as tests of the `Storage_Object` class, making use of the `read_data` and `write_data` methods respectively. The `file_object` and `memory_object` classes, which are defined in our hierarchical interface library as subclasses of `Storage_Object`, were evaluated using this benchmark.

Module-specific analysis	Test-specific analysis: correct	unexpected_error	unexpected_termination	bad_success
correct	correct	correct	correct	bad_success
unexpected_error	correct	unexpected_error	<impossible>	unexpected_error
unexpected_termination	correct	<impossible>	unexpected_termination	<impossible>
bad_success	bad_success	unexpected_error	<impossible>	bad_success

Table 8: A matrix showing how the module-specific and test-specific analyses of a test result are combined to determine the ultimate classification of that result. The categories `correct`, `unexpected_error` and `bad_success` correspond to the ASCM result classes with similar names, while the `unexpected_termination` class supersedes the ASCM `terminated` class. The derivation of this matrix is discussed at length in the text.

Implementation: Both the `read_data` and the `write_data` benchmarks are composed of two parts. The first part generates the objects (i.e. `file_objects` or `memory_objects`) which are used in the testing. This part embodies module-specific knowledge, as it must know at least the data type of the object to be generated, and may need to know the internal details of the object being generated. For example, it may need to know that an object encapsulating a *closed* file is required. The second part of the benchmark embodies the abstracted test routine itself. This part of the test is usually written at the highest possible level of abstraction, and does not embody any module-specific knowledge. For example, the `read_data` and `write_data` tests are written to apply to any `Storage_Object`, and do not require further knowledge of the object they are testing. Thus, the code for testing a `file_object` looks like:

```
for i = 1 to num_test_objects {
        Storage_Object obj;
        obj = file_test_lib::get_test_obj(i);

        test_storage_object(obj);
}
```

The routine `test_storage_object` can be applied to any storage object, such as a file or a piece of virtual memory. In this particular case, it conducts tests of reading and writing using all possible combinations of test buffer addresses and I/O request sizes. On the other hand, `get_test_obj()` is a module-dependent routine, implemented by each of the modules tested, that makes use of local module-specific knowledge to generate the objects to be tested. In the example shown, it returns a `file_object`, which is passed to `test_storage_object` and treated like any other `Storage_Object`.

Result processing: The hierarchical benchmark implemented maintains the six result classifications defined by the ASCM modular benchmark with a seemingly slight, but important, change. The ASCM error class `terminated` has been replaced by an `unexpected_termination` class, and terminations that are the expected outcome of a test are instead counted as `correct`. The reason for this change is discussed later.

The structure of the result processing code resembles that of the testing code in that test results are also processed by a module-dependent and a module-independent routine. A module-specific analysis routine is supplied by each module being tested, and evaluates the outcome of the test in view of the characteristics of the particular object that was tested. In a system without a system-wide integrated error handler, the module-specific routine is also able to examine module-specific error return mechanisms and incorporate their feedback into its analysis. Unlike the module-specific routine, the module-independent routine, also known as the test-specific routine, has no detailed knowledge of the object being tested, but has detailed knowledge only about the test that was applied. It is aware of the tested object only at the same level of abstraction as the test itself was. The test-specific routine is applicable across all modules that are tested. It evaluates outcomes based on system-wide error handling information, and on the observed behavior of the tested object. Note that some information may be available to both result-processing routines, and should be used by both of them in analyzing the outcome of the test.

Given the goal of classifying test results into one of the six classes described above, the two analysis routines each return `correct`, `unexpected_error`, `unexpected_termination`, or `bad_success`. These evaluations correspond to the first four possible result classifications respectively. The module-specific and test-specific evaluations are then combined to yield the final classification of the outcome, which falls into one of these four classes. In situations where either of the remaining two possible outcomes (`warm_restart` or `cold_restart`) occur, the occurrence is detected by an external monitoring agent (such as a human). In these two cases, the outcome is indisputable, and no further module-specific or test-specific analysis need be performed.

Consider, for example, a `write_data` test which attempts to write 1024 bytes of data to a `Storage_Object` that encapsulates a closed file, passing in a buffer that points to only 256 bytes of data. After the test has run, it is the job of the module-specific routine, which is aware of the internal details of the `Storage_Object`, to check whether the result indicates an invalid attempt to write to a closed file, as the module-specific routine is aware that the file object being written was actually closed. If the error code returned agrees with this predicted outcome, the module-specific routine indicates a `correct` return. Otherwise, it indicates `bad_success` or `unexpected_error`, as appropriate. The module-independent routine, on the other hand, is aware only of dealing with a `Storage_Object`, but knows that the test attempted to write more data than was actually supplied to the `write_data` call. It therefore checks to see how much data the test claimed to have written successfully, and once again classifies the outcome as being in one of the four aforementioned categories. The two evaluations are then combined according to the following principles, which are employed to produce the matrix of Table 8:

- If a call *succeeds* (i.e. didn't return an error code), *both* result analyses must agree that success was the expected outcome in order for the result to be deemed `correct`. This is because an error stimulus can be introduced into the test by either the module-specific object generation, or by the abstract test code initialization, without the knowledge of the other. Thus, if either *one* of the analyses expects an error as the correct outcome, the correct outcome must be an error, e.g., if `write_data` is called on a file that is open for writing, but is passed a NULL buffer and asked to write 1024 bytes, only the test-specific routine will predict an error as being the correct outcome. (This is because the decision to use a NULL buffer would have been made by the test-specific initialization code.) If executing this test does not trigger an error as predicted, the outcome is correctly classified as a `bad_success`.

- Conversely, if a call *fails* with an error, only *one* of the analyses need accept this error as the correct outcome in order for the result to be deemed `correct`. Once again, this is because it is possible (and likely) that only one of the two parts of the testing code is aware of a stimulus that is expected to cause an error condition. In the example above, suppose the call failed, and that the return code indicated that invalid data had been passed to the `write_data` call. Only the module-independent routine would have sufficient information to classify this as being the `correct` outcome. To the module-dependent independent routine, knowing only that a valid storage object was being written to, the error return would look like an `unexpected_error` return.

- If a call produces a *termination*, only one result analysis need accept this termination as being `correct` (i.e. expected) in order for the outcome to be considered `correct`. The justification for this case is analogous to that of handling the failure of a call.
- The occurrence of an `unexpected_termination` precludes the possibility of an `unexpected_error` or a `bad_success`. The analysis routines may not disagree on whether or not the benchmark was terminated.
- The analysis routines *may* disagree on whether a call failed or not. This may seem counter-intuitive, and should not be a common occurrence. However, it may occur in the presence of error return channels[5] that are not available to both analyses for examination. For example, a module-specific error-return channel might indicate to only the module-specific routine that a test had failed. Alternatively, the test-specific routine might conclude that a test had failed by observing behavioral data not available to the module-specific routine. In cases of such disagreement, one analysis may return a `bad_success` while the other returns `unexpected_error`. The correct combination of these two results yields `unexpected_error` because the analysis that detected an error had error indication channels available to it that were not available to the routine that did not indicate an error.

Note that the analysis routines in this implementation observe and analyze test termination by the operating system. This is done in order to classify each termination as having been `correct` or having been an `unexpected_termination`. Had the original ASCM classification been retained, classifying terminations in a group unto themselves, benchmark termination would not have required further analysis; terminations could simply have been monitored and handled in the same way as `cold restarts` or `warm restarts`. However, the implementation of the hierarchical benchmark exposed a problem with the ASCM result classification that renders this simpler scheme unusable.

The hierarchical benchmark was initially implemented and run on a `file_object` module interfacing to part of a Mach 3.0 filesystem. With minimal effort (restricted to extending the hierarchical OS interface library), the same test was later run on a `memory_object` module, interfacing to the virtual memory system. The results of running the benchmark are shown in Table 9.

These results serve to demonstrate one of the valuable benefits of hierarchical testing, namely that of enforcing *consistency* across modules in result processing, which is especially valuable when a benchmark is extended. The hierarchical benchmark had initially been implemented using the ASCM result classification scheme, which works reasonably well in the context of file management, because process termination is not the expected outcome of any file operation. However, process termination is often the *correct* means of signalling an error in interactions with the virtual memory

[5] An error return channel is any means by which the operating system can indicate an error condition to an application. Examples include return codes, global status flags, signals and system traps.

Operation Tested	Number of Tests	Result Class					
		`correct`	`unexpected error`	`bad success`	`unexpected termination`	`warm restart`	`cold restart`
read_data (`file_object`)	210	108 (0)	0	57	45	0	0
write_data (`file_object`)	210	92 (0)	9	91	18	0	0
read_data (`memory_object`)	210	115 (99)	8	51	36	0	0
write_data (`memory_object`)	210	101 (93)	8	50	51	0	0

Table 9: Evaluation of the `file_object` and `memory_object` modules using the hierarchical benchmark. The result classes used are those defined by the ASCM benchmark described in Section 5.3, modified so that the `terminated` class has been replaced with a class containing only *unexpected* terminations. Terminations which were the expected outcome of a test are counted as being `correct`. The number of `correct` outcomes that were due to expected terminations is included in parentheses in the table, and represents the number of results that would have been misclassified by the ASCM scheme.

system (e.g., as in the case of an attempt to reference an invalid address). When the benchmark was extended to test `memory_objects` in addition to `file_objects`, the limitations of the ASCM scheme became apparent; many otherwise `correct` outcomes were simply classified as `terminated`. As can be seen from Table 9, the large numbers of test outcomes that are misclassified by the ASCM scheme might lead to a conclusion that the virtual memory system was of very low quality. (Due to the misclassifications, less than 10% of the tests would have appeared to have produced the correct outcome.) However, further analysis revealed that the result classification system, and not the virtual memory system, was at fault. This led to the modified result classifications, wherein terminations are classed as either `correct` or `unexpected_termination`. The original shortcoming of the result analysis was made obvious by the hierarchical structure of the benchmark, which enforced comparability of the results across the two modules. A more *ad hoc* modular approach to the problem might easily have obscured it.

In addition, it should be noted that this case study exemplifies one of the potential pitfalls of robustness benchmark design — it is difficult to define error classes *without* knowing the outcomes of all possible tests to which those classes may be applied. The ASCM benchmark, for example, defined the `terminated` class in the absence of enough data to evaluate its suitability. The ASCM evaluations of the file module (see Table 7) show *no* occurrences of `terminated` tests at all, and only a few occurrences are reported for other modules that were similarly tested. In such cases, hierarchical testing serves, by enforcing consistency in result processing, to maintain a "placeholder" for result classes that are not well characterized. When more data on the vague classification become available, the hierarchical framework serves as a guide towards correct characterization of this new result class.

6. Conclusions and Future Work

In this paper, current robustness benchmark efforts have been examined. These approaches all fail to address certain issues that are critical to the long-term success of a robustness test suite. Several such issues have been delineated, and a proposed benchmarking organization that overcomes many of these problems has been outlined. The proposed hierarchical benchmarking organization does not adversely affect the desirable properties that have been attained by benchmarks to date. In particular, the proposal imposes a hierarchical, extensible structure upon test suites; this structure may mandate a higher initial implementation effort, but promises to improve the lifespan and maintainability of any robustness benchmark that is in use for a long period of time.

There remain several opportunities for improvement that have not been explored in the discussion of an extensible hierarchy. For example, hierarchically structured testing facilitates the combination of individual tests results into an overall "index of robustness". It also suggests a system for determining the relative importance of each of the individual results. In addition, the possibility of extending the hierarchical structure to include benchmarks that operate via fault injection has not been explored, mainly because simple fault injection violates the notion that testing procedures should be abstracted from the implementation details whenever possible. The possibility of reconciling this difference (via an abstracted form of fault injection) should be explored.

Much of the work presented herein has been carried out in the context of measuring the robustness of a Unix-like operating system. The kernel of the operating system has been regarded as an opaque, monolithic entity. However, the advent of micro-kernel based operating systems presents an opportunity for the lowest levels of such a hierarchical benchmark to focus on the robustness of the layers beneath the operating system server — i.e. on the micro-kernel itself. Hierarchically structured benchmarks can easily be extended to incorporate the notion of benchmarking one level deeper (simply by adding one or more layers to the bottom of the benchmark hierarchy).

Finally, this approach to measuring robustness can be applied easily to any large software system that was written in a modular manner. Such application promises the same extensibility, low maintenance cost, and consistency that is obtained in a similar benchmark of operating system robustness. The detailed investigation of such applications remains to be pursued.

References

[Barton90] J. H. Barton, E. W. Czeck, Z. Z. Segall, D. P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Transactions on Computers,* Volume 39, Number 4, April 1990, pp. 575-582.

[Cristian91] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM,* Volume 34, Number 2, February 1991, pp. 56-78.

[Curnow76] H. J. Curnow, B. A. Wichmann, "A Synthetic Benchmark," *The Computer Journal,* Volume 19, Number 1, 1976, pp. 43-49.

[Dingman93] C. Dingman, D. Siewiorek, "Measuring Robustness of a Fault Tolerant Aerospace System," *unpublished.*

[Kanawati92] G. A. Kanawati, N. A. Kanawati, J. A. Abraham, "FERRARI: a tool for the validation of system dependability properties," *The 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems,* Amherst, MA, USA, July 1992.

[Kao93] W.-I. Kao, R. K. Iyer, D. Tang, "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Transactions on Software Engineering,* Volume 19, Number 11, November 1993, pp. 1105-18.

[Miller90] B. P. Miller, L. Fredriksen, B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM,* Volume 33, Number 12, December 1990, pp. 32-43.

[Russinovich92] M. E. Russinovich, Z. Segall, "Open System Fault Management: Fault Tolerant MACH," *Research Report # CMUCDS-92-8, CMU Research Center for Dependable Systems,* Carnegie Mellon University, Pittsburgh, PA 15213-3890.

[SPEC90] Standard Performance Evaluation Corporation, *SPEC Newsletter,* Volume 2, Issue 2, Spring 1990, Waterside Assoc, Freemont, CA.

[Suh93] B.-H. Suh, J. Hudak, D. Siewiorek, Z. Segall, "Development of a Benchmark to Measure System Robustness," *Fault Tolerant Computing Systems: Twenty-Third International Symposium,* Toulouse, Frane, June 1993.

[Sullivan91] M. Sullivan, R. Chillarege, "Software Defects and their Impact on System Availability — A Study of Field Failures in Operating Systems," *Fault Tolerant Computing Systems: Twenty-First International Symposium,* Montreal, Que., Canada, June 1991.

[Weicker84] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM,* Volume 27, Number 10, October 1984, pp. 1013-1030

[Weiderman90] N. Weiderman, "Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications," *Ada Letters,* Volume 10, Number 3, Winter 1990, pp. 126-36.