CARNEGIE MELLON UNIVERSITY

HIGH PERFORMANCE ROBUST COMPUTER SYSTEMS

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

in

ELECTRICAL AND COMPUTER ENGINEERING

by

John Peter DeVale

Pittsburgh, Pennsylvania
October, 2001

Copyright 2001 by John DeVale. All rights reserved.

To my wife, Kobey, who believed in me,
and helped me believe in myself

Abstract

Although our society increasingly relies on computing systems for smooth, efficient operation; computer “errors” that interrupt our lives are commonplace. Better error and exception handling seems to be correlated with more reliable software systems[shelton00][koopman99]. Unfortunately, robust handling of exceptional conditions is a rarity in modern software systems, and there are no signs that the situation is improving. This dissertation examines the central issues surrounding the reasons why software systems are, in general, not robust, and presents methods of resolving each issue.

Although it is commonly held that building robust code is too impractical, we present methods of addressing common robustness failures in a simple, generic fashion. We develop uncomplicated checking mechanisms that can be used to detect and handle exceptional conditions before they can affect process or system state (preemptive detection). This gives a software system the information it needs to gracefully recover from the exceptional condition without the need for task restarts.

The perception that computing systems can be either robust or fast (but not both) is a myth perpetuated by not only a dearth of quantitative data, but also an abundance of conventional wisdom whose truth is rooted in an era before modern superscalar processors. The advanced microarchitectural features of such processors are the key to building and understanding systems that are both fast and robust. This research provides an objective, quantitative analysis of the performance cost associated with making a software system highly robust. It develops methods by which the systems studied can be made robust for less than 5% performance overhead for nearly every case, and often much less.

Studies indicate that most programmers have an incomplete understanding of how to build software systems with robust exception handling, or even the importance of good design with respect to handling errors and exceptional conditions[maxion98]. Those studies, while large in scope and thorough in analysis, contain data from students with little professional programming experience. This work presents data collected from professional programming teams that measured their expected exception handling performance against their achieved performance.

The data provides an indication that despite industry experience or specifications mandating robustness, some teams could not predict the robustness response of their software, and did not build robust systems.

Acknowledgments

First and foremost, I wish to thank my wife Kobey, who endured more than anyone should have to while I struggled through school. You are my light on earth, and I can not thank you enough for believing in me in the face of all adversity.

I would also like to thank my family who helped shape the person I am now. You taught me self reliance, trust, and the importance of personal integrity.

I would like to thank my committee, who gave freely of their time to mentor and guide me through this process. Especially Phil Koopman, my advisor, who taught me how to think critically about problems, perform effective research, and most importantly, to communicate results effectively.

I want to thank everyone at Carnegie Mellon who made this work possible and helped make it a fun place to be: Bryan Black for putting everything in perspective and making pizza with me every Tuesday. By the way, thanks for the house. Cindy Black for not getting too mad at Bryan and I when we took over her kitchen every Tuesday to make pizza, and for laughing at our outrageous behavior. Lemonte Green for being a true friend and setting the best of examples. My good friend Kymie Tan, the perfectionist, for appreciating my dark humor. Bill and Melanie Nace, who trooped off to renaissance fairs with us, and shared their lives with us. Akiko Elaine Nace for running around the office with the exuberance only a four year old can have, making me pictures, and giving everyone a reason to smile. Lou and Jim, our network engineers who kept everything running smoothly despite the dumb things we did to the system. Karen, Melissa, Lynn, Elaine and Elaine (Yes, two Elaines) for doing their jobs so well and with such dedication that we never had to worry about anything except working on our projects.

Thank you all.

Table of Contents

Figure list

Table List

1 Introduction

As our society becomes more dependant on the complex interactions among electronic systems, the ability of these systems to tolerate defects, errors, and exceptions is critical to achieving service goals. Every aspect of life is becoming dependant on computers, and the software that runs on them. From banking to traffic control, weapons systems to a trip to the grocery store, the things we take for granted are now irrevocably tied to the correct functionality of computing systems.

Of course the need for fault tolerance is not an entirely new thing. Military, aerospace, medical, and financial systems have always been built to be as tolerant of faults as practical. Though they have not always been as robust as their designers may have hoped [jones96] [leveson93] [lions96], the effort to build robust systems was made. Further, the designers of fault tolerant systems continued their efforts with the goal of improving subsequent system generations.

Though fault tolerance issues are addressed in safety critical computing sectors (i.e. medical, aerospace), other sectors lack the time, money and expertise to build robust systems using the methods developed in the traditional safety critical sectors. Further, as the promise of reusable, modular, computational components (beans, OLE containers, etc.) becomes closer to reality, building systems that can tolerate faults in reused/purchased components becomes critical.

1.1 Motivation

The research in this thesis tells a complex story, and is one whose motivation is at times subtle. This section is an attempt to relate the path of experiences and observations (in roughly chronological order) that led to the inception of this work, and guided its direction. Readers who are more interested in the results and technical details are invited to skip to the section immediately after this one, **Terms and Definitions** (1.2).

The historical basis for this work is deeply rooted in the fault injection community within the larger fault tolerance community. Fault tolerance has been a goal of engineers since engineering

was invented as a discipline. After all, no one is happy if the first stiff wind to come along blows their house over. This philosophy is applied today to a wide variety of computing systems, including banking, aerospace, military, and medical, just to name a few.

One of the drawbacks of fault injection [kanawati92], has been that it usually required special purpose hardware (or was specific to a hardware setup). It was difficult to trace the root cause of problems it detected, and it could be difficult to reproduce due to the random element involved. Of course, during the height of its popularity, this approach worked well because target systems were relatively simple, and were designed specifically for a purpose requiring fault tolerance.

Another potential problem is the lack of a consistent, meaningful, numerical result that could relate what was measured by the existing tools. It can be unclear exactly what fault injection tools measure, and how to relate that measurement back to the system.

Dissatisfaction with some of the limitations of traditional fault injects led to a line of research at Carnegie Mellon University that culminated in the Ballista tool. Ballista is a fault injection system that is software based, and provides portable, repeatable fault injection at the API level with meaningful numerical results that can be compared and related back to the system under test. The initial version of this tool was very primitive, with test data types that were hard-coded into the system, and little flexibility as to what type of API's it would test.

Armed with our new tool, we set out to change the world (or at least our little corner of it) by testing several POSIX compliant operating systems. The tool was applied to every POSIX compliant OS we could get our hands on. Although our instincts told us the systems would not do well in the tests, even we were surprised and a bit appalled by what results we obtained.

Although we did publish our complete results, we gave the results to the OS vendors well before publication to allow them to fix the problems and discuss them with us. In a few cases, the vendors elected to send us new versions of the operating system for retest. In each of these cases however, the newer system had a different robustness, and not necessarily better.

We did succeed in our goal of interacting with the developers, and their responses were varied and ranged from "We will fix everything we can"(IBM), "Dumping core is the right thing"(FreeBSD), "We understand, but are not interested in addressing these issues"(Linux), to

"You guys are crazy" (name withheld to protect the guilty). Some companies, like AT&T, Cisco, and military contractors agreed that these results were important. But by and large the prevailing sentiment was that fixing these issues just couldn't be done. It was just too hard to do, it would cost too much in terms of performance, and no one can do any better.

Confronted by what appeared to be the cold hard reality of the situation, we stopped and took stock of our position. Although nearly everyone we talked to disagreed, we felt that there had to be a way to address these issues in a reasonable fashion. Unfortunately, we didn't have any evidence to support our position.

We took some time to enhance our tool, and completely rebuilt Ballista, making it extensible to any API. A custom language and compiler were created to allow easy creation of custom data types for testing. This gave us the ability to test a wide range of software systems.

Having completely rewritten the tool, we began the process of deciding the next best step in addressing the problem of improved system robustness. As a simple expedient, we elected to investigate the FreeBSD math libraries. We wanted to determine how difficult it would be to address the existing robustness problems, and what the cost of fixing them was.

We were able to fix all the robustness failures in the math library, and it turned out to be not very costly in terms of performance(<1%). Additionally, we discovered that there was already a lot of checks in libm similar to the ones we needed to add for improved robustness. These tested ranges and boundary conditions that were for the most part necessary to make the calculations correct, but also checked for some error conditions, which forced a signal if the condition were detected. As it turned out, the stock libm implementation already paid most of the cost of making the library robust, just to get the calculation correct. For just a few percent more we were able to make it robust, and afford to check return codes after each call.

Of course, the response to this result was predictably negative. The math libraries represent simple, computational code. They don't use any complex data structures, and don't affect machine state. Thus the results were generally seen as providing little solid evidence supporting our idea that it was possible build robust software, and you could do so with little performance cost as well.

Now that we had at least one example however, the criticisms changed a bit to include a new idea. The idea was that developers *could* build robust code, but didn't because of complexity and speed issues. Thus indicating that there were no insights to be gained from even using our tool for robustness testing. The issues we uncovered were believed to be understood, but simply ignored.

To address the new direction our critics were taking, we decided look at a package written with software reliability and fault tolerance in mind: the Safe, Fast, I/O library. The authors of SFIO were able to create and I/O library that avoided some of STDIO's inefficiencies, and added in a variety of safety features that greatly improved its robustness over that of STDIO. It did suffer failures (despite comments by the authors indicating the contrary) – more than we suspected it should.

We took our results to the SFIO authors, and they firmly believed that the failures remaining were too hard, and too performance costly to remove. They pointed out that they had improved the robustness to a large extent, and had removed all the failures they felt they could without sacrificing performance.

This explanation seemed plausible, so we elected to look to see what kinds of performance vs robustness tradeoffs it was possible to make. To facilitate this we obtained a copy of the SFIO source code, and waded in trying to fix the problems Ballista found.

After a couple of weeks we were able to build in additional checks to make the code more robust. So robust, in fact, that many functions exhibited no failures, or only fractional percentages of failures. The only thing that remained was to benchmark the new code, and figure out what performance tradeoffs were possible.

We recreated the original benchmarks used by the authors of SFIO (scaled for today's architectures), and found that through a few simple optimizations, the performance penalty was not large(<2%). Despite what was originally thought, the SFIO didn't fix everything, and even the failures it didn't address could be fixed without detracting significantly from the performance of the library.

Finally, we looked at several speed critical OS services under Linux including memory and process synchronization primitives. We were able to completely harden even these low level service modules for only a slight performance penalty(<5% for the pessimistic case). Thus what began as an attempt to quantify the performance tradeoffs inherent with building robust systems became an work detailing simple, generically applicable methods one might use to build high performance robust computing systems.

The last piece to this puzzle was the nagging issue of the purpose of the tool itself, Ballista. If, as some would have us believe, developers understand how to build robust software, and were not simply because of speed and complexity issues, then the tool is redundant. If that is not the case, however, then the tool is critical, because it offers feedback as to how well robustness goals are being met.

A great deal of thoughtful experimentation had already been done in this area by Roy Maxion, but the last piece fell into place during the course of a joint research project with IBM's Center for Software Engineering Research. Peter and his group were interested in coming up with a methodology to describe the non-functional qualities of components in a way that encourages reuse. One aspect of this is robustness. Using Maxion's work as a springboard, we came up with a system whereby developers within IBM could classify the robustness of a software system.

An experiment was then devised whereby developers could classify their software modules and indicate how robust they expected them to be. The modules could then be tested, and the actual versus expected robustness values compared. This experiment was executed by the research team at the IBM Center for Software Engineering Research, and the data and components were turned over to us for testing and comparison.

Once we had tested the components and related their measured robustness to their actual robustness, we noticed that there were a number of discrepancies between the expected and measured robustness. We attribute this to the phenomena discussed by Maxion in [maxion98]. It thus seems likely that for any team to build robust software some tool such as Ballista is necessary to perform tests to determine how well robustness goals are being met, and feed back into the software process.

Although the path of research seemed long and serpentine during its execution, it is in retrospect fairly straight and logical. We started with a metric. We addressed each concern we heard detailing reasons why robust software systems were not being developed. Finally, we relearned that even though people are smart, you still need a some method or tool to check to make sure what we think is happening is actually happening.

At the end of the day it is the software groups themselves who determine if they will build robust systems or not – but now there are no more technical excuses.

1.2 Terms and Definitions

For the purposes of this thesis, we use the definition of *robustness* as that given in the IEEE software engineering glossary, dropping the environmental conditions clause: "The degree to which a system or component can function correctly in the presence of invalid inputs[IEEE90]."

We quantify robustness in terms of *failure rate.* The *failure rate* for a function represents the percentage of test cases that cause robustness failures to occur.

We define the normalized failure rate for a particular operating system to be:

$$F = \sum_{i=1}^{N} w_i \times \frac{f_i}{t_i}$$

with a range of values from 0 to 1 inclusive, where: $\sum w_i = 1$

N = number of functions tested - Equal to the number of calls a particular OS supports of the 233 in our suite
w_i is a weighting of importance or relative execution frequency of that function where
f_i is the number of tests which produced robustness failure for function i
t_i is the number of tests executed for function i

We define *hardening* as the act of adding software to perform the following tasks:

- Perform run-time checks and validation to determine the validity of operand data
- Detect and handle any exceptional conditions.

Thus a software package that has been *hardened* is by definition *robust.*

1.3 Exception Handling

Anecdotal data collected by robustness testing seems to suggest that systems incapable of gracefully handling exceptional conditions (including exceptions caused by software defects in application programs calling other software packages) tend to be somewhat less reliable at a

system level, and much less reliable at the task level. While the evidence does not prove causality, in many cases overall system failures tend to be within modules with poor overall exception handling characteristics [christian95][shelton00][koopman99].

Despite a general need for better exception handling and the existence of tools to identify exception handling shortcomings, few projects pay anything other than passing attention to this aspect of the system. Some developers simply lack exposure to the need and methods for exception handling [maxion98]. Others eschew it because of perceived performance problems and development difficulty. Neither of these need be the case. As Maxion points out, even a small amount of effort applied to raising the awareness of the importance of solid exception handling can result in significant improvements [maxion98]. Additionally, there are now several research and commercial tools to help developers detect potential robustness weaknesses [devale99][hastings92][carreira98][ghosh99]. But, beyond the issue of finding and correcting robustness problems, our experience is that developers greatly overestimate the performance penalty of making software highly robust and use this as a reason to avoid robustness improvement.

1.4 Language based approaches

The idea of building good exception handling is not a novel one. It is a fundamental part of some languages that have well defined constructs for handling exceptional conditions. One of the more popular such languages today is Java.

Java provides optional mechanisms that enforce exception handling constructs for modules. It allows developers to specify which exceptions a given method can throw, amongst predefined or user defined types. The compiler then forces any calling routine to catch and handle every type of exception the module is defined to throw. Similarly, C++, Ada, ML, and other languages all provide various levels of compiler enforceable exception handling.

One problem with these approaches is incomplete foresight on the part of the people writing the software modules. It is nearly impossible to anticipate every possible event or condition, and often there are large gaps where no or improper exception handling exists.

It is conceivable that programmers can grow to rely too heavily on a language that is “safe” and become complacent. As we will see later in this work, some Java components have many robustness failures despite it being a “safe” language.

```
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>
jmp_buf EC;
Xsegfault_error(int signum) {
    signal(SIGSEGV,(void *)Xsegfault_error);
    longjmp(EC,1);
}

int main() {
    int i=0;
    {
        int exc_code;
        signal(SIGSEGV,(void *)Xsegfault_error);
        if ((exc_code = setjmp (EC)) == 0) {
            i=10;
            i=fileno(((void *)0));
        } else {
            i--;
            printf("Exception handled %i\n"
                  ,exc_code);
        }
    }
    printf("Complete %i\n",i);
    return(0);
```

Figure 1. Example of C signal exception handling

Also of great importance is the question of *recoverability* after a language exception is received. Although any of the languages with built in exception handling will deliver language exceptions instead of low level signals, they do little to ensure that the exceptional condition can be safely recovered from without a task restart.

1.5 Operating System Trap Support

Many of the languages with built in routines for exception handling rely on specific support from a virtual machine, as in the case of Java, or operating system support for signal handling. Signals caused by exceptional conditions (or software generated conditions) are interpreted by a small runtime environment and converted into the appropriate language constructs. These constructs are then handled by the built in mechanisms.

Other languages without any specific built in support for exception handling may use the same mechanisms to trap and handle signals (machine/software generated exceptions). In C, for example , the programmer may set up exception handling mechanisms by using the signal handling defined by POSIX (Figure 1. Example of C signal exception handling). In essence, the

programmer can set up a large number of small, localized exception handling routines to be called upon the occurrence of an exception (in essence modeling the try-catch framework), or setup large, global handlers.

1.6 Performance Issues

Although exception handling is generally perceived to be expensive in terms of performance, there are some software systems that attempt to be robust without sacrificing performance. An example of a software package developed with safety and robustness as a goal, without compromise to performance, is the safe, fast, I/O library (SFIO) developed by David Korn and K.-Phong Vo at AT&T research [korn91]. The functions included in SFIO implement the functionality of the standard C I/O libraries found in STDIO. This library adds a large amount of error checking ability (as well as other functionality) to the standard I/O libraries, and manages to do so without adversely affecting performance.

While the authors of SFIO were able to demonstrate that it was a high performance library, at the time it was developed there was no method for quantifying robustness. They could make a case that the library was safer due to their design decisions, but there was no method available to quantify how much they had improved over STDIO. Furthermore, discussions with the developers of SFIO revealed that even they were concerned about the performance impact of increasing the amount of exception checking done by their code.

The existence of SFIO can be viewed as an opportunity to gain an understanding of how robust an Application Programing Interface (API) implementation might be made using good design techniques but no metrics for feedback, and what the actual performance penalty might be for further improving robustness beyond the point judged practical by SFIO developers. If the performance cost of adding solid exception handling and error checking is as high as conventional wisdom holds, then how can we reduce that cost? Conversely, if the cost is not prohibitive, what is it, and how good can the error checking and exception handling be while still keeping the cost within reason?

1.7 Knowledge of Code

Beyond the cost of exception handling, it is important to understand the ability of developers to estimate how well their own code performs with respect to its graceful response to exceptional conditions. Once this is done, it will aid in determining the reason some software is not as complete in this regard as it might be, and perhaps put us on the path toward being able to develop software with good exception handling in a wider range of projects - not just military aerospace applications.

Available evidence suggests inexperienced developers are not able to write code with robust error and exception handling[maxion98]. This might support the hypothesis that professional developers would have a similar deficiency. However, without even cursory data to explore this notion, any effort to understand and improve the ability of professional developers to successfully write software systems with robust exception handling is just speculation.

1.8 Outline

This dissertation begins with an overview of previous work in the general area of exception handling in Chapter 2, which discusses the broad nature of existing work, and how this dissertation fits in to the overall spectrum of exception handling research. Chapter 3 presents the Ballista software robustness testing tool in detail. Chapter 4 presents experimental results on the robustness of POSIX operating systems that shaped the core idea that, in general, robustness is poorly understood, and is not a priority for many professional software developers. Vendor response to these results led to the further investigation presented here as to the tractability and performance characteristics of robust exception handling. Chapter 5 presents experimental data and analysis of the standard math libraries distributed with FreeBSD. This represents the earliest exploratory work done in this investigation. Chapter 6 presents the results from the analysis and hardening of the Safe, Fast I/O library (SFIO)[korn91]. The results from applying techniques and lessons developed while hardening SFIO and the math libraries to operating system services can be found in Chapter 7. Data collected from professional development groups comparing their expected robustness with measured robustness is presented in Chapter 8. This dissertation concludes in Chapter 9 with a discussion of contributions and possible future work.

2 Prior Work

The literature written on exceptions and exception handling is vast. Exception handling has been studied since the incept of computing, and is important for detecting and dealing with not only problems and errors, but also expected conditions. The research falls largely into three major categories with respect to exception handling:

- How to describe it
- How to perform it
- How to do it fast

2.1 Describing Exception Handling

Exception handling code can be difficult to represent in terms of design and documentation, largely because it generally falls outside normal program flow, and can occur at virtually any point in a program. Accordingly, a large body of work has been created to help develop better methods to describe, design and document the exception handling facilities of a software system.

Early work strove to discover multiple ways to handle exceptional conditions [hill71] [goodenough75]. Over the years two methods have come to dominate current implementations. These methods are the termination model and the resumption model [gehani92].

In current systems the two main exception handling models manifest themselves as error return codes and signals. It has been argued that the termination model is superior to the resumption model [cristian95]. Indeed, the implementation of resumption model semantics in POSIX operating systems (signals), provides only large-grain control of signal handling, typically at the task level resulting in the termination of the process (e.g. SIGSEGV). This can make it difficult to diagnose and recover from a problem, and is a concern in real-time systems that cannot afford large-scale disruptions in program execution.

Implementations of the termination model typically require a software module to return an error code (or set an error flag variable such as `errno` in POSIX) in the event of an exceptional condition. For instance a function that includes a division operation might return a divide by zero

error code if the divisor were zero. The calling program could then determine that an exception occurred, what it was, and perhaps determine how to recover from it. POSIX standardizes ways to use error codes, and thus provides portable support for the error return model in building robust systems [IEEE94].

At a higher level of abstraction, several formal frameworks for representing exception handling and recovery have been developed [edelweiss98][hofstede99]. These methods attempt to build an exception handling framework that is easy to use and understand around a transactional workflow system.

Highly hierarchical object oriented approaches seek to build flexible and easy to use frameworks that bridge the gap between representation and implementation [dony90]. Yet another approach is to use computational reflection [maes87] to separate the exception handling code from the normal computational code [garcia99][garcia00].

2.2 Performing Exception Handling

As we can see from the C language example in Figure 1, exception handling mechanisms can often make code generation and reading difficult. This is a problem throughout the development lifecycle. Easing the burden of developing, testing, and maintaining software with exception handling constructs through better code representation is important for not only reducing costs, but improving product quality. Consequently, there is a large field of related work.

One common way of easing the burden of writing effective exception handling code is through code and macro libraries. This type of approach has the benefit of being easily assimilated into existing projects, and allows developers to use traditional programming languages [lee83] [hull88] [hagen98] [buhr00]. More aggressive approaches go beyond simple compiler constructs build entire frameworks [govindarajan92] [romanovsky00] or language constructs [lippert00].

The focus of this research is more along the lines of identifying exceptional conditions before an exception is generated (in an efficient manner), rather than developing exception handling mechanisms that are easier to use. As such, the most closely related work is Xept [vo97]. The Xept method is a way in which error checking can be encapsulated in a wrapper, reducing

flow-of-control disruption and improving modularity. It uses a tool set to facilitate intercepting function calls to third party libraries to perform error checking. Xept is a somewhat automated version of the relatively common manual "wrapper" technique used in many high availability military systems.

Xept has influenced the research presented here, and the work leading up to it. The research presented here uses the idea of avoiding exception generation in order to harden a software interface against robustness failures. Further, it explores the practical limits of such hardening, in terms of detection capabilities and performance cost. In some cases, a tool such as Xept might work well as a mechanism for implementing checks discussed in our work. Unfortunately it does incur the overhead of at least one additional function call in addition to the tests performed per protected code segment. Though the Xept check functions can not be inlined due to the structure of its call-intercept methodology, it is not difficult to imagine practical modifications to the technology that would allow the inlining optimization.

2.3 High Performance Exception Handling

In today's high performance culture, the desire for fast exception handling is obvious. The work discussed in this section largely focuses on generating, propagating, and handling exceptions as fast as possible. This is complementary to the work presented in this document. Once generated, exceptions can be difficult to recover from in a robust fashion. This work is mainly interested in developing methods of including enhanced error detection in software systems to detect exceptional conditions to the maximum extent possible before they generate exceptions, and to do so without sacrificing performance.

Exception delivery cost can be substantial, especially in heavily layered operating systems where the exception needs to propagate through many subsystems to reach the handling program. In [thekkath94], the authors present a hardware/software solution that can reduce delivery cost up to an order of magnitude. In [zilles99], the use of multithreading is explored to handle hardware exceptions such as TLB misses without squashing the main instruction thread. The work presented here may benefit from multithreading technologies that are beginning to emerge in new commercial processor designs by allowing error checking threads to run in parallel. However,

synchronizing checking threads with the main execution thread may prove to be costly in terms of execution overhead, and certainly machine resources. This work performs checks in the main thread, building them such that the enhanced multiply branch prediction hardware[rakvic00] and block caches[black99] can simply execute checks in parallel and speculatively bypass them with little or no performance cost.

In [wilken93][wilken97] the authors propose a hardware architecture to allow the rapid validation of software and hardware memory accesses. Their proposed architecture imposed only a 2% speed penalty. Unfortunately, the authors also determined that without the special hardware, the scheme was too costly to implement in software alone. Other work proposes a code transformation technique to detect memory exceptions, resulting in performance overheads of 130%-540% [austin94].

This work expands on these ideas in some key areas. It creates a simple, generically applicable construct for exception detection. It quantifies the performance cost of using the construct to provide robust exception detection and handling, and it discusses ways in which emerging microprocessor technologies will improve the construct's performance even further.

3 The Ballista Robustness Benchmarking Service

Although computing systems continue to take on increasingly important roles in everyday life, the dependability of these systems with respect to software robustness may not be as high as one would like. The disruption of communications services or a business server can cause substantial problems for both the service provider and the consumer dependent upon the service. Personal computer users have become inured to crashes of popular desktop computer software. But, these same people are notoriously unsympathetic when faced with computer problems that disrupt services in their daily lives.

Software robustness failures and exceptional conditions are not a new problem. An early, highly visible software robustness failure in a critical system came in the Apollo Lunar Landing Program despite the ample resources used for engineering, development and test. In 1969 the Apollo 11 space flight experienced three mission- threatening computer crashes and reboots during powered descent to lunar landing, caused by exceptional radar configuration settings that resulted in the system running out of memory buffers [jones96].

More recent software development methodologies for critical systems (*e.g.*, [rational01]) have given engineers some ability to create robust systems. But, cost cutting, time pressure, and other real-world constraints on even critical systems can lead to less than perfectly robust software systems. One of the more spectacular recent instances of this was the maiden flight of the Ariane 5 heavy lift rocket. Shortly after liftoff the rocket and payload were lost due to a failure originating from an unhandled exceptional condition during a conversion from a floating point value to an integer value [lions96]. It stands to reason that everyday projects with lower perceived criticality and budget size are likely to be even more susceptible to robustness problems.

As illustrated by the space flight examples above and most people's daily experience with personal computer software, all too often the response to exceptional conditions is less than graceful. Frequently the focus of software development processes is either that a given input is exceptional and therefore should not have been encountered, or that the lack of a specification for that input was a defect in the requirements. But, from a user's perspective, a failure to handle an

exceptional condition gracefully amounts to a software failure even if it is not strictly speaking caused by a software defect (it is irrelevant to laypeople that software development documents leave responses to a particular exceptional condition "unspecified" if they have lost anything from an hour's work to a family member due to a software crash).

History and common sense tell us that specifications are unlikely to address every possible exceptional condition. Consequently, implementations developed by typical software development teams are probably not entirely robust. And, in the world of commercial software it is even more likely that resource and time constraints will leave gaps where even exceptional conditions that might have been anticipated will be overlooked or left unchecked.

As a simple example, consider an ASCII to integer conversion inadvertantly fed a null string pointer expressed by the C call `atoi(NULL)`. As one might expect, on most systems this call causes a segmentation fault and aborts the process. Of course one would not expect programmers to deliberately write "`atoi(NULL)`." However, it is possible that a pointer returned from a user input routine purchased as part of a component library could generate a NULL pointer value. That pointer could be passed to `atoi()` during some exceptional condition – perhaps not documented anywhere (just to pick an example, let's say that this happens if the user presses backspace and then carriage return, but that isn't in anyone's test set).

Should `atoi()` abort the program in this case? Should the application programmer check the input to `atoi()` even though there is no specified case from the input routine component that could generate a null pointer? Or should `atoi()` react gracefully and generate an error return code so that the application program can check to see if the conversion to integer was performed properly and take corrective action? While it is easy to say such a problem could be handled by a bug patch, a lack of robustness in even so simple a case could cause problems ranging from the expense of distributing the patch to embarrassment, loss of customers, or worse.

Resource and time constraints too often allow thorough testing of only the functional aspects of a software system. Functional testing is often the easiest testing expense to justify as well. After all, it is easy to conjure visions of unhappy customers and poor sales to even the most budget-conscious cost cutter if a product fails to perform advertised functions. However,

development methodologies, test methodologies, and software metrics typically give short shrift to the issue of robustness. In particular, there has previously been no comprehensive way to quantify robustness, so it is difficult to measure the effects of spending money to improve it. Furthermore, exceptional conditions are typically infrequent, so it is difficult to justify spending resources on dealing with them. (However, it is a mistake to think that *infrequent* or *exceptional* necessarily equates to *unlikely*, as can be demonstrated by year values rolling from 1999 to 2000 with two-digit year data fields; this is infrequent, but certain to happen.)

While robustness may not be an important issue in every software system, those systems which require software robustness can pose special difficulties during testing. In a "robust" software system it is typical for up to 70% of the code to exist for the purpose of dealing with exceptions and exceptional conditions[gehani92]. Unsurprisingly in light of this, other sources state that mature test suites may contain 4 to 5 times more test cases designed to test responses to invalid conditions and inputs ("Dirty" tests) than those designed to test functionality ("Clean" tests)[beizer95]. In short, writing robust code and testing for robustness are both likely to be difficult, time consuming, and expensive. Given that the bulk of development effort in such situations is spent on exceptions rather than "normal" functionality, it would seem useful to have tools to support or evaluate the effectiveness of exception handling code.

Other problems arise when attempting to use off-the-shelf software components. It may be difficult to evaluate the robustness of software without access to complete development process documentation (or in some cases even source code!). Yet if robustness matters for an application, it would seem that robustness evaluations would be useful in selecting and using a component library. Evaluations would be even more useful if they were performed in a way that permitted "apples-to-apples" comparisons across the same or similar component libraries from multiple vendors.

The Ballista approach to robustness testing discussed in this chapter provides an automated, scaleable testing framework that quantifies the robustness of software modules with respect to their response to exceptional input values. It generates specific test cases that deterministically reproduce individual robustness failures found during testing in order to help developers pin

down robustness problems. Additionally, it has been used to compare off-the-shelf software components by measuring the robustness of fifteen different implementations of the POSIX operating system Application Programming Interface (API).

3.1 Previous Work

While the Ballista robustness testing method described here is a form of software testing, its heritage traces back not only to the software testing community, but also to the fault tolerance community as a form of software-based fault injection. Ballista builds upon more than fifteen years of fault injection work at Carnegie Mellon University, including [schuette86], [czeck86], [barton90], [siewiorek93], [mukherjee97], [dingman95], and [dingman97], and makes the contribution of attaining scalability for cost-effective application to a reasonably large API.

An early method for automatically testing operating systems for robustness was the development of the Crashme program [carrette96]. Crashme operates by writing random data values to memory, then spawning large numbers of tasks that attempt to execute those random bytes as concurrent programs. While many tasks terminate almost immediately due to illegal instruction exceptions, on occasion a single task or a confluence of multiple tasks can cause an operating system to fail. The effectiveness of the Crashme approach relies upon serendipity – in other words, if run long enough Crashme may eventually get lucky and find some way to crash the system.

Similarly, the Fuzz project at the University of Wisconsin has used random noise (or "fuzz") injection to discover robustness problems in operating systems. That work documented the source of several problems [miller90], and then discovered that the problems were still present in operating systems several years later [miller98]. The Fuzz approach tested specific OS elements and interfaces (as opposed to the completely random approach of Crashme), although it still relied on random data injection.

Other work in the fault injection area has also tested limited aspects of robustness. The FIAT system [barton90] used probes placed by the programmer to alter the binary process image in memory during execution. The FERRARI system [kanawati92] was similar in intent to FIAT, but uses software traps in a manner similar to debugger break-points to permit emulation of

specific system-level hardware faults (*e.g.*, data address lines, condition codes). The FTAPE system [tsai95] injected faults into a system being exercised with a random workload generator by using a platform-specific device driver to inject the faults. While all of these systems produced interesting results, none was intended to quantify robustness on the scale of an entire OS API.

Earlier work at CMU attempted to attain scalable testing for large API [mukhergee97]. That work attempted to generically map a method's functionality into testing groups. While the approach worked, it did not fulfil its goal of being scalable because of the large effort required on a per function basis to specify the generic functionality of a software function.

While the hardware fault tolerance community has been investigating robustness mechanisms, the software engineering community has been working on ways to implement robust interfaces. These efforts are discussed in detail in Chapter 2. For the moment let us simply reiterate that the POSIX supported mechanism for creating portable, robust systems is the error return code model[IEEE94, lines 2368-2377].

Typical software testing approaches are only suitable for evaluating robustness when robustness is included as an explicit and detailed requirement reflected in specifications. When comprehensive exceptional condition tests appear in code traceable back to requirements, regular software engineering practices should suffice to ensure robust operation. However, software engineering techniques tend to not yield a way to measure robustness if there is no tracability to specifications. For example, test coverage metrics tend to measure whether code that exists is tested, but may provide no insight as to whether code to test for and handle non-specified exceptions may be missing entirely.

There are tools which test for robustness problems by instrumenting software and monitoring execution (*e.g.*, Purify [rational01], Insure++ [para01], and BoundsChecker [numega01]). While these tools test for robustness problems that are not necessarily part of the application software specification, they do so in the course of executing tests or user scripts. Thus, they are useful in finding exceptional conditions encountered in testing that might be missed otherwise, but still rely on traditional software testing (presumably traceable back to specifications and acceptance

criteria) and do not currently employ additional fault injection approaches. Additionally, they require access to source code, which is not necessarily available. In contrast, Ballista works by sending selected exceptional inputs directly into software modules at the module testing level, rather than by instrumenting existing tests of an integrated system. Thus it is complementary to, rather than a substitute for, current instrumentation approaches.

3.2 The Ballista Testing Methodology

The Ballista robustness testing methodology is based on combinational tests of valid and invalid parameter values for system calls and functions. In each *test case*, a single software Module under Test (or *MuT*) is called once to determine whether it provides robust exception handling when called with a particular set of parameter values. These parameter values, or *test values*, are drawn from a pool of normal and exceptional values based on the data type of each argument passed to the MuT. A test case therefore consists of the name of the MuT and a tuple of test values that are passed as parameters (*i.e.*, a test case would be a procedure call of the form: *MuT_name(test_value1, test_value2, ..., test_valueN)*). Thus, the general Ballista approach is to test the robustness of a single call to a MuT for a single tuple of test values, and then iterate this process for multiple test cases that each have different combinations of valid and invalid test values.

Thus, the general approach to Ballista testing is to test the robustness of a single call to a MuT for a single tuple of test values, and then repeat this process for multiple test cases that each have different combinations of valid and invalid test values. Although this style of testing excludes timing and sequence issues, the system could be extended to cover these areas. Phantom parameters or test scaffolding (discussed later in this chapter) provide the opportunity to investigate sequence or timing issues. Of course, the significant drawback to doing so is a potential loss of scalability.

3.2.1 Scalable testing without functional specifications

The Ballista testing framework achieves scalability by using two techniques to abstract away almost all of the functional specification of the MuT. The reasoning is that if the functional specification can be made irrelevant for the purposes of testing, then no effort needs to be made to

create such specifications (this is especially important for testing legacy code, code from third party software component vendors, or code with specifications that are not in machine-usable form).

The first technique to attain scalability is that the test specification used by Ballista is simply "doesn't crash; doesn't hang." This simple specification describes a broad category of robust behavior, applies to almost all modules, and can be checked by looking for an appropriate timeout on a watchdog timer and monitoring the status of a task for signs of abnormal termination. Thus, no separate functional specification is required for a module to be tested (those few modules that intentionally terminate abnormally or hang are not appropriate for testing with Ballista).

The second technique to attain scalability is that test cases are based not on the functionality of the MuT, but rather on values that tend to be exceptional for the data types used by the MuT. In other words, the types of the arguments to a module completely determine which test cases will be executed without regard to what that module does. This approach eliminates the need to construct test cases based on functionality. Additionally (and perhaps surprisingly), in full-scale testing it has proven possible to bury most or all test "scaffolding" code into test values associated with data types. This means that to a large degree there is no need to write any test

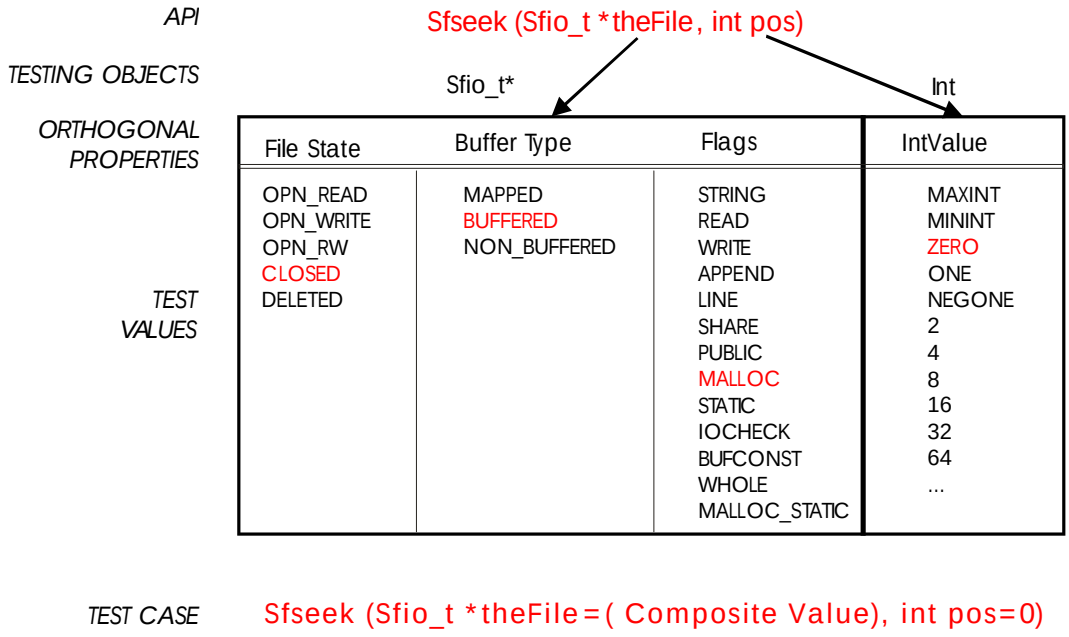


Figure 2. Ballista test parameter creation

scaffolding code, nor any test cases when testing a new module if the data types for that module have been used previously to test other modules. The net result is that the effort to test a large set of modules in an API tends to grow sublinearly with the number of modules tested.

For each function tested, an interface description is created with the function name and type information for each argument. The type information is simply the name of a type which Ballista is capable of testing. In some cases specific information about how the argument is used was exploited to result in better testing (for example, a file descriptor might be of type int, but has been implemented in Ballista as a specific file descriptor data type).

Ballista bases the test values on the data types of the parameters used in the function interface (Figure 2). For example, if the interface of a function specifies that it is passed an integer, Ballista builds test cases based on what it has been taught about exceptional integer values. In the case of integer, Ballista currently has 22 potentially exceptional test values. These include values such as zero, one, powers of two, and the maximum integer value.

Some complex data types have orthogonal properties which, if abstracted properly, can greatly reduce the complexity and effectiveness of test objects. Ballista can easily accommodate such orthogonal deconstruction of its test data types. This provides a greater ability to distinguish which aspect of a parameter might be causing a specific robustness failure (in addition to decreasing code complexity). Consider the type Sfio_t* in Figure 2, a function that seeks to a specific location in a file. There are 3 orthogonal categories, with 5, 3, and 13 possible values each. Thus there are 5 x 3 x 13 possible generated values for Sfio_t*, or 195 total values.

Ballista uses the interface description information to create all possible combinations of parameters to form an exhaustive list of test cases. For instance, suppose a MuT took one parameter of type Sfio_t* and an integer as input parameters. Given that there are 22 integer test values and 195 Sfio_t* values, Ballista would generate 22 x195 x 11 = 4290 test cases.

3.2.2 Implementation of test values

Data types for testing can themselves be thought of as modules, which fit into a hierarchical object-oriented class structure. This simplifies the creation of new data types and the refinement of existing types. Each data type is derived from a parent type, and inherits all of the parent's test

cases up to the root Ballista type object. For instance, a data type created to specifically represent a date string would have specific test cases associated with it that might include invalid dates, valid dates, dates far in the past, and dates in the future (Figure 3). Assuming the direct parent of the date string data module were string, it would inherit all of the test cases associated with a generic string, such as an empty string, large "garbage" strings, and small "garbage" strings. Finally the generic string might inherit from a generic pointer data type which would be a base type and include a test for a NULL pointer.

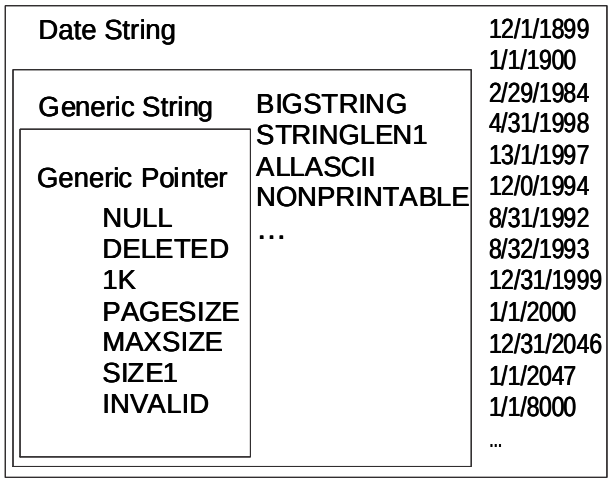


Figure 3. A Date String type inherits tests from Generic String and Generic Pointer

Each test value (such as Sfio_T_Closed_Buffered_Malloc, or Int_zero in Figure 2) refers to a set of code fragments that are kept in a simple database comprised of a specially formatted text file. The first fragment for each test value is a constructor that is called before the test case is executed (it is not literally a C++ constructor, but rather a code fragment identified to the test harness as constructing the instance of a test value). The constructor may simply return a value (such as a NULL), but may also do something more complicated that initializes system state. For example, the constructor for Sfio_T_Closed_Buffered_Malloc creates a safe/fast file with a call to `malloc`, puts a predetermined set of bytes into the file, opens the file for read, then returns a pointer to the file structure.

The second of the code fragments for each test value is commit code that changes any values necessary prior to calling the test. For example, the commit code for Sfio_T_Closed_Buffered_Malloc closes the file.

The last fragment deletes any data structures or files created by the corresponding constructor. For example, the destructor for Sfio_T_Closed_Buffered_Malloc deletes the file created by its matching constructor and frees the buffer.

Tests are executed from within a test harness by having a parent task fork a fresh child process for every test case. The child process first calls constructors for all test values used in a selected test case, then executes a call to the MuT with those test values, then calls destructors for all the test values used. Special care is taken to ensure that any robustness failure is a result of the MuT, and not attributable to the constructors or destructors themselves. Functions implemented as macros are tested using the same technique, and require no special treatment.

The test values used in the experiments were a combination of values suggested in the testing literature (*e.g.*, [marick95]) and values selected based on personal experience. For example, consider file descriptor test values. File descriptor test values include descriptors to existing files, negative one, the maximum integer number (MAXINT), and zero. Situations that are likely to be exceptional in only some contexts are tested, including file open only for read and file open only for write. File descriptors are also tested for inherently exceptional situations such as file created and opened for read, but then deleted from the file system without the program's knowledge.

The guideline for test value selection for all data types were to include, as appropriate: zero, negative one, maximum/minimum representable values, pointers to non-existent memory, lengths near virtual memory page size, pointers to heap-allocated memory, files open for combinations of read/write with and without exceptional permission settings, and files/data structures that had been released before the test itself was executed. While creating generically applicable rules for thorough test value selection remains a subject of future work, this experience-driven approach was sufficient to produce useful results.

It is important to note that this testing methodology does not generate tests based on a description of MuT functionality, but rather on the data types of the MuT's arguments. This approach means that per-MuT "scaffolding" code does not need to be written. As a result, the Ballista testing method is highly scalable with respect to the amount of effort required per MuT, needing only 20 data types to test 233 POSIX MuTs.

Data types are created through the use of a stand-alone data type compiler. The compiler, completed as a part of this work, allows data types to be written in a higher level language that makes it easy to define the orthogonal characteristics of a data type, and associate them with the appropriate code fragments for creation. The compiler takes the high level specification and generates a C++ object that can be linked in with the Ballista testing system. Appendix B contains a sample template file (data type description), as well as the code generated by the compiler.

An important benefit derived from the Ballista testing implementation is the ability to automatically generate the source code for any single test case the suite is capable of running. In many cases only a score of lines or fewer, these short programs contain the constructors for each parameter, the actual function call, and destructors. These single test case programs have been used to reproduce robustness failures in isolation for use by OS developers and to verify test result data.

3.2.3 Testing results

Ballista categorizes test results according to the CRASH severity scale: [kropp98]

- Catastrophic failures occur when the OS itself becomes corrupted or the machine crashes and reboots.
- Restart failures occur when a call to a MuT never returns control to the caller, meaning that in an application program a single task would be "hung," requiring intervention to terminate and restart that task. These failures are identified by a watchdog timer which times out after several seconds of waiting for a test case to complete. (Calls that wait for I/O and other such legitimate "hangs" are not tested.)
- Abort failures tend to be the most prevalent, and result in abnormal termination (a "core dump," the POSIX SIGSEGV signal, *etc.*) of a single task.
- Silent failures occur when the MuT returns with no indication of error when asked to perform an operation on an exceptional value. For example, the floating point libraries distributed with several operating systems fail to return an error code, and instead return as if the result were accurate when computing the logarithm of zero[devale99]. Silent failures

can currently be identified only by using inferences from N-version result voting, and so are not identified by the Ballista testing service.

- Hindering failures occur when an incorrect error code is returned from a MuT, which could make it more difficult to execute appropriate error recovery. Hindering failures have been observed as fairly common (forming a substantial fraction of cases which returned error codes) in previous work [koopman97], but are not further discussed due to lack of a way to perform automated identification within large experimental data sets.

There are two additional possible outcomes of executing a test case. It is possible that a test case returns with an error code that is appropriate for invalid parameters forming the test case. This is a case in which the test case passes – in other words, generating an error code is the correct response. Additionally, in some tests the MuT legitimately returns no error code and successfully completes the requested operation. This happens when the parameters in the test case happen to be all valid, or when it is unreasonable to expect the OS to detect an exceptional situation (such as pointing to an address past the end of a buffer, but not so far past as to go beyond a virtual memory page or other protection boundary).

Although the Ballista methodology seems unsophisticated, it uncovers a surprising number of robustness failures even in mature software systems. For example, when used to test 233 function calls across 15 different POSIX compliant operating systems, Ballista found normalized robustness failure rates from 9.99% to 22.69%, with a mean of 15.16% [devale99]. Testing of other software systems seems to indicate comparable failure rates.

3.3 An example of testing

In order to illustrate the usage of Ballista, consider the code segment for performing trapezoidal approximation of numerical integration given in Figure 4. It is typical of textbook algorithms in that is is efficient, and easy to understand. The testing was done on a Digital Unix Alpha Station 500, running Digital Unix 4.0D.

All that is required is to simply add a line to a configuration file that describes the function or module to be tested. This allows Ballista to locate the interface (.h) file, the library, binary or source code of the module, and which data types to use during the test. Ballista processes the

information supplied by a user, performs the tests, and builds an HTML page describing the results, including links that generate the source code required to replicate any robustness failures.

For this example, Ballista ran a total of 2662 tests on the inttrap() function in Figure 4. Ballista found 807 tests which caused abort failures and 121 tests which caused restart failures, for a robustness failure rate of 35%. Of course, it is important to realize that any failure rate is going to be directly dependant on the test cases themselves. In this case the MuT was passed two floats and an integer. There were several values for both float and int that were not exceptional for inttrap() (such as e and pi). This means that any failure rates are relative, and any comparisons to be made among multiple modules should use the same data type implementations.

On the surface, it seems astonishing that a simple published algorithm such as this would have so many robustness failures. However, in all fairness, this is a textbook algorithm intended to convey how to perform a particular operation, and is presumably not intended to be "bullet-proof" software. Additionally, the Ballista methodology tends to bombard a MuT with many test cases containing exceptional conditions which generate failures having the same root cause. One must examine the details of the test results to obtain more insight into the robustness of the MuT.

By looking at the results more closely, it became apparent that all of the restart failures were due to the integer value of MAXINT being used as the value of N, the number of subdivisions. In this case, the algorithm attempted to sum the areas of roughly 2 billion trapezoids. After waiting about 1 second (this value is user configurable) Ballista decided that the process had hung, terminated it, and classified it as a restart failure. While there may be some applications in which this is not a restart failure, in many cases it would be one, and it seems reasonable to flag this situation

```
double f(double i)
{
  return  i*i*i;
}
double inttrap(double a,double b,int N)
{
   int i; double t=0; double w = (b-a)/N;
   for (i=1;i<=N;i++)
     t+= w*(f(a+(i-1)*w)+f(a+i*w))/2;
   return t;
}
```

Figure 4. Trapezoidal numerical integration algorithm [sedgewick92] and target function f.

as a potential problem spot. Thus, the importance of Restart failures found by Ballista varies depending on timing constraints of the application, and will detect true infinite loops if present.

Of the 807 abort failures detected by Ballista for this example, 121 were directly caused by a divide by zero floating point exception. This is probably the easiest type of exceptional condition to anticipate and test for. Seasoned programmers can be expected to recognize this potential problem and put a zero test in the function. Unfortunately, there is a large amount of software developed by relatively unskilled programmers (who may have no formal training in software engineering), and even trained programmers who have little knowledge about how to write robust software.

The remaining 686 aborts were due to overflow/underflow floating point exceptions. This is an insidious robustness failure, in that nearly every mathematical function has the potential to overflow or underflow. This exception is commonly overlooked by programmers, and can be difficult to handle. Nonetheless, such problems have the potential to cause a software system to fail and should be handled appropriately if robust operation is desired.

The floating point class of exceptions can be especially problematic, especially in systems whose microarchitecture by default masks them out. Although this may seem incongruous, the fact that the exception is masked out may lead to problems when hardening algorithms against robustness failures. If the exception does not cause an abort and the software does not check to make certain that an overflow/underflow did not occur, Silent failures (undetectable by the current Ballista service) may result. Resultant calculations may be at best indeterminate and at worst wrong, depending on the application; thus it is strongly recommended that full IEEE floating point features [IEEE85] such as propagation of NaN ("Not A Number") values be used even if Ballista testing has been applied to a computational software module.

3.4 Generalizing the Approach

The first-generation, standalone Ballista test harness was developed having in mind POSIX operating system calls as a full-size example system. The results achieved the goals of portability and scalability. However, there were two significant limitations. The POSIX testing harness assumed that any signal ("thrown" exception) was a robustness failure because the POSIX

standard considers only error return codes to be robust responses. Second, the result that POSIX testing required absolutely no scaffolding code for creating tests was a bit too good to be true for general software involving distributed computing environment initialization and call-backs. However, both of these issues have been addressed in the Web-based testing system.

3.4.1 Support for exception handling models

The Ballista framework supports both the termination model and resumption model as they are implemented in standard C/C++. With the termination model Ballista assumes that exceptions are thrown with some amount of detail via the C++ try-throw-catch mechanism so that corrective action can be taken. In the case of error code returns, it is assumed that there is an error variable that can be checked to determine if an error has occured (currently the `errno` variable from POSIX is supported). Thus, an "unknown" exception or a condition that results in an unhandled generic exception such as a SIGSEGV segmentation violation in a POSIX operating system is considered a robustness failure.

Additionally, Ballista has the ability to add user defined exception handlers to support those modules which use C++ exception handling techniques. Ballista embeds each test call into a standard try-catch pair. Any thrown exception not caught by the user defined catch statements will be handled by the last `catch (...)` in the test harness and treated as an Abort robustness failure.

3.4.2 Support for callbacks and scaffolding

Fernsler [fernsler99] used the Ballista tool to test the High Level Architecture Run Time Infrastructure (HLA RTI — a simulation backplane system used for distributed military simulations[dahman97]). This example system brought out the need for dealing with exceptions, since errors are handled with a set of thrown exceptions specified by the API. Additionally, the HLA RTI presented problems in that a piece of client software must go through a sequence of events to create and register data structures with a central application server before modules can be called for testing.

While at first it seemed that per-function scaffolding might be required for the HLA RTI, it turned out that there were only 12 equivalence classes of modules with each equivalence class

able to share the same scaffolding code. Thus the Ballista Web testing service has the ability to specify scaffolding code (a preamble and a postamble) that can be used with a set of modules to be tested. The preamble also provides a place for putting application-specific "include" files and so on. While there are no doubt some applications where clustering of functions into sets that share scaffolding code is not possible, it seems plausible that this technique will work in many cases, achieving scalability in terms of effort to test a large set of modules.

There are some software systems which require a calling object to be able to support a series of callbacks, either as independent functions or methods associated with the calling object. Two examples of this are a function that analyzes incoming events and dispatches them to registered event handlers, or a registration function which uses member functions of the calling object to obtain the information it requires. These situations require the testing framework itself to be enhanced with either the functions or appropriate object structure to handle these cases.

To facilitate Ballista's ability to test these types of systems we provide a user the ability to build the main testing function call into an arbitrary sub-class, or add any independent functions needed. As with the other customizable features of Ballista, the additions can be changed between modules. Although this feature adds back some of the per function scaffolding that we eschewed in the previous version of Ballista, it was added to allow testing of code that requires it if desired (and it may be completely ignored by many users).

3.4.3 Phantom parameters – a generalization of the Ballista testing method

At first glance, the parameter-based testing method used by Ballista seems limited in applicability. However, a bit of creativity allows it to generalize to software modules without parameters, modules that take input from files rather than parameters, and tests for how system state affects modules that do not have parameters directly related to that state.

In order to test a module without parameters, all that need be done is create a dummy module that sets appropriate system state with a parameter, then calls the module to be tested. For example, testing a pseudo-random number generator might require setting a starting seed value, then calling a parameterless function that returns a random number in a predefined range. While this could be accomplished by creating preamble test scaffolding to set the starting seed, the

Ballista tool supports a more elegant approach. When specifying parameters for a module, "phantom" parameters can be added to the list. These parameters are exercised by Ballista and have constructor/destructor pairs called, but are not actually passed to the MuT. So, for example, testing the random number generator is best done by creating a data type of "random number seed" that sets various values of interest, and then using that data type as a phantom parameter when testing the otherwise parameterless random number generator.

A similar approach can be taken for file-based module inputs rather than parameter-based inputs. A phantom parameter can be defined which creates a particular file or data object (or set of such objects) with a particular format. If that object is accessed other than with a parameter (for example, by referencing a global variable or looking up the object in a central registry) the phantom parameter can appropriately register the object. In a future version of Ballista, phantom parameters might be able to themselves take parameters, so that for instance a file name could be passed to a generic file creation data type in support of this scheme, and recursive/iterative creation of test sets could be performed.

Given the concept of phantom parameters, it becomes clear how to test modules for system states which are not reflected in explicit parameters. Phantom parameters can be added to set system state as desired. As an example, a phantom parameter data type might be created to fill up disk space before a disk accessing function is called.

Thus, Ballista testing has a reasonable (although certainly not complete) ability to test not only single module calls with single sets of parameters, but the effects of such calls in a wide variety of system states. With the extension of phantom parameters, it can provide highly deterministic testing of what amounts to a wide variety of sequences of function calls (reflected in system state set by that sequence of calls).

3.5 Summary

Ballista can test a large variety of software modules for robustness to exceptional input conditions. It is highly scalable, and can provide a rich state in which to test modules. Early versions of this testing approach found robustness failure rates ranging from 10% to 23% on a

full-scale test of fifteen POSIX operating system implementations. Any module capable of being linked into GNU C++ binaries can be tested by the Ballista system.

Ballista's ability to abstract testing to the data type/API level is, in part, responsible for its scalability. Functions or methods that operate on similar data types can use the same Ballista test type for input creation. This minimizes the amount of code that needs to be written to run the tests. Due to this abstraction, test effort (in terms of test development time) tends to scale sub-linearly with the number of functions to be tested.

The Ballista data type compiler allows the easy creation of new testing data types. It allows the developer to write test objects in a simple, abstract language that is powerful enough to express orthogonal properties and associate them with code needed to create the test values at run time.

4 The exception handling effectiveness of POSIX operating systems

The genesis of the work presented in this thesis involved the testing and analysis of POSIX operating systems. This provided us a rich set of available test targets built by professional teams, with operating stability as an explicit goal. Although we suspected that the tested systems would not be as robust as the authors seemed to believe, we were surprised at how poor the robustness of some systems was. The results of the investigation presented in this chapter afforded us the opportunity to interact with OS developers who were interested in our results. It was during those discussions that is became clear that the prevailing wisdom was that building a robust system would be too complex, too hard, and the resultant system would be too slow.

Operating systems form a foundation for robust application software, making it important to understand how effective they are at handling exceptional conditions. The Ballista testing system was used to characterize the handling of exceptional input parameter values for up to 233 POSIX functions and system calls on each of 15 widely used operating system (OS) implementations. This identified ways to crash systems with a single call, ways to cause task hangs within OS code, ways to cause abnormal task termination within OS and library code, failures to implement defined POSIX functionality, and failures to report unsuccessful operations.

Overall, only 55% to 76% of the exceptional tests performed generated error codes, depending on the operating system being tested. Approximately 6% to 19% of tests failed to generate any indication of error despite exceptional inputs. Approximately 1% to 3% of tests revealed failures to implement defined POSIX functionality for unusual, but specified, situations. Between 18% and 33% of exceptional tests caused the abnormal termination of an OS system call or library function, and five systems were completely crashed by individual system calls with exceptional parameter values. The most prevalent sources of these robustness failures were illegal pointer values, numeric overflows, and end-of-file overruns. There is significant opportunity for improving exception handling within OS calls and especially within C library functions. However, the role of signals *vs.* error return codes is both controversial and the source of divergent implementation philosophies, forming a potential barrier to writing portable, robust applications.

4.1 Introduction

The robustness of a system can depend in large part on the quality of exception handling of its operating system (*OS*). It is difficult to produce a robust software application, but the task becomes even more difficult if the underlying OS upon which the application is built does not provide extensive exception handling support. This is true not only of desktop computing systems, but also of embedded systems such as telecommunications and transportation applications that are now being built atop commercial operating systems. A trend in new Application Programming Interfaces (*APIs*) is to require comprehensive exception handling (*e.g.*, CORBA [OMG95] and HLA [DoD98]). Unfortunately, while the POSIX API [IEEE94] provides a mechanism for exception reporting in the form of error return codes, implementation of this mechanism is largely optional. This results in uncertainty when adopting a POSIX operating system for use in a critical system, and leads to two important questions. 1) Given the lack of a firm requirement for robustness by the POSIX standard, how robust are actual Commercial Off-The-Shelf (COTS) POSIX implementations? 2) What should application programmers do to minimize the effects of non-robust OS behavior?

These questions can be answered by creating a direct, repeatable, quantitative assessment of OS exception handling abilities. Such an evaluation technique would give the developers feedback about a new OS version before it is released, and present the opportunity to measure the effectiveness of attempts to improve robustness. Additionally, quantitative assessment would enable system designers to make informed comparison shopping decisions when selecting an OS, and would support an educated "make/buy" decision as to whether a COTS OS might in fact be more robust than an existing proprietary OS. Alternately, knowledge about the exception handling weak spots of an OS would enable application designers to take extra precautions in known problem spots.

POSIX exception handling tests were conducted using Ballista on fifteen POSIX operating system versions from ten different vendors across a variety of hardware platforms. More than one million tests were executed in all, covering up to 233 distinct functions and system calls for

each OS. Many of the tests identified instances in which exceptional conditions were handled in a non-robust manner, ranging in severity from complete system crashes to false indication of success for system calls. Other tests managed to uncover exception-related software defects that apparently were not caught by the POSIX certification process.

Beyond the robustness failure rates measured, analysis of test data and discussion with OS vendors reveals a divergence in approaches to dealing with exceptional parameter values. Some operating systems attempt to use the POSIX-documented error codes to provide portable support for exception reporting at run time. Alternately, some operating systems emphasize the generation of a signal (typically resulting in abnormal process termination) when exceptional parameter values are encountered in order to facilitate debugging. However, there is no way to generalize which OSs handle what situations in a particular manner, and all OSs studied failed to provide either indication of exceptions in a substantial portion of tests conducted. While it is indeed true that the POSIX standard itself does not *require* comprehensive exception reporting, it seems likely that a growing number of applications will need it. Evaluating current operating systems with respect to exception handling is an important first step in understanding whether change is needed, and what directions it might take.

The following sections describe the testing methodology used (briefly), robustness testing results, what these results reveal about current operating systems, and potential directions for future research.

4.2 Ballista testing methodology for POSIX

The Ballista approach to robustness testing has been implemented for a set of 233 POSIX functions and calls defined in the IEEE 1003.1b standard [IEEE94] ("POSIX.1b" or "POSIX with real-time extensions with C language binding"). All standard calls and functions were tested except for calls that take no arguments, such as `getpid()`; calls that do not return, such as `exit()`; and calls that intentionally send signals, such as `kill()`.

For each POSIX function tested, an interface description was created with the function name and type information for each argument. In some cases, specific information about argument use

was exploited to result in better testing (for example, a file descriptor might be of type `int`, but was described to Ballista as a more specific file descriptor data type).

As an example, Figure 5 shows the actual test values used to test `write(int filedes, const void *buffer, size_t nbytes)`, which takes parameters specifying a file descriptor, a memory buffer, and a number of bytes to be written. This example is of an older version of Ballista used in the collection of the OS data and is a precurser to the previously illustrated testing example that showed the more sophisticated example using the "dials" approach. Because `write()` takes three parameters of three different data types, Ballista draws test values from separate test objects established for each of the three data types. In Figure 5, the arrows indicate that the particular test case being constructed will test a file descriptor for a file which has been opened with only read access, a NULL pointer to the buffer, and a size of 16

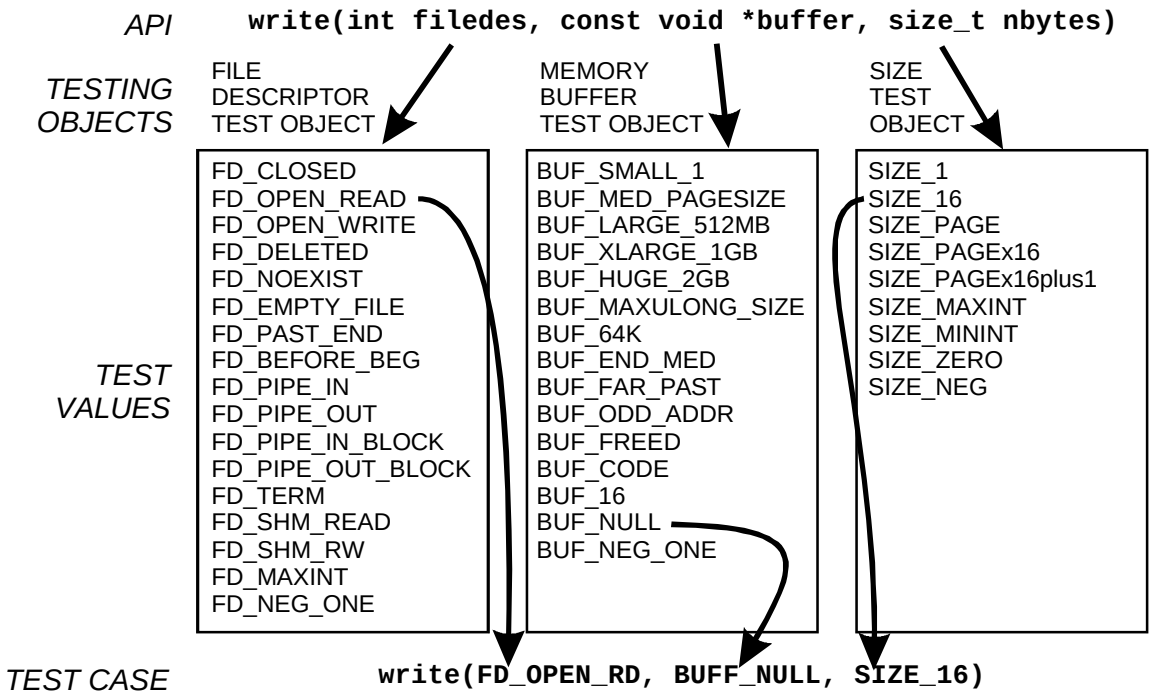


Figure 5. Ballista test case generation for the `write()` function. The arrows show a single test case being generated from three particular test values; in general, all combinations of test values are tried in the course of testing.

bytes. Other combinations of test values are assembled to create other test cases. In the usual case, all combinations of test values are generated to create a combinatorial number of test cases. For a half-dozen POSIX calls, the number of parameters is large enough to yield too many test cases for exhaustive coverage within a reasonable execution time. In these cases a pseudo-random sampling of 5000 test cases is used. (Based on a comparison to a run with exhaustive searching on one OS, this sampling gives results accurate to within 1 percentage point for each function.)

4.2.1 Categorizing test results

After each test case is executed, the Ballista test harness categorizes the test results according to the first three letters of the "C.R.A.S.H." severity scale [kropp98]. Although Silent failures are not normally detectable using the Ballista system, we were able to infer Silent failures in this experiment. The details of this are explained in Section 4.4.

4.3 Results

A total of 1,082,541 test cases were executed during data collection. Operating systems which supported all of the 233 selected POSIX functions and system calls each had 92,658 total test cases, but those supporting a subset of the functionality tested had fewer test cases.

4.3.1 Raw Testing Results

The compilers and libraries used to generate the test suite were those provided by the OS vendor. In the case of FreeBSD, NetBSD, Linux, and LynxOS, the GNU C compiler version 2.7.2.3 and associated C libraries were used to build the test suite.

Table 1 reports the robustness failure rates as measured by Ballista. In all, there were five MuTs across the tested systems that resulted in Catastrophic failures. Restart failures were relatively scarce, but present in all but two operating systems. Abort failures were common, indicating that in all operating systems it is relatively straightforward to elicit an abnormal task termination from an instruction within a function or system call (Abort failures do not have to do with subsequent use of an exceptional value returned from a system call – they happen in response to an instruction within the vendor-provided software itself).

Any MuT that suffered Catastrophic failures could not be completely tested due to a lack of time for multiple reboots on borrowed equipment, and thus is excluded from failure rate calculations beyond simply reporting the number of MuTs with such failures. A representative test case causing a Catastrophic failure on Irix 6.2 is:

```
munmap(malloc((1<<30+1)), MAXINT);
```

Similarly, the following call crashes the entire OS on Digital Unix (OSF 1) version 4.0D:

```
mprotect (malloc ((1 << 29) + 1), 65537, 0);
```

Other calls causing Catastrophic failures were: `munmap()` on QNX 4.22, `mprotect()` on QNX 4.22, `mmap()` on HPUX 10, `setpgid()` on LynxOS, and `mq_receive()` on Digital Unix/OSF 3.2. Note that the tables all report Digital Unix version 4.0B, which did not have the Catastrophic failure found in 4.0D, but is otherwise quite similar in behavior.

4.3.2 Normalized Failure Rate Results

Comparing OS implementations simply on the basis of the number of tests that fail is problematic because, while identical tests were attempted on each OS, different OS implementations supported differing subsets of POSIX functionality. Furthermore, MuTs having many parameters execute a large number of test cases, potentially skewing results.

Rather than use raw test results, comparisons should be made based on normalized failure rates. The rightmost column of Table 1 show the normalized failure rates computed by the following process. A ratio of robustness failures to total tests is computed for each MuT within each OS (*e.g.*, a ratio of 0.6 means that 60% of the tests failed). Then, the mean ratio across all MuTs for an OS is computed using a simple arithmetic average. This definition produces an exposure metric, which gives the probability that exceptional parameter values of the types tested will cause a robustness failure for a particular OS. This metric has the advantage of removing the effects of differing number of tests per function, and also permits comparing OS implementations with differing numbers of functions implemented according to a single normalized metric.

Overall failure rates considering both Abort and Restart failures range from the low of 9.99% (AIX) to a high of 22.69% (QNX 4.24). As shown in Figure 6, the bulk of the failures found are

System	POSIX Fns. Tested	Fns. with Catastro-phic Failures	Fns. with Restart Failures	Fns with Abort Failures	Fns. with No Failures	Number of Tests	Abort Failures	Restart Failures	Normalized Abort + Restart Rate
AIX 4.1	186	0	4	77	108	64009	11559	13	9.99%
FreeBSD 2.2.5	175	0	4	98	77	57755	14794	83	20.28
HPUX 9.05	186	0	3	87	98	63913	11208	13	11.39
HPUX 10.20	185	**1**	2	93	92	54996	10717	7	13.05
IRIX 5.3	189	0	2	99	90	57967	10642	6	14.45
IRIX 6.2	225	**1**	0	94	131	91470	15086	0	12.62
Linux 2.0.18	190	0	3	86	104	64513	11986	9	12.54
Lynx 2.4.0	222	**1**	0	108	114	76462	14612	0	11.89
NetBSD 1.3	182	0	4	99	83	60627	14904	49	16.39
OSF1 3.2	232	**1**	2	136	96	92628	18074	17	15.63
OSF1 4.0B	233	0	2	124	109	92658	18316	17	15.07
QNX 4.22	203	**2**	6	125	75	73488	20068	505	20.99
QNX 4.24	206	0	4	127	77	74893	22265	655	22.69
SunOS 4.13	189	0	2	104	85	64503	14227	7	15.84
SunOS 5.5	233	0	2	103	129	92658	15376	28	14.55

Table 1. Directly measured robustness failures for fifteen POSIX operating systems.

Abort failures. OS implementations having Catastrophic failures are annotated with the number of MuTs capable of causing a system crash.

The first set of experimental data gathered included several relatively old OS versions, representing machines that were in service under a conservative campus-wide software upgrade policy. At the insistence of vendors that newer versions would be dramatically better, tests were run on several borrowed machines configured with the newest available OS releases. The results showed that even major version upgrades did not necessarily improve exception handling capabilities. Failure rates were reduced from Irix 5.3 to Irix 6.2, from OSF 3.2 to OSF 4.0, and from SunOS 4 to SunOS 5, although in all cases the improvement was not overwhelming. However, the failure rates actually increased from HPUX 9 to HPUX 10 (including addition of a

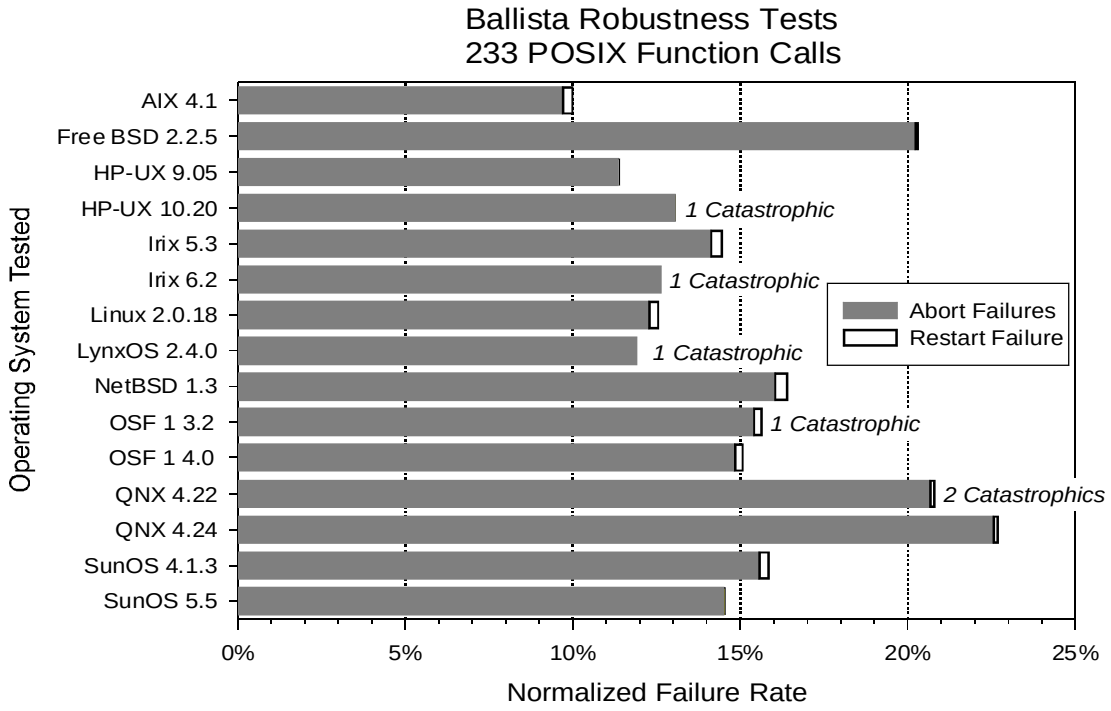


Figure 6. Normalized failure rates for POSIX operating systems

Catastrophic failure mode), increased from QNX 4.22 to QNX 4.24 (although with elimination of both Catastrophic failure modes), and stayed essentially identical from OSF 1 4.0B to OSF 1 4.0D (although 4.0D introduced a new Catastrophic failure mode).

4.3.3 Failure Rates Weighted By Operational Profile

The use of a uniformly weighted average gives a convenient single-number metric for comparison purposes. However, it is important to dig a little deeper into the data to determine what functions are driving the failure rates, and whether they are the functions that are frequently used, or instead whether they are obscure functions that don't matter most of the time.

In some situations it may be desirable to weight vulnerability to exception handling failures by the relative frequency of invocation of each possible function. In other words, rather than using an equal weighting when averaging the failure rates of each MuT in an OS, the average could instead be weighted by relative execution frequency for a "typical" program or set of programs.

This approach corresponds to a simple version of operational profiles as used in traditional software testing [musa96].

Collecting profiling information at the OS system call and function level turned out to be surprisingly difficult for the POSIX API, because most tools are optimized for instrumenting user-written calls rather than OS calls. However, instrumentation of the IBS benchmark suite [uhlig95] and the floating point portions of the SPEC95 benchmark suite were possible using the Atom tool set [srivastava94] running on Digital Unix to record the number of calls made at run time. Due to problems with compiler option incompatibility between Atom and the benchmark programs, only the IOZone, compress, ftp, and gnuChess programs from IBS were measured.

The results were that the weighted failure rates vary dramatically in both magnitude and distribution among operating systems, depending on the workload being executed. For example, IBS weighted failure rates varied from 19% to 29% depending on the operating system. However, for SPEC95 floating point programs, the weighted failure rate was less than 1% for all operating systems except FreeBSD. Because FreeBSD intentionally uses a SIGFPE floating point exception signal instead of error return codes in many cases, it happens have a high percentage of Abort results on functions heavily used by SPEC95.

Specific weighted failure rates are not described because the results of attempting operational profiling point out that there is no single operational profile that makes sense for an interface as versatile as an OS API. The only definite point that can be made is that there are clearly some profiles for which the robustness failure rates could be significant. Beyond that, publishing a weighted average would, at best, be overly simplistic. Instead, interested readers are invited to obtain the raw OS failure rate data and apply operational profiles appropriate to their particular application area.

4.3.4 Failure Rates By Call/Function Category

A somewhat different way to view the failure rate data is by breaking up aggregate failure rates into separate failure rates grouped by the type of call or function [ostrand88]. This gives some general insight into the portions of the implementations that tend to be robust at handling exceptions without becoming bogged down in the details of individual call/function failure rates.

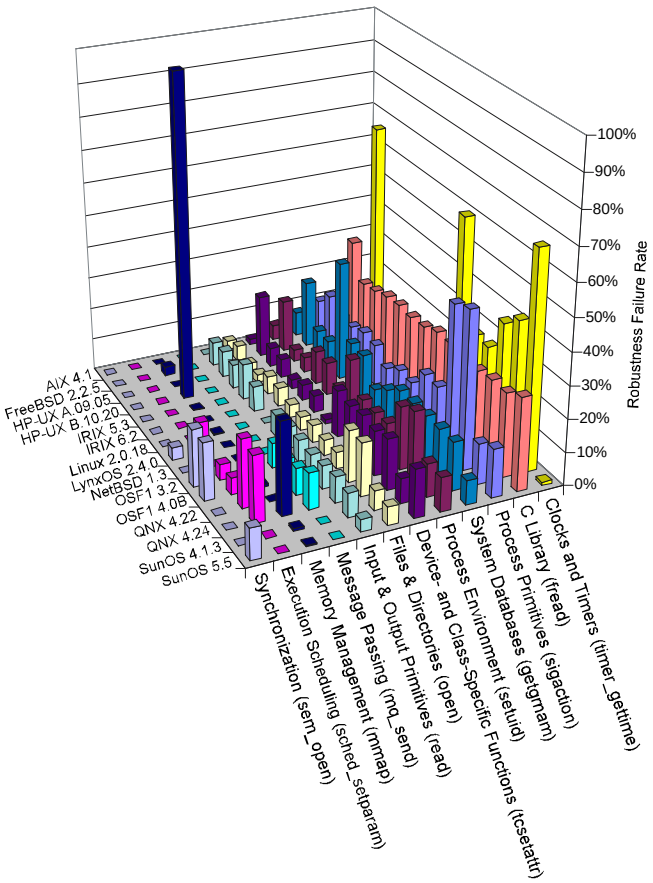


Figure 7. Normalized failure rates by call/function category, divided per the POSIX document chapters

Figure 7 shows the failure rate of different categories of calls/functions as grouped within the chapters of the POSIX specification [IEEE93]. In this figure both a general name for the category and an example function from that category are given for convenience. The failure rates for each category are calculated by taking the normalized average of the failure rates of each function in that category. For instance if category X had three member function with a 4%, 5%, and 6% failure rate, the category failure rate would be 5%.

Several categories in Figure 7 have pronounced failure rates. The clocks and timers category (Section 14 of the Standard) had a bimodal distribution of failure rates: 30% to 69% for seven of the OS implementations (the visible bars in Figure 7), and low values for the remaining OSs (the hidden bars are 7% for Irix 6.2, 1% for SunOS 5.5, and 0% for the rest). This set and the memory management set (Section 12 of the Standard, which deals with memory locking/mapping/sharing,

but not “malloc”-type C-library operations) are representative of areas in which there is a very noticeable variation among OS implementations with respect to exception handling.

While in many cases failure rates are comparable across OS implementations for the different call categories, there are some bars which show significantly higher failure rates. HPUX 10 has a 100% failure rate for memory management functions. Worse, it was one of the memory management calls that produced HPUX 10’s Catastrophic system failure, indicating that this area is indeed a robustness vulnerability compared to HPUX 9, which had no such problems. (We have learned that HPUX 10 has a new implementation of these functions, accounting for a potentially higher failure rate.) This and other similar observations indicate that there may be specific areas of reduced exception handling effectiveness within any particular OS.

4.3.5 C-Library Failure Rates

The general failure rate of the C library calls in Figure 7 is uniformly high across all OS implementations tested. Figure 8 shows the same data as Figure 6, but shows the portions of each failure rate that are attributable to the C library functions. Part of the large contribution of C library functions to overall failure rates is that they account for approximately half of the MuTs

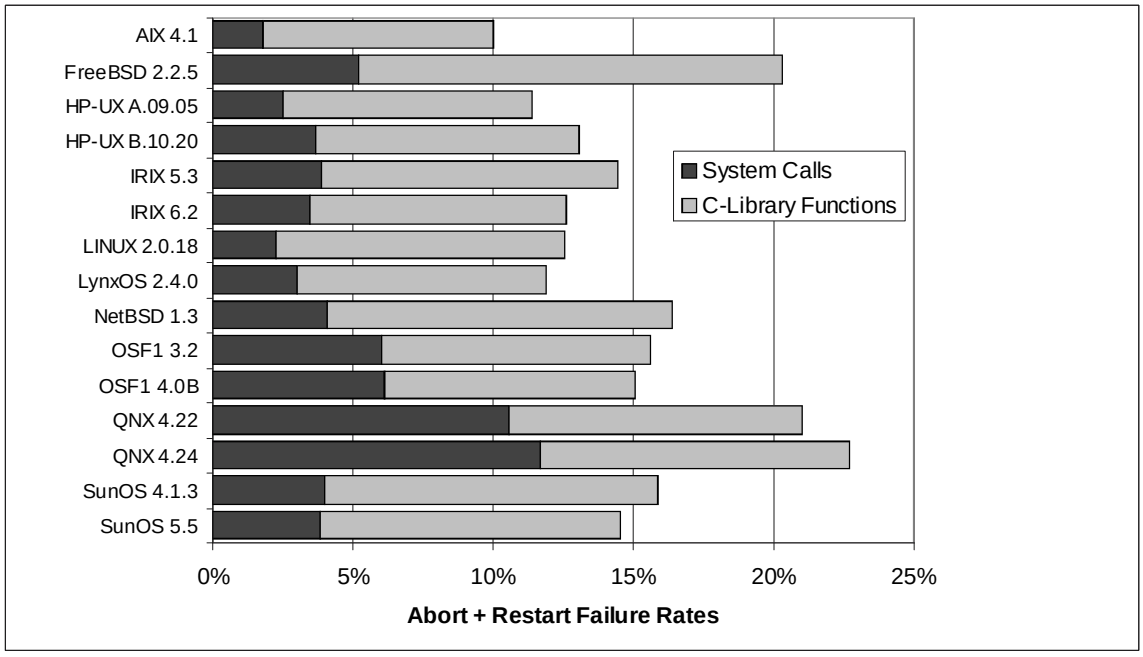


Figure 8. The C-Library functions contribute a large proportion of the overall raw failure rates

for each OS (a total of 102 C library functions were tested, out of 175 to 223 total MuTs per OS). The C library functions had a failure rate about the same as system calls on QNX, but had failure rates between 1.8 and 3.8 times higher than system calls on all other OS versions tested. Within the C library, string functions, time functions, and stream I/O tended to have the highest robustness failure rates.

4.4 Data analysis via N-version software voting

The scalability of the Ballista testing approach hinges on not needing to know the functional specification of a MuT. In the general case, this results in having no way to deal with tests that have no indication of error – they could either be non-exceptional test cases or Silent failures, depending on the actual functionality of the MuT. However, the availability of a number of operating systems that all conform to a standardized API permits estimating and refining failure rates using an idea inspired by multi-version software voting (*e.g.*, [avizienis85]). Ballista testing results for multiple implementations of a single API can be compared to identify test cases that are either non- exceptional, or that are likely to be Silent failures. This is, of course, not really multi-version software voting, but rather a similar sort of idea that identifies problems by finding areas in which the various versions disagree as to results for identical tests.

4.4.1 Elimination of non-exceptional tests.

The Ballista test cases carefully include some test values which are not exceptional in any way. This is done intentionally to prevent the masking of robustness failures. A correctly handled exceptional condition for one value in a tuple of those passed into a function may cause the system to not even look at other values. The concept is similar to obtaining high branch coverage for nested branches in traditional testing. For instance, in the test case: `write(-1,NULL,0)`, some operating systems test the third parameter, a length field of zero, and legitimately return with success on zero length regardless of other parameter values. Alternately, the file descriptor might be checked and an error code returned. Thus, having a second parameter value of a NULL pointer might never generate a robustness failure caused by a pointer dereference unless the file descriptor parameter and length fields were tested with

non-exceptional values. In other words, exceptional values that are correctly handled for one argument might mask non-robust handling of exceptional values for some other argument. If, on the other hand, the test case `write(FD_OPEN_WRITE, NULL, 16)`were executed, it might lead to an Abort failure when the NULL pointer is dereferenced. Additionally, test cases that are exceptional for some calls may non-exceptional for others (*e.g.*, using read permissions for testing `read()` vs. `write()`). Thus, by including non-exceptional test cases we force the module under test to attempt to handle each value that might be exceptional. While both uses of non-exceptional test values are important, they necessarily lead to test cases that are not, in fact, tests of exceptional conditions (*e.g.*, reading from a read-only file is not exceptional).

Multi-version software comparisons can prune non-exceptional test cases from the results data set. This is done by assuming that any test case in which all operating systems return with no indication of error are in fact non-exceptional tests (or, are exceptional tests which cannot be detected within reason on current computer systems). In all, 129,731 non-exceptional tests were removed across all 15 operating systems. Figure 9 shows the adjusted abort and restart failure

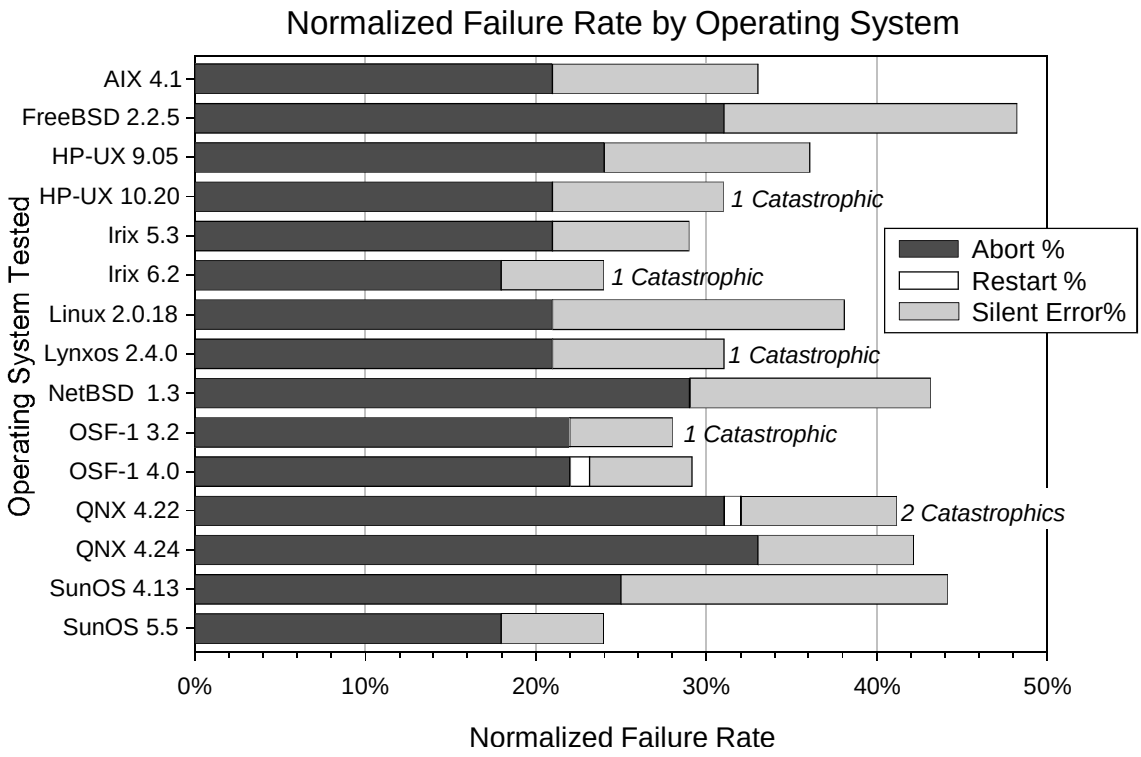


Figure 9. Adjusted, normalized robustness failure rates after using muti-version software techniques. Results are APPROXIMATE due to the use of heuristics.

rates after removing non-exceptional tests. Manual verification of 100 randomly selected test cases thus removed indicated that all of them were indeed non-exceptional, but it was impractical to examine a larger sample using this very labor-intensive process. While it is possible that some test cases were incorrectly removed, based on this sample and intuition gained during the sampling process, it seems unlikely that the number of false removals involved would materially affect the results.

4.4.2 An estimation of Silent failure rates

One of the potential problems with leaving out Silent failures in reporting results is that an OS might conceivably be designed to avoid generating Abort failures at any cost. For example, AIX intentionally permits reads (but not writes) to the memory page mapped to address zero to support legacy code, meaning that dereferences of a NULL pointer would not generate Abort failures. And, in fact, AIX does have a moderately high Silent failure rate because of this implementation decision.

Once the non-exceptional tests were removed, a multi-version software comparison technique was again used to detect Silent Failures. The heuristic used was that if at least one OS returns an error code, then all other operating systems should either return an error code or suffer some form of robustness failure (typically an Abort failure).

As an example, when attempting to compute the logarithm of zero, AIX, HPUX-10, and both versions of QNX completed the requested operation without an error code, whereas other OS implementations did return an error code. This indicated that AIX, HPUX-10, and QNX had suffered Silent robustness failures for that test case.

Of course the heuristic of detection based on a single OS reporting an error code is not perfect. Manual verification of 100 randomly sampled test cases, with each test case compared across all the OS implementations, indicates that approximately 80% of cases predicted to be Silent failures by this technique were actually Silent failures. Of the approximately 20% of test cases that were mis-classified:

- 28% were due to POSIX permitting discretion in how to handle an exceptional situation. For example, `mprotect()` is permitted, but not required, to return an error if the address of memory space does not fall on a page boundary.
- 21% were due to bugs in C library floating point routines returning false error codes. For example, Irix 5.3 returns an error for `tan(-1.0)` instead of the correct result of -1.557408. Two instances were found that are likely due to overflow of intermediate results – HPUX 9 returns an error code for `fmod(DBL_MAX,PI)`; and QNX 4.24 returns an error code for `ldexp(e,33)`
- 9% were due to lack of support for required POSIX features in QNX 4.22, which incorrectly returned errors for filenames having embedded spaces.
- The remaining 42% were instances in which it was not obvious whether an error code could reasonably be required. This was mainly a concern when passing a pointer to a structure containing meaningless data, where some operating systems (such as SunOS 4.13, which returned an error code for each test case it did not abort on) apparently checked the data for validity and returned an error code.

Examining potential Silent failures manually also revealed some software defects ("bugs") generated by unusual, but specified, situations. For instance, POSIX requires `int fdatasynch(int filedes)` to return the EBADF error if `filedes` is not valid, and the file open is for write [IEEE94]. Yet when tested, only one operating system, IRIX 6.2, implemented the specified behavior, with the other OS implementations failing to indicate that an error occurred. The POSIX standard also specifically permits writes to files past EOF, requiring the file length be updated to allow the write [IEEE94]; however only FreeBSD, Linux, and SunOS 4.13 returned successfully after an attempt to write data to a file past its EOF, while every other implementation returned EBADF. It is estimated that the failure rates for these problems is quite low (perhaps 1% to 3% overall depending on the OS), but is definitely present, and is apparently not caught by the process of validating POSIX compliance.

A second approach was attempted for detecting Silent failures based on comparing test cases having no error indication against instances of the same test case suffering an Abort failures on

some other OS. With some surprise, this turned out to be as good at revealing software defects as it was at identifying Silent failures. A relatively small number (37,434 total) of test cases generated an Abort failure for some operating systems, but completed with no error indication at all for other operating systems. But, manual verification 100 randomly sampled test cases indicated that this detection mechanism had approximately a 50% false alarm rate.

Part of the high false alarm rate for this second approach was due to differing orders for checking arguments among the various operating systems (related to the discussion of fault masking earlier). For example, reading bytes from an empty file to a NULL pointer memory location might abort if end-of-file is checked after attempting to move a byte, or return successfully with zero bytes having been read if end-of-file is checked before moving a byte. The other part of the false alarm rate was apparently due to limitations within floating point libraries. For instance, FreeBSD suffered an Abort failure on both `fabs(DBL_MAX)` and `fabs(-DBL_MAX)`whereas it should have returned without an error.

Based on these estimated accuracy rates, results reported in Figure 9 reflect only 80% of the Silent errors measured and 50% of the Silent Aborts measured, thus compensating for the estimated false alarm rates. With all of the manual examination techniques it was impractical to gather a much larger sample, so these percentages should be considered gross approximations, but are believed to be reasonably accurate based on intuition gained during the sampling process.

4.4.3 Frequent sources of robustness failure

Given that robustness failures are prevalent, what are the common sources? Source code to most of the operating systems tested was not available, and manual examination of available source code to search for root causes of robustness failures was impractical with such a large set of experimental data. Therefore, the best data available is based on a correlation of input values to robustness failures rather than analysis of causality. The test values most frequently associated with robustness failures are:

- 94.0% of invalid file pointers (excluding NULL) were associated with a robustness failure
- 82.5% of NULL file pointers were associated with a robustness failure

- 49.8% of invalid buffer pointers (excluding NULL) were associated with a robustness failure
- 46.0% of NULL buffer pointers were associated with a robustness failure
- 44.3% of MININT integer values were associated with a robustness failure
- 36.3% of MAXINT integer values were associated with a robustness failure

Perhaps surprisingly, system state changes induced by any particular test did not prove to be a source of robustness failures for other tests. Apparently the use of a separate process per test case provided sufficient inter-test isolation to contain the effects of damage to system state for all tests except Catastrophic failures. This was verified by vendors reproducing tests in isolation with single-test programs, and by verifying that test results remained the same even if tests were run in a different order within the test harness. This is not to say that such problems don't exist, but rather that they are rather more difficult to elicit on these operating systems than one might think.

4.5 Issues in attaining improved robustness

When preliminary testing results were shown to OS vendors, it became very apparent that some developers took a dim view of a SIGSEGV or SIGFPE signal being considered a robustness failure. In fact, in some cases the developers stated that they specifically and purposefully generated signals as an error reporting mechanism, in order to make it more difficult for developers to miss bugs. On the other hand, other developers provide extensive support for a wide variety of error return codes and make attempts to minimize abnormal task terminations from within system calls and library functions. The importance of such comprehensive exception handling was underscored by many conversations with application developers who develop critical systems. There are two parts to this story: the relative strengths and weaknesses of each philosophy, and whether either goal (robust return codes or signaling for all exceptions) was attained in practice.

4.5.1 Signals vs. error codes

While discussions with OS developers have proven that exception handling robustness is a controversial, even "religious" subject, the fact remains that there are significant applications in

several industries in which developers have stated very clearly that fine-grain error reporting is extremely desirable, and that signals accompanied by task restarts are unacceptable. These applications include telecommunication switches, railroad train controllers, real-time simulations, uninterruptible power supplies, factory automation control, ultra-high availability mainframe computers, and submarine navigation, to name a few real examples encountered during the course of this project. While these may not be the intended application areas for most OS authors, the fact is that COTS OS implementations are being pressed into service for such critical systems to meet cost and time-to-market constraints. Thus, evaluating the robustness of an OS is useful, even though robustness is not required by the POSIX standard.

That having been said, the results reported here suggest that there are issues at hand that go beyond a preference for signals vs. error return codes. One issue is simply that divergence in implementations hinders writing portable, robust applications. A second issue is that no operating systems examined actually succeeded in attaining a high degree of robustness, even if signals were considered to be a desirable exception reporting mechanism.

4.5.2 Building more robust systems

Traditionally, software robustness has been achieved through a variety of techniques such as checking error codes, performing range checks on values, and using testing to flush out problems. However, Ballista robustness testing results have eliminated any slender hopes that these approaches were entirely sufficient for critical systems. Checking error codes might work on one OS, but might not work when porting to another OS (or even to a minor version change of the same OS) which generates a SIGSEGV instead of an error code, or which generates no error code at all in response to an exceptional situation. Similarly, it is clear that POSIX functions often do not perform even a cursory check for NULL pointer values, which could be accomplished with minimal speed impact. Finally, vendor testing of OS implementations has been demonstrated to miss some very simple ways to cause system crashes in both major and minor version changes.

Thus, a useful additional step in building more robust systems is to use API-level fault injection such as that performed by the Ballista testing system. This will, at a minimum, identify certain classes of Catastrophic failures so that manual intervention can be performed via software

"wrappers" to screen out exceptional parameters for specific system calls, or to permit application developers to otherwise pay specific attention to eliminating the possibility of such situations.

For C library functions, it may be possible to use alternate libraries that are specifically designed for increased robustness. One example is the Safe Fast I/O library (SFIO) [korn91] that can replace portions of the C library. For system calls, one can select an existing OS that tends to have low failure rates as shown in Figure 8, if Abort failures are a primary concern. Or, one might even find it necessary to add extra parameter-checking wrappers around system calls to reduce Silent failure rates.

For any application it is important to realize that abnormal task terminations are to be expected as a matter of course, and provide for automatic recovery from such events. In some applications this is sufficient to attain a reasonable level of robustness. For other applications, this is merely a way to reduce the damage caused by a system failure, but is not a viable substitute for more robust error identification and recovery.

Finally, a potential long-term approach to increasing the robustness of OS implementations is to modify the POSIX standard to include a requirement for comprehensive exception handling, with no exception left undefined. While this might impose a modest performance penalty, it might well be viable as an optional (but well specified) extended feature set. Further research should be performed to quantify and reduce any associated performance penalties associated with increased exception handling abilities.

4.6 Summary of OS Results

The Ballista testing approach provides repeatable, scalable measurements for robustness with respect to exceptional parameter values. Over one million total tests were automatically generated for up to 233 POSIX function and system calls spanning fifteen operating systems. The most significant result was that no operating system displayed a high level of robustness. The normalized rate for robust handling of exceptional inputs ranged from a low of 52% for FreeBSD version 2.2.5 to a high of 76% for SunOS version 5.5 and Irix version 6.2. The majority of robustness failures were Abort failures (ranging from 18% to 33%), in which a signal was sent from the system call or library function itself, causing an abnormal task termination.

The next most prevalent failures were Silent failures (ranging from 6% to 19%), in which exceptional inputs to a Module under Test resulted in erroneous indication of successful completion. Additionally, five operating systems each had at least situation that caused a system crash in response to executing a single system call. The largest vulnerabilities to robustness failures occurred when processing illegal memory pointer values, illegal file pointer values, and extremely large integer and floating point numbers. In retrospect it is really no surprise that NULL pointers cause problems when passed to system calls. Regardless, the single most effective way to improve robustness for the operating systems examined would be to add tests for NULL pointer values for relevant calls.

Application of the Ballista testing approach to measuring OS robustness led to several insights. It documents a divergence of exception handling strategies between using error return codes and throwing signals in current operating systems, which may make it difficult to write portable robust applications. All operating systems examined had a large number of instances in which exceptional input parameter values resulted in an erroneous indication of success from a system call or library function, which would seem to further complicate creating robust applications.

The observations and insights gained from the application of Ballista to operating systems provided the impetus for the remainder of the investigation described in this thesis. It became clear that the developers we interacted with held very strong beliefs about the tractability of robust exception detection and handling in terms of complexity and performance. In response to the experimental results, many OS vendors stated that their code was as robust as possible given complexity and performance constraints. Without any real evidence to the contrary, we could not refute their opinions, and thus became determined to investigate further.

These observations led directly to the questions addressed in remainder of the research contained in this dissertation:

- Can code be more robust, or was there a fundamental limit as some claimed?
- Is it possible to build robust systems without losing significant performance?

- Is the fundamental problem one of education, or in other words, do developers understand robustness? Do they understand what it takes to make a system robust?

5 Hardening and analysis of math libraries

User pressure to provide the fastest possible executable may lead developers to leave out run-time data checks which may prevent software robustness failures at the expense of execution speed. While it can be show with a hand waving argument that these checks do indeed sacrifice performance, it is not clear what the impact is, and if it is even significant. This chapter presents the first attempt to quantify this penalty. The math library for FreeBSD 3.0 was modified to perform robust error detection and handling, and a separate version with all error checking and handling removed was created. At a function level, the robust library performed an average of 1% slower than the unmodified library, and on average 14% slower than the library with all error handling code removed. Further, application code written to explicitly check return codes after library function calls ran 2% slower on average than application code which neglected such checks.

5.1 Background

The result of testing operating systems and reporting their robustness failures spurred a great deal of discussion about the tractability and performance cost of detecting and gracefully handling exceptional conditions. The general consensus that it was too hard and too costly generated interest in determining if this was indeed the case, and if so, why.

As a simple expedient, the standard BSD libm.a source code was chosen for modification and testing. This made it a simple exercise to not only remove existing error detection and handling code, but also to add more robust error handling code. Since these functions have been rigorously mathematically defined, this library also enjoys having degenerate cases and input ranges which have been well defined for over a century.

Table 2 lists the functions which were selected for test. The functions were selected because each function takes a single double precision floating point number and returns a single double precision floating point number. This

acos
acosh
asinh
atan
atanh
cos
cosh
erf
exp
floor
gamma
log
sin
sinh
sqrt
tanh

Table 2. Test Function List

simplified creation of the performance testing harness, as well as robustness testing using the Ballista testing service.

Three distinct versions of the math library were analyzed. The first version was that of the original, unmodified source code as distributed with FreeBSD 3.0. The second was the original code, with all existing error checking and exception handling removed. The third version was the original source, modified to be robust.

To make the robust version of the math library, extensive acceptance testing was added to the function in the form of inline input filters. These filters determined if the operation could be completed and return a meaningful result, as well as complete without causing a floating point exception. If a condition existed that precluded successful completion of the function, `errno` was set to convey a meaningful error message identifying the exact condition preventing such completion.

5.2 Performance testing methodology

The performance of the libraries was measured on a Pentium-133 with 64 MB of main memory running FreeBSD 3.0. All source code was compiled with GCC version 2.7.2.3 with the -O3 optimization switch. A single program was written to exercise each of the functions listed in Table 2. The program code was then linked with the appropriate library – robust, stock, or non-robust, and then executed. The input values for the functions were carefully chosen so that they would cause the function to complete successfully, and not fall into any special cases. This resulted in the execution of each test that checked for exceptional conditions, to ensure the measured cost for enhancing robustness was determined for the non-exceptional case.

For each function tested, the time it took to execute the function 1 million times within a C `for` loop was measured. Each test was run 5 times, and the average is reported. This method provided a repeatable time measurement with reasonable assurance that the time measured is representative of the code speed. This complete procedure was performed for each library - robust, non-robust, and unmodified.

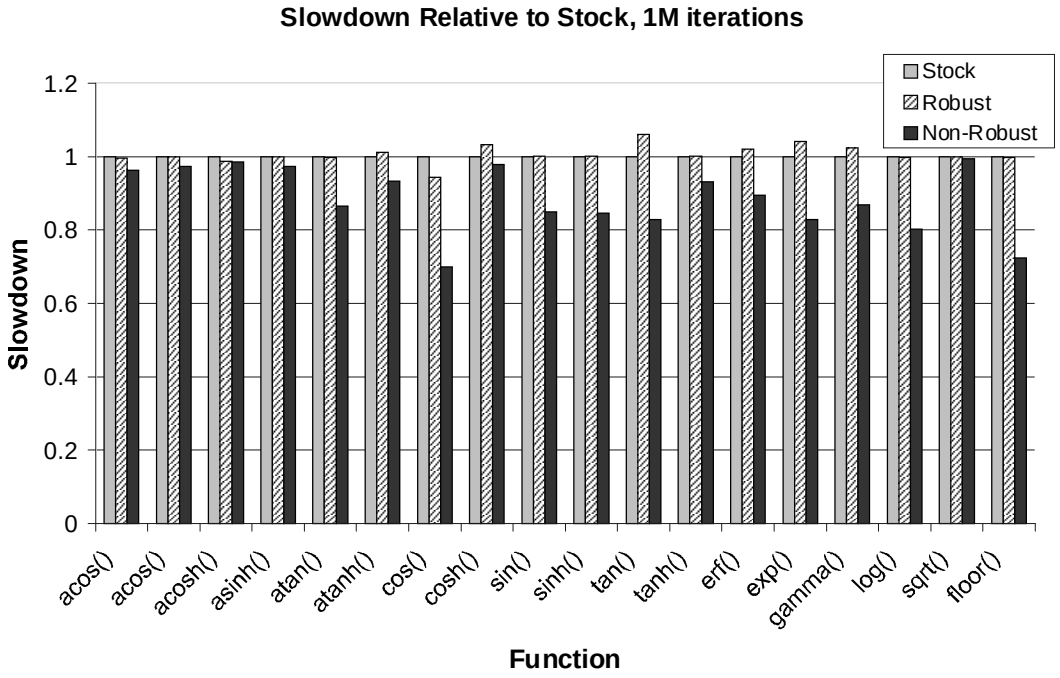


Figure 10. Performance of libm variants normalized to the unmodified source.

5.3 Results

Overall the cost associated with making a library robust was 14% when compared to the non-robust case. However, the standard math library was only 1% faster than the robust version. Complete results for the relative performance of each library function call tested are shown in Figure 10.

The robustness of each library was measured with the Ballista tool to determine the relative performance with respect to software robustness. Results are detailed in Table 3. The robust library had an average failure rate of 0.0%, while the stock and non-robust failure rates were 15.3% and 12.5% respectively. No restart or catastrophic failures were detected.

5.4 Analysis

The performance data taken on the libraries provide a surprise, in that the robust library is very close to the unmodified library in terms of speed performance. In fact, the performance difference was on average less than 1%.

It is important to realize that the unmodified library code does perform a fair amount of error checking already, to perform such functions as ensuring the input parameter is within the proper range of the function. This is the source of the performance penalty of the stock library versus the non-robust library. The lack of these checks can in some cases result in an incorrect answer being returned by the non-robust library. Although the stock library performs many checks, in most cases it forces a floating point exception to occur in hardware when an error condition is detected. This of course causes the OS to send a SIGFPE to the application. On the other hand, the robust library implementation sets `errno` and returns zero (or NaN if supported).

	Robust Library	Stock Library	Non-robust Library
acos	0.00%	54.55%	9.09%
acosh	0.00%	45.45%	18.18%
asinh	0.00%	0.00%	0.00%
atan	0.00%	0.00%	0.00%
atanh	0.00%	72.73%	36.36%
cos	0.00%	0.00%	0.00%
cosh	0.00%	0.00%	18.18%
erf	0.00%	0.00%	0.00%
exp	0.00%	9.09%	18.18%
floor	0.00%	0.00%	0.00%
gamma	0.00%	9.09%	45.45%
log	0.00%	36.36%	18.18%
sin	0.00%	0.00%	0.00%
sinh	0.00%	0.00%	18.18%
sqrt	0.00%	18.18%	18.18%
tanh	0.00%	0.00%	0.00%

Table 3. Robustness failure rate of math library variants.

The stock math library tested already performed most (but not all) of the error checking required to implement a truly robust system. Most of this checking was performed in order to ensure a correct answer. The added cost of setting an error code and checking its value upon completion is extremely small for most of the math functions. This is especially true considering that any penalty is amortized over the total running time of the user application, which likely spends the majority of its time processing in its own code and not in the library.

The non-robust library has all error checking removed from it. That the non-robust version has a lower measured failure rate seems on the surface to be a counter-intuitive result. This effect is caused by the fact that the stock library does a fair amount of error checking and purposefully generates signals in response to these errors. The non-robust library does no error checking. This means that the only signals which will be generated are those generated directly due to a hardware exception. Any input which yields only undefined or indeterminate results will simply return an answer which is wrong at best.

5.5 Summary of Math Library Robustness Results

This work represents the first attempt to quantify the performance cost of building a software system with robust error detection and handling. It can be argued based on the results presented here that the performance impact is small when compared with the large gains in software robustness, provided they have characteristics which are similar to the math library. For a cost of only a few percent on the actual function call, a robust system can be implemented using error return codes which are easy to diagnose and recover from, and leave the process in a defined state.

Another important conclusion we can reach from the results presented here is that in the case of the math library, the error checking performed by the existing code base to ensure correct functionality is almost sufficient to ensure robust operation. They paid most of the price of building a robust system without reaping the benefits. With the addition of a few more instructions, the library can be completely robust, at the average cost (in terms of performance) a few percent, or roughly 90 ns on average per function call.

This work was an important first step toward quantifying the performance cost of robustness. It however only hints at the result, as the math libraries are relatively specialized, and as such no general conclusions about the performance cost for building robust software systems can be drawn.

Although the results were suggestive, the next logical step was to look at a software system that was more complex. A higher complexity system tends to have more complex data types and memory structures. The complexity makes it more difficult to fix robustness problems, and possibly makes it more sensitive to performance issues.

We searched for a software system that was complex, stressed a different aspect of system architecture than the math libraries, and preferable was one thought to be already hardened. Finding a system that met these criteria would allow us to see if the result of little performance cost for added robustness. Equally important, a pre-hardened library would give us a data point to determine what types of robustness problems were perceived as most difficult or costly, providing a significant challenge for our assumptions and methodologies.

6 Hardening and Analysis of Safe, Fast I/O

The Safe Fast IO library developed by AT&T Research improves robustness by a factor of 3 to 10 over STDIO without sacrificing performance. This was largely accomplished by optimizing STDIO, and then adding features to make operation more safe and robust. Based on robustness testing results, we were able to improve the robustness of eight critical SFIO functions by another factor of 5. Thus, use of a robustness measurement tool has enabled quantifying and reducing robustness failure rates by a factor of up to 70 from standard I/O functions, with an average performance penalty of 1% as measured by the original SFIO benchmark scheme. Future processor architecture improvements will further improve checking speed, essentially eliminating performance as an obstacle to improving software robustness.

6.1 Introduction

SFIO was written over a decade ago in an attempt to address the issues of speed and safety (robustness), and while the authors of SFIO were able to demonstrate that it was a high performance library, at the time it was developed there was no method for quantifying robustness. They could make a case that the library was safer due to their design decisions, but there was no method available to quantify how much they had improved over STDIO. Furthermore, discussions with the developers of SFIO revealed that even they were concerned about the performance impact of increasing the amount of exception checking done by their code.

We saw the existence of SFIO as an opportunity to gain an initial understanding of how robust an Application Programing Interface (API) implementation might be made using good design techniques but no metrics for feedback, and what the actual performance penalty might be for further improving robustness beyond the point judged practical by SFIO developers. First, we used the Ballista tool to measure the robustness of SFIO to exceptional parameter values at the API level. This allowed us to quantify SFIO robustness and find that it was significantly more robust than STDIO, but still had room for improvement. Then we found some common types of robustness vulnerabilities in SFIO and hardened against them, further improving robustness. At first the improved SFIO did in fact have some performance problems; however, these were

largely remedied by optimizing for the common case and the result proved to be significantly more robust than the original SFIO with only a slight performance penalty.

The remainder of this chapter describes our efforts to identify and fix general robustness failures within the SFIO system and quantify the performance impact of the additional code added to harden the system against those failures. Additionally, we discuss the types of robustness failures that are still expensive to check for, and how near-term processor architecture enhancements for general purpose computing will also reduce the cost of improving robustness. First, the results of the initial robustness testing of SFIO are presented and discussed. Next, the robustness test results of the hardened SFIO are presented. The benchmark results are then discussed for the initial hardened version (unoptimized). We then discuss the results of benchmarking the SFIO version with the hardening code optimized.

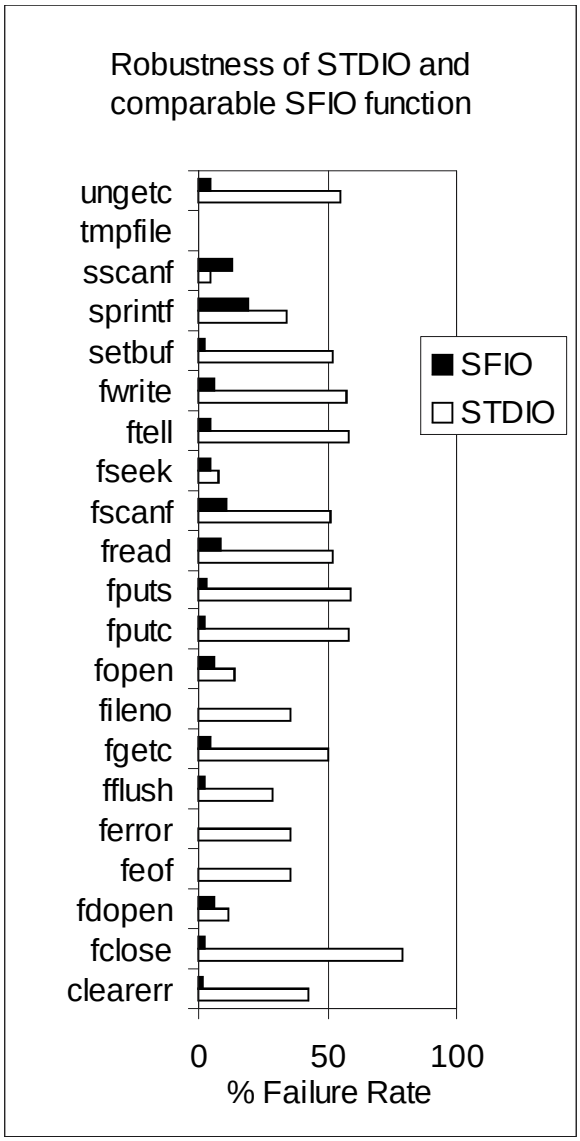


Figure 11. Robustness failure rates for SFIO, STDIO compared for 20 functions with direct functional equivalence as measured on the Linux test system. Failure rates on Digital Unix were lower for some SFIO functions and are addressed later in Section 5.2

6.2 Robustness testing of SFIO

We used the Ballista testing suite to measure the robustness of the 36 functions in the SFIO API. This allowed us to objectively evaluate the SFIO library in terms of exception handling robustness. To test SFIO we used existing data types for POSIX tests and created two custom Ballista test types capable of generating tests cases for the SFIO types Sfio_t and Void_t. These types fit directly

into the Ballista data type framework, and inherited much of their functionality from the generic pointer type. This made implementation a simple exercise, requiring only a few hours to implement the types, and to test the types themselves within the Ballista framework to ensure that they themselves were robust.

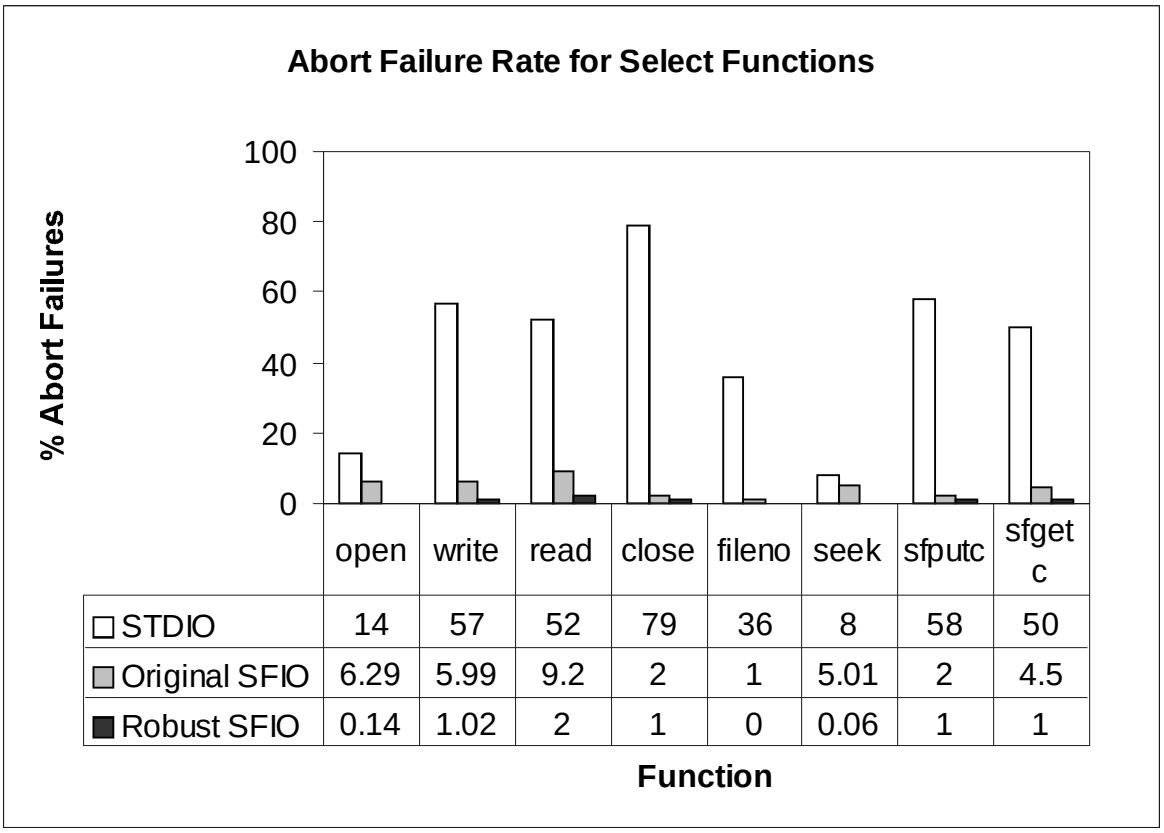
Our testing showed that while the robustness of SFIO is far better than STDIO (Figure 11), SFIO still suffers from a fair number of robustness failures in critical IO function such as write and read. Analysis of the testing data showed that there were three broad causes for many of the SFIO robustness failures. Specifically these were:

- Failure to ensure a file was valid
- Failure to ensure file modes and permissions were appropriate to the intended operation
- Failure to check buffers and data structures for size and accessibility

These problems were not a case of defective checking code in the software itself, but rather a lack of attempting to check for these types of exceptions.

Once identified, these potential causes of failures were addressed in a generic fashion across the eight most important IO functions in which they occurred: sfopen, sfwrite, sfread, sfclose, sffileno, sfseek, sfputc, and sfgetc (the "s" prefix indicates a "safe" version of the corresponding STDIO library call). For every function we were able to reuse the parameter validation code for each specific failure mode, thus reducing the cost of developing such checks to being linear with the number of parameter types, rather than the number of functions hardened using our techniques. For this first version of what we will call Robust SFIO functions, only ordinary attention was paid to performance – the emphasis was instead placed on reducing robustness failure rates. Figure 12 shows that the percent of Abort failures (*i.e.*, percent of test cases resulting in an abnormal task termination instead of an error code) were significantly reduced for the Robust SFIO software version.

File validity and permissions were relatively easily checked. A call to `fstat()` was sufficient to ensure that the file existed, and that our reference to it was currently valid. Once we determined that our reference was valid, a call to `fcntl()` was sufficient to obtain the flags and



	open	write	read	close	fileno	seek	sfputc	sfgetc
STDIO	14	57	52	79	36	8	58	50
Original SFIO	6.29	5.99	9.2	2	1	5.01	2	4.5
Robust SFIO	0.14	1.02	2	1	0	0.06	1	1

Figure 12. Abort Failure Rate for Select SFIO Functions under Linux

permissions associated with the file. These were checked against the intended operation, and errors were flagged and returned accordingly.

Checking for valid buffers and data structures was more difficult. While SFIO provides some information to ensure that buffers and data structures are valid, there is always the possibility of de-referencing an invalid memory pointer when performing those checks, or the checks simply failing to detect a potential problem. Because the POSIX standard gives no assurance that a task's state will be valid after a memory access fault, we validated memory prior to the function execution by striding (read then write) through the memory structure with a stride size of the memory page size for the architecture the code was executed on. This allowed us to catch exceptions during a validation stage before modifying the system state, eliminating issues of performing rollbacks or otherwise dealing with partial completion of functions in the event of an exception that resulted in an error code instead of successful completion of a function.

We used the mechanisms described in [lee83] to set up and perform signal handling on a per call basis. While this is more time consuming than setting up global handlers, it does ensure that the exact state of the program at the time of the signal is known. This reduces the complexity of the signal handlers, and makes the recovery from such exceptions easier to design and code.

Figure 12 shows the Abort failure rates for the 8 modified functions, both before and after treatment. The failures that remain in the modified functions represent cases where the data values passed into the functions have been corrupted in a manner that is difficult to check with data structure bounds checking, pointer checking, or other similar techniques. Overall the unmodified SFIO library had an average normalized Abort failure rate of 5.61%, based on uniformly weighting the per-function failure rates of 186389 test cases across 36 functions tested. The underlying operating system can sometimes affect robustness[fernsler99], and our testing showed that the normalized failure rates for SFIO running on Digital Unix were 2.86% for the 8 functions of interest. The Robust SFIO library had an average failure rate of 0.44% (Digital Unix) and 0.78% (Linux) across the 8 modified functions.

While even the Robust SFIO library does not achieve perfect robustness failure prevention, it is significantly better than both STDIO and the original SFIO. Additionally, it is possible that Robust SFIO could be improved even further by employing techniques for detecting invalid memory structures (e.g., using techniques from [wilken93][austin94][wilken97]). However, many of these techniques have a hefty performance penalty without their proposed architectural

Benchmark Name	Description	File Size	
		Linux	Alpha
Copyrw	Copies file with a succession of reads and writes	1000MB	2000MB
Getc	Reads file one byte at a time	250MB	2000MB
Putc	Writesfile one byte at a time	250MB	2000MB
Read	Reads file	1000MB	2000MB
Revrd	Reads file in reverse block order	1000MB	2000MB
Seekrw	Seeks to random file position, reads one block, and writes block to position 0 of same file until a number of bytes equal to the filesize has been seeked, read, and written	1000MB	2000MB
Write	Writes	1000MB	2000MB

Table 4. SFIO benchmark descriptions

support to identify “bad data” situations. Thus, further robustness improvements will become practical only when they are supported by future generations of microprocessor hardware.

6.3 Performance Results

Once the evaluation of SFIO had been completed and several key functions had been hardened, we measured the performance of the original and hardened versions and compared them to each other, and to STDIO. To measure the performance of the robust SFIO functions, we used the benchmarks (Table 4) as described by the authors of the original SFIO [korn91]. The results presented are the averages from 10 benchmark runs and are presented in Figures 13& 14 (execution time variance across runs was negligible). Each run consisted of a single complete execution of each benchmark. The benchmarks were run on two diverse architectures with different development ideologies and goals. The first test system had 333 MHz dual Pentium II processors, 128 MB RAM, and executed Redhat Linux version 6, with kernel 2.2.12smp and Gnu STDIO library version 2.1.2-11. The second system was an AlphaServer 4000 with two 600 MHz 21164 processors and 1GB of physical RAM, running Digital Unix 4.0D and libc version 425.

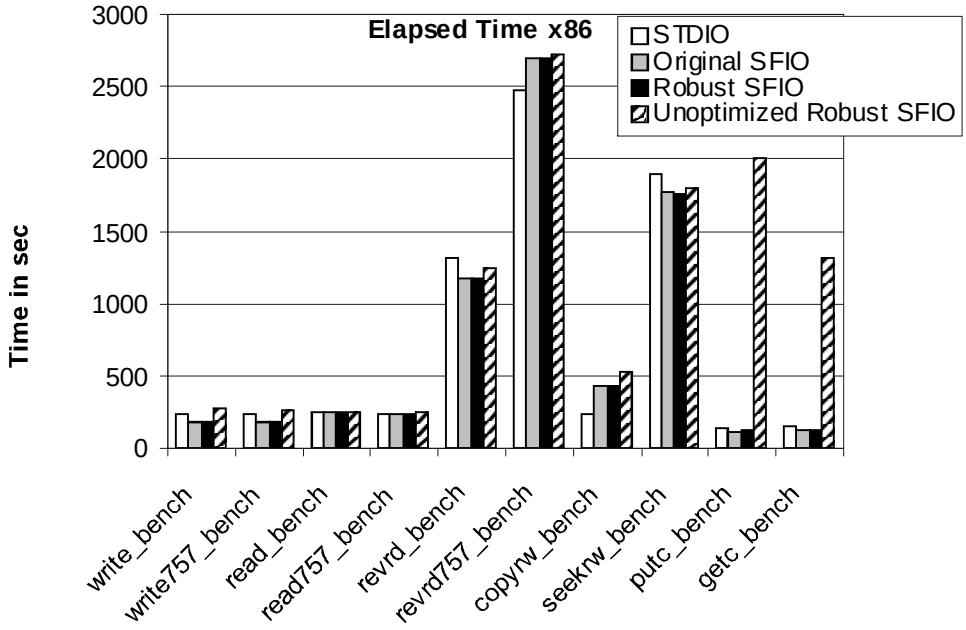


Figure 13. Elapsed time of benchmark running on x86 architecture

Table 4 describes the operations performed by each benchmark, with a block size of 8K. Benchmarks with a 757 suffix appended to the name used a block size of 757 bytes. The reason for the different transfer sizes is due to the difference in how the machines are configured. We chose sizes that were large enough to ensure the data was not being cached in main memory, and thus would have to be re-read from disk between each run. The Linux platform had to be run on smaller benchmarks than the AlphaServer to keep execution times reasonable.

The goal of using two different testing platforms was not to directly compare performance of the hardware in questions, but to present platforms whose OS developers have divergent philosophies and goals. Digital Unix is a proprietary operating system developed to provide maximum throughput, and is optimized for a small number of architecturally similar, advanced processors with fast IO hardware. Linux is an open source OS that runs on an very wide range of hardware platforms, from Intel x86 based workstations to the IBM System/390 mainframes. One side effect of targeting such a wide range of architectures for Linux is that some performance enhancements can't be included in the code base due to problems with cross platform compatibility. Further, it can be argued that although Linux is widely used in small scale server applications (and occasionally in larger scale ones), it is most commonly used as a workstation

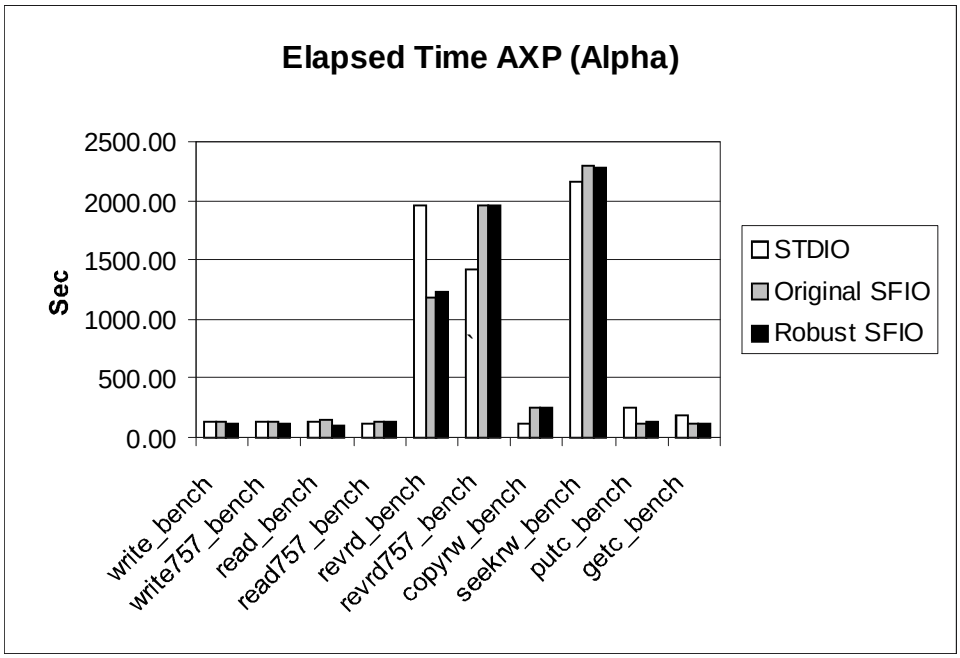


Figure 14. Elapsed time of benchmark running on the AXP (Alpha) architecture

OS and as such is optimized more for latency and less for raw throughput. Finally, commodity PC hardware is extremely cost sensitive and, even on the high-end system we used, sacrifices significant bandwidth potential to keep costs down. We hope that by satisfactorily showing that the cost of achieving a high degree of I/O robustness is low on these diverse systems, it is likely that similar techniques will work on other systems whose design points fall between these two extremes.

The block IO benchmarks perform IO on large files - 1000 MB on the Linux platform and 2000 MB on the AlphaServer. Byte IO benchmarks use a 256 MB file and 2000 MB on the Linux and Alpha systems respectively. The seek benchmarks performed 125,000(Linux) or 250,000(Alpha) seek + read + write operations, totaling 1000 MB or 2000 MB respectively. These are in some cases a few orders of magnitude greater than the original SFIO benchmarks published in 1990 because original sizes tended to result in files being entirely cached in memory buffers and completed too quickly for accurate measurement.

As might be expected, the performance penalty for our original implementation of the more robust SFIO was in some cases substantial. Execution times of the getc and putc benchmarks were especially long. Such a result tends to support the contention that past a certain point, robust exception handling simply costs too much in terms of performance to be worthwhile. SFIO seemed to find an optimal point where performance is improved or very close to STDIO, with large gains in robustness and exception handling ability.

Upon reflection however, we were determined to address the performance concerns and try to obtain the additional exception handling at as low a cost possible. One obvious place to look for performance optimization is getc and putc. Obviously, the overhead of the checks had a significant negative impact on performance. This is in part due to the structure of the code put in to handle exceptional conditions. Every time a byte is to be read or written, a signal handling context is set up, and the SFIO data structure is validated in terms of memory accessibility, file existence and setup. This demonstrates that robustness checks can adversely affect performance if applied without thought to actual usage. However, we were able to largely eliminated the speed penalty for performing these checks even on byte-wise I/O with getc and putc.

To speed up the robust versions of getc and putc we applied a variation of optimistic incremental specialization [pu95]. The idea behind this technique is that generated code should be optimized for the most likely, though not guaranteed system state. This is similar to the common programming adage – "Optimize for the common case."

We specialized the robust SFIO implementation for the case where the same file or buffer is used for IO many times before moving on to perform IO on a different file. This seems to be the most likely case when performing byte at a time IO. This was a simplifying design choice, and could have easily been designed and implemented using a different "common case", or some more complicated caching scheme.

To speed up getc and putc, we added caching of validated results to the SFIO data type pointer and SFIO buffer pointer data types. In any function that successfully validates one or both of the types, their values are cached for future comparison. Functions such as close that destroy types reset the cached values to their NULL value. During execution, the values of the parameters passed in for execution are compared with the cached values. Those functions with both types must match both cached values to pass the comparison. Similarly, functions with only one of the types must only match that corresponding cached value.

In the event the parameter(s) of interest match the cached values, all checks are bypassed. This includes skipping the construction of the exception-handling context for the function call. In

	elapsed			usr			sys		
Benchmark Name	Original	Robust	Relative Speed	Original	Robust	Relative Speed	Original	Robust	Relative Speed
write	175.30	175.40	1.00	0.43	0.50	0.86	24.88	24.85	1.00
write757	183.50	183.10	1.00	6.94	7.09	0.98	26.09	26.28	0.99
read	247.10	249.00	0.99	3.08	3.35	0.92	66.28	66.38	1.00
read757	236.20	240.10	0.98	2.15	2.20	0.97	99.15	99.23	1.00
revrd	1171.30	1174.50	1.00	0.98	1.19	0.83	48.42	49.03	0.99
revrd757	2696.40	2693.20	1.00	6.22	6.37	0.98	28.27	28.49	0.99
copyrw	428.80	427.00	1.00	3.98	7.27	0.55	93.63	96.02	0.98
seekrw	1763.20	1761.20	1.00	2.90	3.16	0.92	88.84	87.75	1.01
putc	111.10	128.30	0.87	66.22	94.07	0.70	10.82	10.62	1.02
getc	124.70	127.70	0.98	66.39	99.35	0.67	24.24	18.69	1.30

Table 5. Usr, sys, and elapsed time data for original and hardened SFIO (Intel Architecture)

this way the overhead is reduced to a single branch in the case that the same structure is used more than once in succession, after the initial penalty for the checks have been paid.

Table 5 gives complete user and system level performance information for the original SFIO and the final robust SFIO with incremental specialization as described above. As with the previous performance data, the values are the average of 10 complete runs. Total process time is broken down into the user and system components as measured by libc function call time().

6.4 Analysis

It should be no surprise that the performance data clearly show that the common operations selected for additional hardening are IO bound. This is typical in a modern super-scalar machine where the CPU can be IO bound even on simple memory requests. Although there is much work being done to improve this [griffin00], it seems unlikely that the IO speed will catch up to the speed of the processing unit in the near to mid-term future. Thus, hardening of IO functions can be accomplished basically for free on latency-based computational tasks.

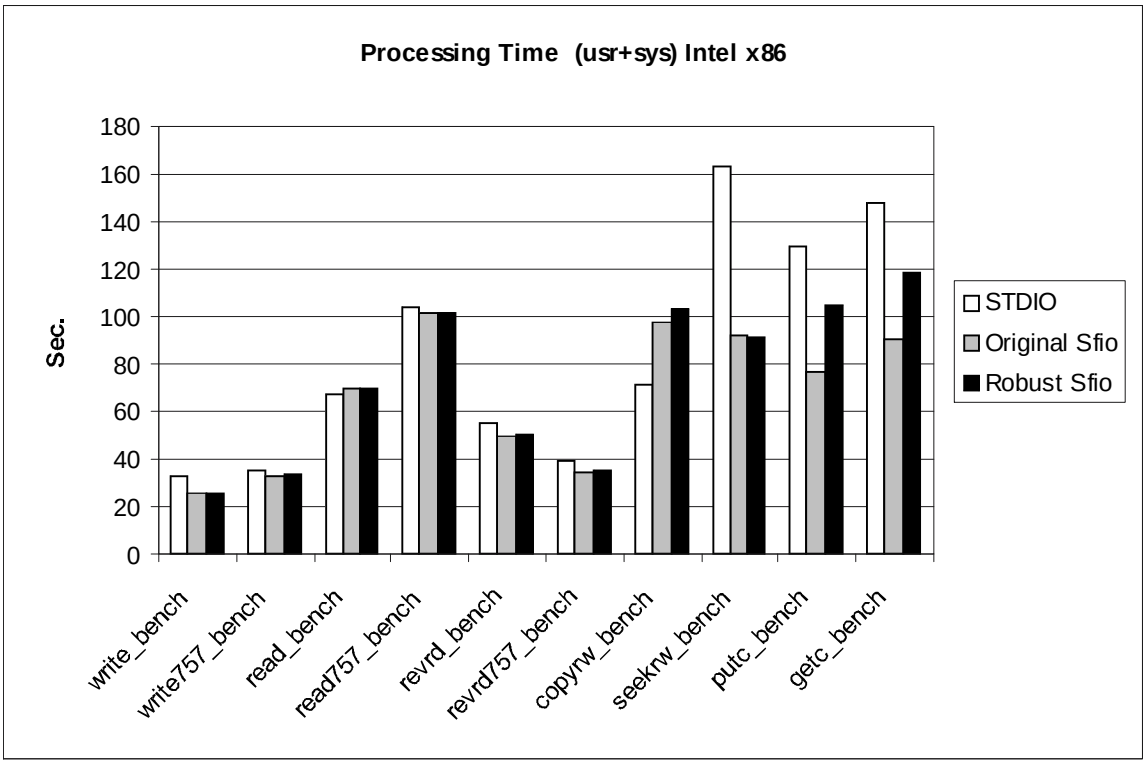


Figure 15. Total processing time for benchmark on x86 architecture

In particular, although file I/O operations are state rich and require much error checking and handling, the latency added for increasing the ability of the functions to handle exceptions and behave in a robust manner is mostly hidden by the latency of the overall operations. Block file operations suffer an execution time penalty of only a few percent compared with the less robust implementations.

Though the elapsed time for the benchmarks to run to completion tell part of the story, it isn't enough to simply look at this data. Elapsed time hides the intricacies of what is going on inside the OS and hardware that can be critical to the performance of a system, especially for throughput-limited operating environments. After all, the time spent during IO wait can be used to perform other useful work in a multi-tasking system.

Figures 15 and 16 show the total time spent performing computation (*i.e.*, usr+sys time but not IO wait time) of the hardened SFIO is in some cases less than that of STDIO, and except for the 757 block size and copy benchmarks is within 2% of STDIO on Linux. Both SFIO implementations used much less actual processing time than did STDIO on the AlphaServer

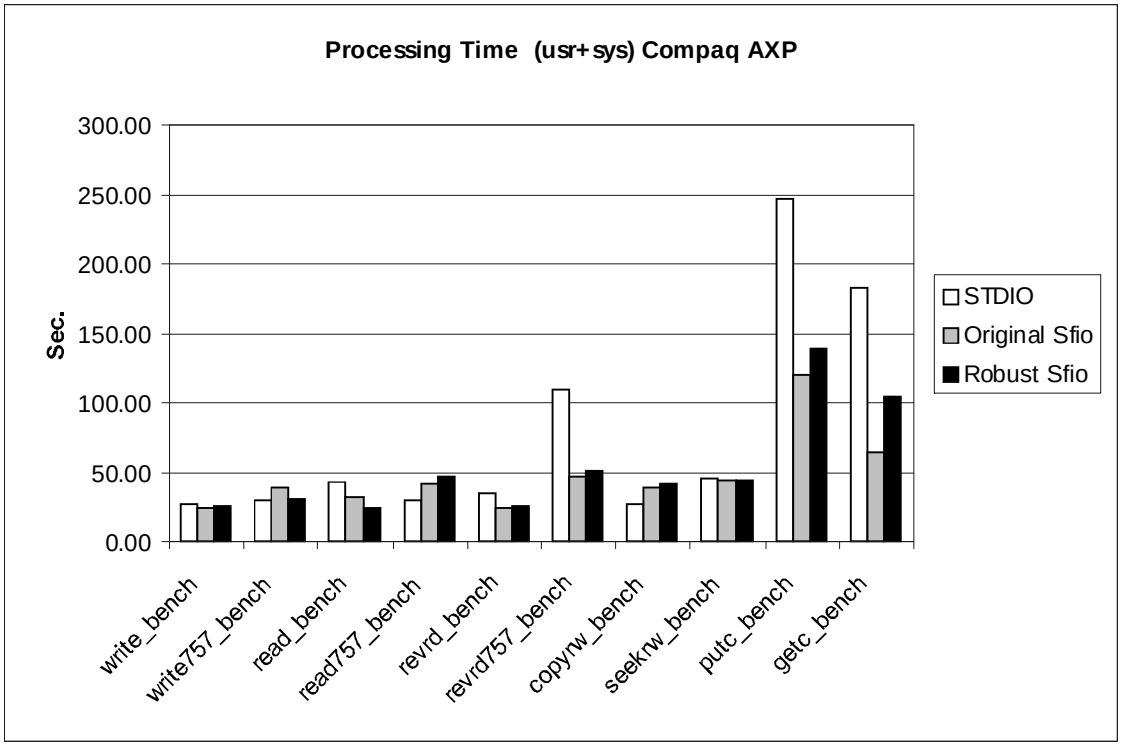


Figure 16. Total processing time for benchmark on AXP(Alpha) architecture

platform (except seekrw, copyrw, revrd757 and read757) though the elapsed time tended to be close to or slower than STDIO. This seems to indicate that the Digital Unix STDIO libraries perform a fair amount of processing to optimize the disk transfers, and is born out by the fact that the benchmarks spend less time in IO wait when using the STDIO libraries. From this one can infer that disk transfer scheduling optimizations consume far more CPU cycles than would increased robustness checks.

One benchmark that has a significant performance shortfall in both SFIO implementations is copyrw. It seems likely that the STDIO libraries are able to avoid a memory copy that the SFIO libraries are performing. This seems to be supported by the data in that nearly all of the extra time manifests as “system” time, typically where large block memory copies would be measured.

The processing time penalty paid by robust SFIO compared to original SFIO consists largely of occasional exception handling context setup and parameter checks. In addition to the penalty from constructing exception handling contexts that occurs when the parameters require validation, there is a mandatory penalty that represents the check to determine if the validation must be done. However, we expect the processing cost for such checks to diminish significantly in the near future.

Of the penalties incurred, the penalty for determining if validation should occur is likely to be almost completely negated by improved hardware branch prediction that will be available in new processors soon, though fragmenting block size with a branch can still affect performance[rotenberg96]. Actually achieving this requires creating a compiler that can structure exception-checking code sequences in a way that will help the CPU predict that exceptions will not occur, but there is no technical reason this should be difficult to accomplish.

Processors that use a trace cache[rotenberg96], such as the Intel Pentium 4 processor, will lessen the cost of additional checks by allowing the unit to fetch past branches that may otherwise throttle fetch bandwidth. While more advanced checking and caching techniques might degrade performance in ways the trace cache can not help (such as multi branch direction traces), we anticipate techniques to solve such problems will be incorporated in processors in the near future. These include such techniques as completion time multiple branch prediction

[rakvic 00] and block caches[black99]. In general it seems reasonable to expect that exception checking branches, which are easily predictable as taking the non-exceptional code path, will become increasingly efficient as processor hardware incorporates more predictive execution capabilities.

Thus, robust SFIO libraries can achieve dramatically reduced robustness vulnerabilities compared to STDIO and even original SFIO implementations. For latency-bound applications the performance impact of providing extra robustness is minimal. For throughput-bound applications there can be a moderate increase in CPU time used to perform extra checking for some routines, but this can be minimized by caching check results. Furthermore, it is likely that as CPUs increase their use of concurrency and branch prediction that any speed penalties for performing exception checking will decrease dramatically over time.

6.5 Summary

We used the Ballista robustness testing tool to find and address robustness problems in the Safe/Fast I/O library (SFIO), and found that we were able to improve the robustness of the code by an average factor of 5.9 across the treated functions, despite the fact that SFIO already improves robustness over STDIO robustness by an order of magnitude. The achieved robustness level was approximately 0% to 2% robustness failure rates, compared to 0% to 79% failure rates for STDIO. We have found that the remaining failures generally involve incorrect or corrupt data within otherwise valid data structures, but speculate that such failures might be dealt better with during interface design.

Contrary to commonly held opinion, very robust software need not come at the price of reduced performance. The data show that the performance penalty for providing thorough exception handling and error handling tends to be low in terms of elapsed time, and similarly small in terms of processing overhead. Robust SFIO was only ~0%-15%(avg. of 2%) slower than ordinary SFIO, while providing better robustness. Furthermore, near-term architectural improvements in processors will tend to reduce the costs of providing robust exception handling by exploiting the fact that exception checks can be readily predicted and executed concurrently with mainstream computations.

7 Hardening and Analysis of Operating System Internals

The theory and techniques developed for building high performance, robust systems were successfully applied to IO bound (SFIO) and CPU bound (libm) software systems. To establish the generality of the approach, a third domain is identified and treated .

This chapter examines the application of techniques developed for implementing high performance robust systems on operating system services. The robustness of any software system can be argued to be dependant on the robustness of the underlying system. Thus any system, despite the best effort and exacting attention to detail by its creators, may be doomed to low robustness due to a non-robust underlying operating system.

Speed, in terms of latency and throughput, is often a critical deciding factor when evaluating operating systems. If the operating system can not be made robust without sacrificing performance, then spending the time and money to harden application level code may be of questionable virtue. Commonly used elements in the Linux API's memory and process synchronization modules were selected for study. Each method was hardened to a 0% robustness failure rate as measured by Ballista. The resulting modules suffered a performance loss of only 5% using a lightweight synthetic application benchmark, when compared to the non-robust libraries.

7.1 Robustness testing of Linux

In order to determine the best system areas to target for robustness hardening, the Ballista robustness benchmarking tool was run against the Linux API. The results for the complete tests are found in Appendix A.

Although the system calls in the Linux kernel have improved in robustness over the last several releases, areas traditionally thought of as "too hard" to address have been left in an unhardened state, and are among the most critical, including process synchronization and memory/buffer manipulation.

For example, as illustrated in Figure 17 the semaphore module (in Linux implemented outside the kernel in a thread library) has a relatively poor response to exceptional conditions, with an

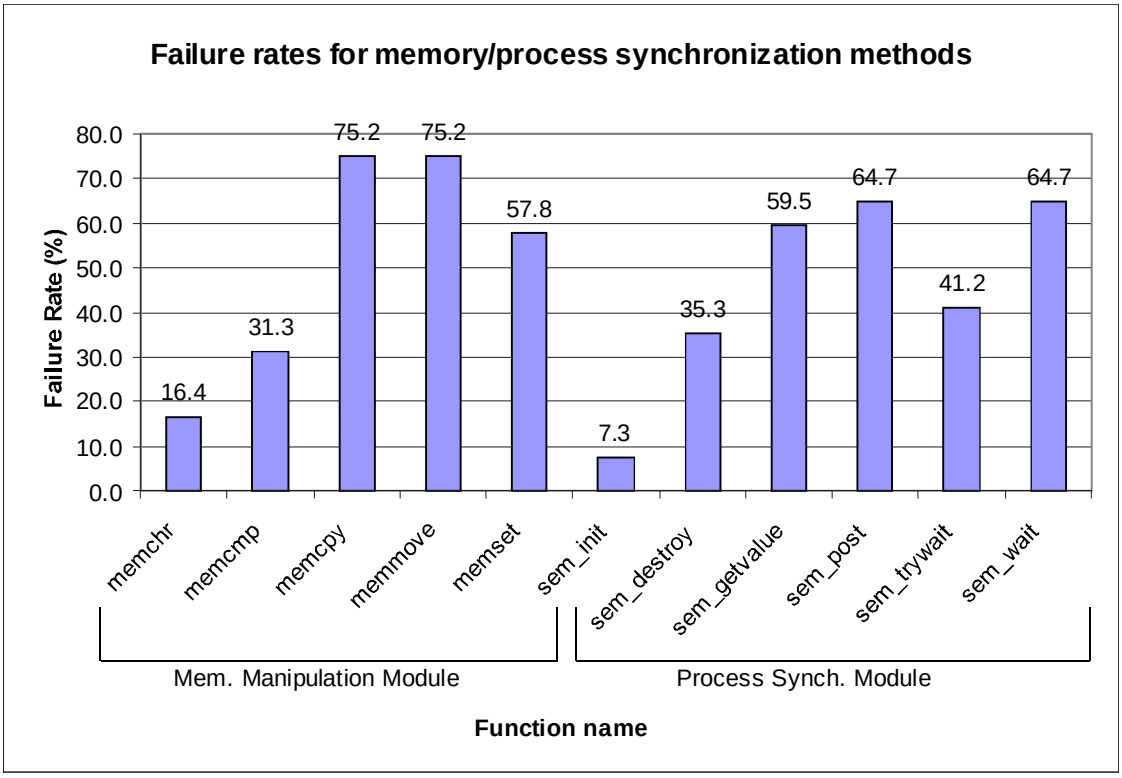


Figure 17. Initial failure rates for memory manipulation and process synchronization methods

average failure rate of 45.4%, normalized per function call. The memory manipulation functions have an average failure rate of 51.2%, normalized per function call.

These "too hard" areas are interesting, because execution speed is critical, and they are commonly used in modern software systems to support multiple threads of control. Additionally, problems in these areas can have dire consequences, thus making addressing the problems a potentially high impact activity.

The testing and benchmark results in this chapter were executed on a dual Intel Pentium III 600Mhz processor machine with 512MB RAM running Linux 2.2.16 with glibc 6.13.01.

7.2 Hardening of select Linux API calls

In earlier efforts, our ability to address these most problematic areas of robustness (structure and memory validation) were limited. This was partially due to the fact that the software systems looked at previously did not suffer many failures of this type. The math libraries have no failures in these areas due to a lack of using complex structures and buffers. SFIO solved a fair number

of these problems internally, maintaining speed by removing overall inefficiencies within the standard IO systems. Thus the number of such failures encountered was small. In contrast, process synchronization functions and (perhaps obviously) memory manipulation functions such as `memcpy()` and `memmove()` suffer large numbers of failures in these troublesome areas.

7.2.1 Failure of previous techniques

The techniques developed earlier in this work fell short in their ability to address the class of robustness failures prevalent in these modules. Checks that satisfied the requirements of largely bufferless (libm) systems of those already partially hardened (sfio) were not sufficient. Figure 18 contains results from the initial hardening of select Linux API calls using techniques developed during the hardening of the math and SFIO libraries.

While the earlier methods were adequate to harden the simpler functions, several of the memory functions still exhibited failures. These were instances where the memory locations being manipulated were valid areas the process had read/write permissions to, but overwriting them simply cause the process to fail catastrophically as a result of data or process corruption.

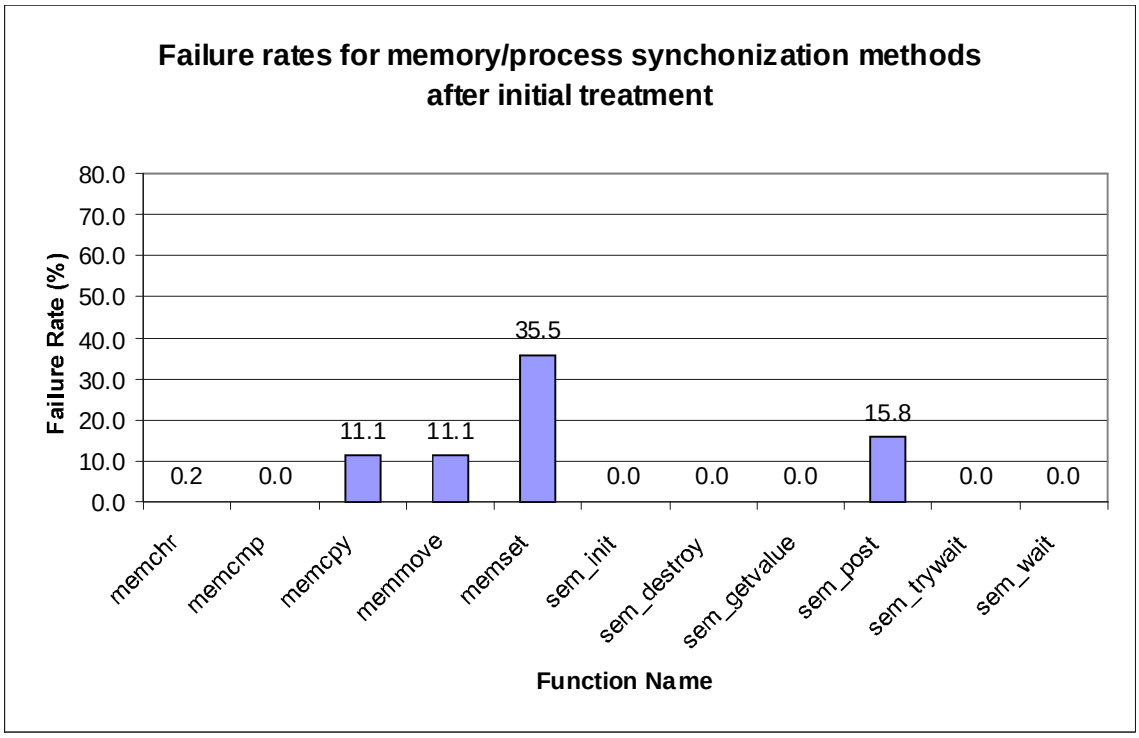


Figure 18. Failure rate of memory and process synchronization functions after initial treatment

Unfortunately, these failures are exceptionally difficult to deal with due to the limitations in the way systems tend to organize and allocate memory. Many systems (even the widely used Windows OS) have operating modes that developers can use to force allocation routines to put memory in separate, isolated memory pages. While this aids in detecting faults during testing, it does nothing to improve robustness at run time issue on deployed (in service) systems running in "normal" mode.

There is no easy, fast way that a process can tell if it can write to a memory area safely when the system is deployed and in use running in a normal operational mode. Prior work [wilkin93] proposes methods of building special purpose hardware into micro architectures and compiler tools to aid in the determination that a structure or memory location has been allocated properly and correctly initialized. Unfortunately, their approach results in a substantial speed penalty if the special purpose hardware is not available. Recognizing that the micro architecture community is only recently beginning to seriously address issues of fault tolerance and reliability, it is unlikely that hardware to enhance robustness will be included in the short term, if ever.

We believe that this represents a fundamental limitation on the ability to design a fast, robust system, imposed by the design of the API. While it is possible to backfill a design to have elements to allow for the safe testing of structures and buffers to determine if they are safe to use, it would likely result in a substantial speed penalty, and would almost certainly occur in a somewhat ad-hoc manner.

7.2.2 Approach for enhanced memory robustness checks

The approach we used to overcome this situation was to create a version of malloc (Figure 19) that added an application transparent 8 byte tag to each dynamic structure created. This tag allowed us to store information regarding its size and validity. Building it onto the structure facilitates fast lookups and

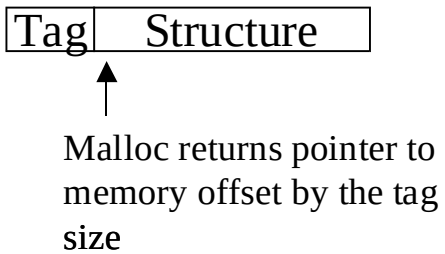


Figure 19. Altered malloc helps address fundamental robustness limitation

verification since the static offset eliminates the need for complex indexing calculations.

In essence, we are adding to the internal state that malloc already keeps for allocated memory. While malloc does keep size information (as well as other information such as lists of memory blocks, etc), it is hidden from the process. We provide a mechanism whereby the state of the memory block is exposed to the robustness checking code. When a process requests for a memory location to be validated, the added data can be used to determine that the memory has been properly allocated and sized. The possibility for other, context aware checks is also possible by embedding other information into the tag.

Figure 20 shows the results of hardening with the advanced memory checking and validation techniques. All functions are completely hardened against all robustness failures detectable by the current suite of robustness tests.

7.3 Performance analysis of hardened code

It is not enough to make a system robust. A system must also be fast enough to meet its performance goals while still being robust. Of course since the performance goals of

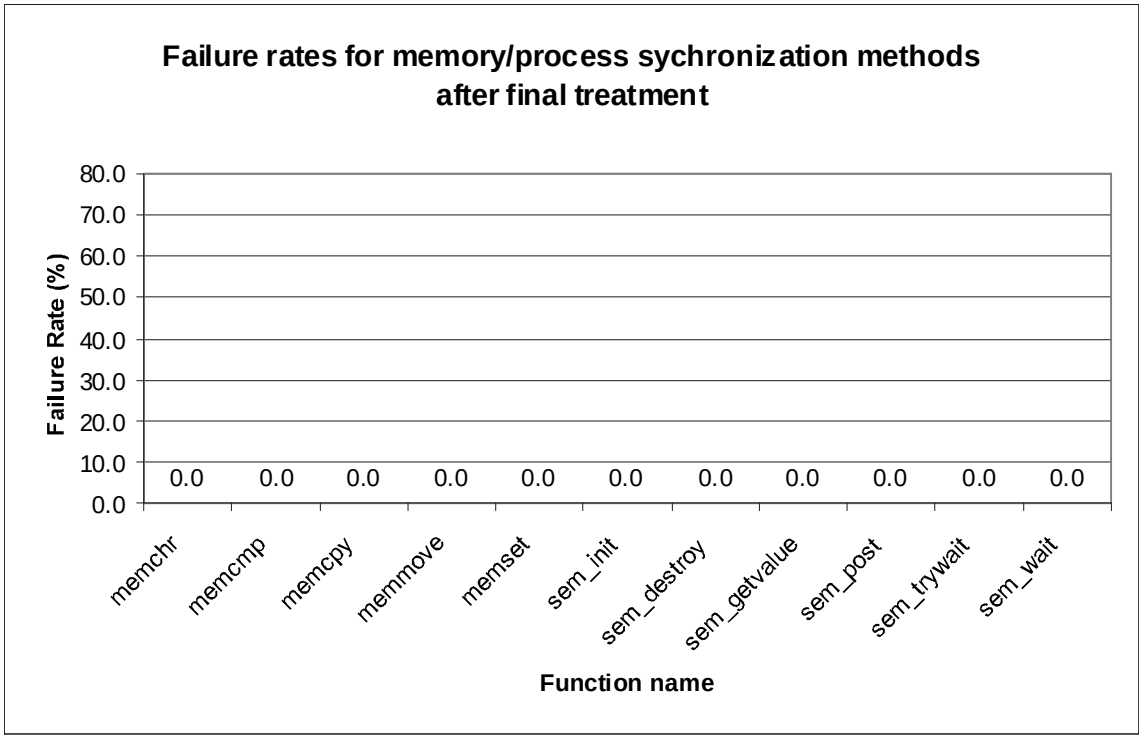


Figure 20. Failure rate of memory and process synchronization functions after final treatment

non-embedded applications are "as fast as possible", the overhead must be kept to an absolute minimum.

We developed a variation of a technique known as optimistic incremental specialization as described in [pu95] to enhance the performance of SFIO. This simple approach worked well, as IO is routinely done in large blocks, and the caching of a single validation was cheap and effective. The performance impact of hardening SFIO was less than 1% on average.

Process synchronization and memory manipulation functions tend to have a much different use profile. Thus, our approach from SFIO was adapted to function in an environment where many resources were being used in conjunction with each other and in varied sequences, thus complicating matters significantly.

7.3.1 The Robustness Check Cache

To address these issues a traditional cache style approach was used. This allows the construction of arbitrarily sized structures to track any number of resources that have been validated by robustness checks, or invalidated by various operations (such as `free()`, or `sem_destroy()`).

The robustness check cache is implemented entirely in software and required no special hardware support. The operation of the check cache is a close parallel to traditional

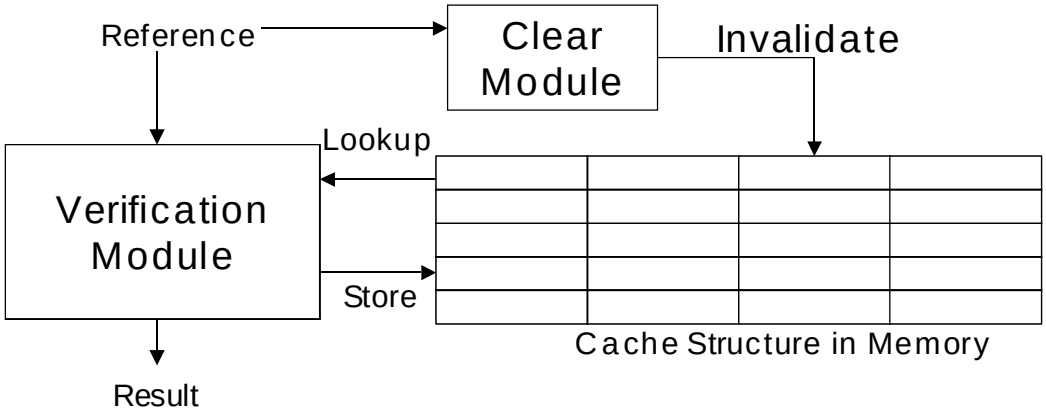
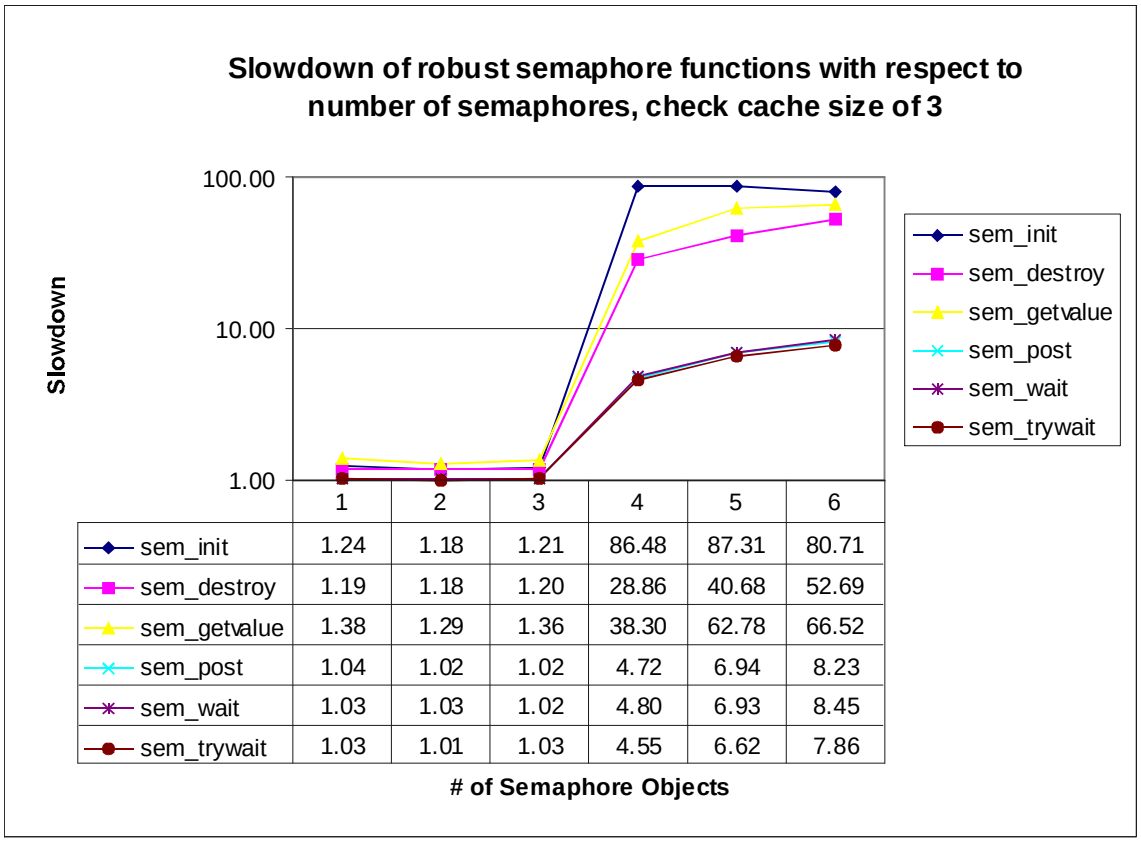


Figure 21. The software implemented robustness check cache

data/instruction caches, and is illustrated in figure 21. When a robustness check is successfully performed on a structure by the validation module, its address is placed in the cache, indexed by its low bits. Before a complete set of robustness checks are performed the cache it checked. A successful hit bypasses the checks. Any function that destroys or critically modifies a structure or object calls the clear module which causes the appropriate cache entry (if any) to be invalidated.

7.3.2 Iterative Benchmark

In order to present the worst case application slowdown, the performance of the robust system functions is measured using simple iterative benchmarks. By this we mean that the performance of the methods was measured during repetitive calls to the method being benchmarked, with no useful calculations being performed between calls. In other words, we asked "what was the slowdown of function A, given that we call it 20,000,000 times in a row?" For the simple



	1	2	3	4	5	6
sem_init	1.24	1.18	1.21	86.48	87.31	80.71
sem_destroy	1.19	1.18	1.20	28.86	40.68	52.69
sem_getvalue	1.38	1.29	1.36	38.30	62.78	66.52
sem_post	1.04	1.02	1.02	4.72	6.94	8.23
sem_wait	1.03	1.03	1.02	4.80	6.93	8.45
sem_trywait	1.03	1.01	1.03	4.55	6.62	7.86

Figure 22. Iterative performance slowdown of robust process synchronization functions

iterative benchmarks we use a very small cache to demonstrate that a smaller, more optimized cache can further improve performance.

Performance results show three of the treated function performing poorly when compared to the non-robust function (Figure 22). The functions include `sem_init` – 24% slowdown, `sem_destroy` – 19% slowdown, and `sem_getvalue` – 38% slowdown.

These functions represent only a few hardware instructions (get_value is just an address calculation and a read), to which we needed to add a read, an integer operation, a compare, and a branch. Although the overhead seems high, we will show later in this work that it represents only a few cycles of actual CPU time. Additionally, given that the original code was small in relation to the code added to make it robust, one might think that the measured overhead is smaller than it should be. This effect is indicative of how our checks are being hidden by the existing microarchitecture, and will be even further reduced in future chip generations such as the Intel Pentium 4. A more complete treatment of this expectation is presented in Section 7.4.

Figure 23 shows the performance of the robust memory functions. Even for small buffer sizes the overhead is 3% or less, with the exception of memset (8%) and memchr (19%). Both memset

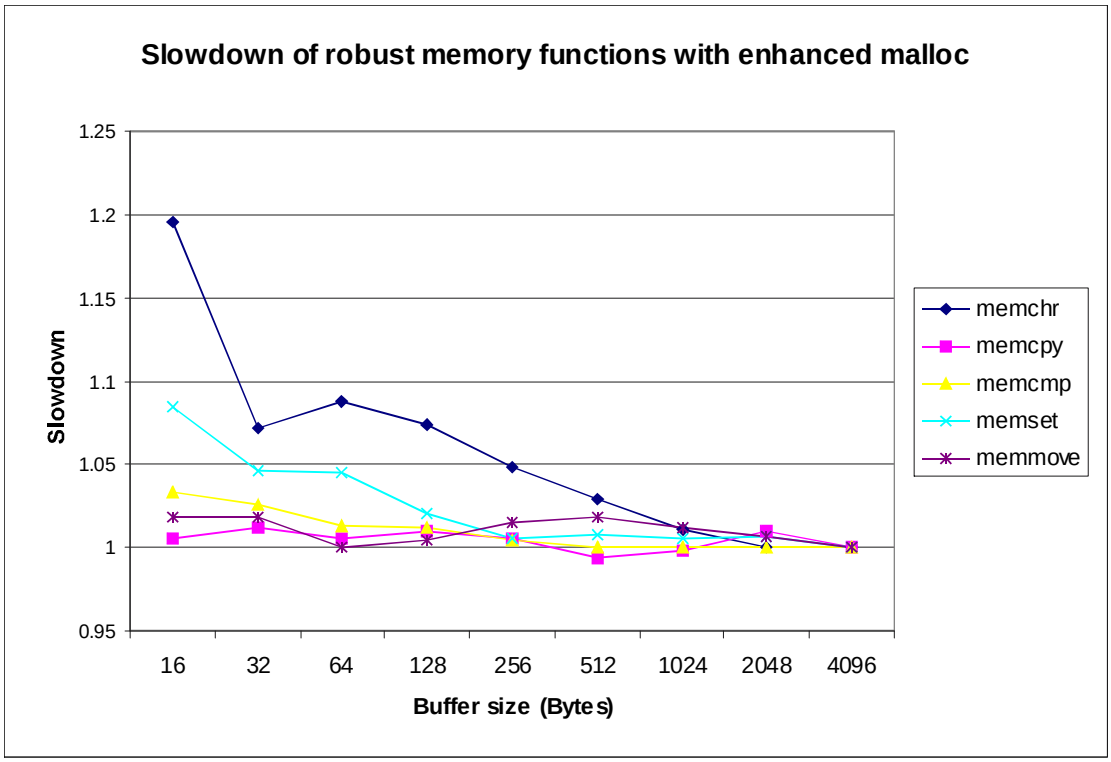


Figure 23. Iterative performance slowdown of robust memory functions

and memchr are both heavily optimizable, especially for small buffer sizes. Obviously, as buffer lengths increase, the overhead is reduced.

7.3.3 Lightweight Synthetic Application Benchmark

The iterative benchmarks are useful in illustrating a worst case performance bound for individual process synchronization functions. Unlike memory functions which often perform a complete task in and of themselves, the semaphore functions are tools used to facilitate coordination. As such the functions are used in specific sequences.

```
Create semaphores

Create empty queue
Do
      Foreach semaphore
            Obtain lock

      Enqueue integer value

      Dequeue integer value

      Foreach semaphore
            Release lock
```

Figure 24. Pseudo-code for lightweight synthetic benchmark

To obtain a clear picture of how performance is impacted in a manner consistent with use; a lightweight synthetic benchmark was created. This benchmark performs no significant computation, rather its purpose is only to call sequences of process synchronization functions that are consistent with normal use. Figure 24 contains the psuedo-code for the benchmark. In brief it simply creates semaphores, and then obtains and releases locks, with the small intermediate computation of the enqueing and dequeuing of an integer value.

The purpose of this benchmark is to avoid burying the overhead of making the functions robust in the cost of a complex computation. It illustrates the overhead of process synchronization as a whole rather than isolated by function call, and presents the most pessimistic operational scenario.

Figure 25 shows the results of the synthetic application benchmark with a small cache size of 3. On average, slowdown is 4%. This approaches the performance impact of `sem_wait` and `sem_post` as tested in isolation. This is to be expected, as the functions with the worst performance penalty are typically only called once in the course of an applications (i.e. `sem_init` and `sem_destroy`). `Sem_wait` and `sem_post` however are the primary

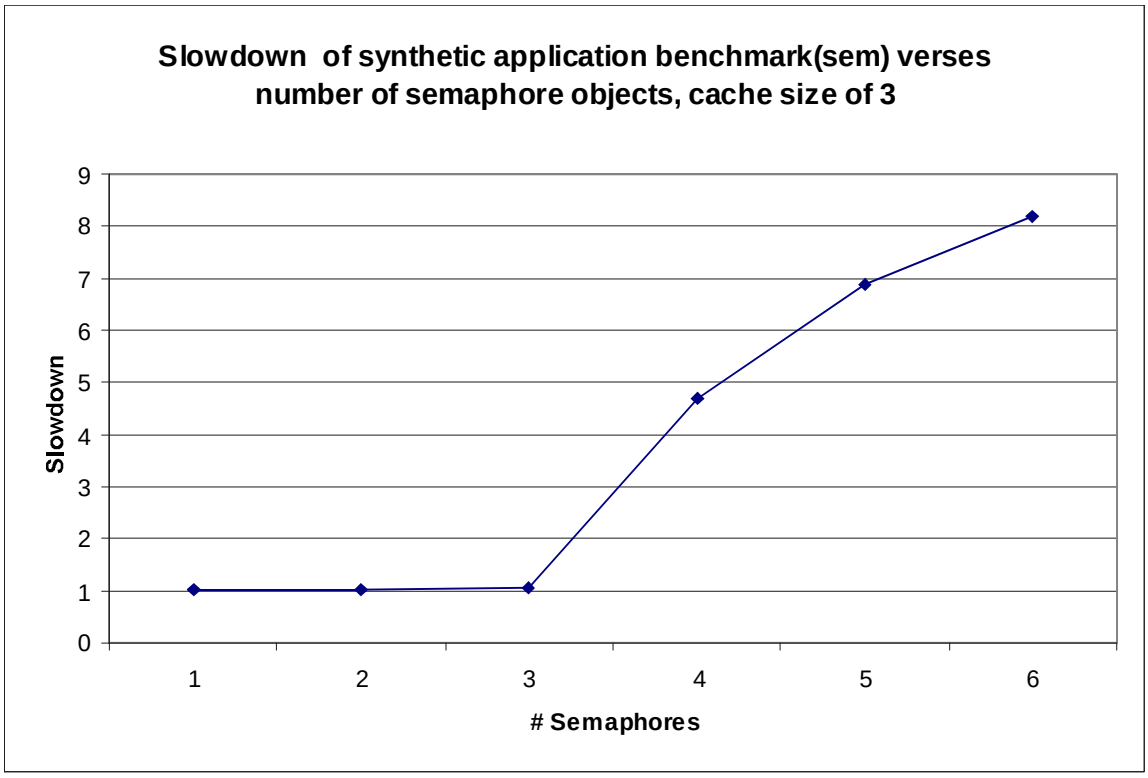


Figure 25. Slowdown of synthetic benchmark using robust process synchronization functions

functions used to test and set semaphores for process control purposes, and thus dominate the performance for the synthetic application.

A second experiment was run to look at the scalability of the approach for a system that requires checks done on large numbers of objects, and thus a larger cache. The increase in cache size increases slightly the cost of managing the cache by about 1%, with a wider range of performance as the cache fills and an increasing number of conflict misses occur.

Performance of the code using the cache for robustness checks is quite good, with a slowdown of on average 5% if less than 50 addresses are involved (Figure 26). Performance tapers off to a penalty of 18% in typical cache manner as the number of addresses increases causing conflict misses. As expected, when the number of addresses involved exceeds the cache capacity, performance drops off rapidly as capacity misses force ever increasing numbers of checks to be performed. Indexing is accomplished by doing address calculation explicitly, rather than allowing the compiler to generate the array index operation.

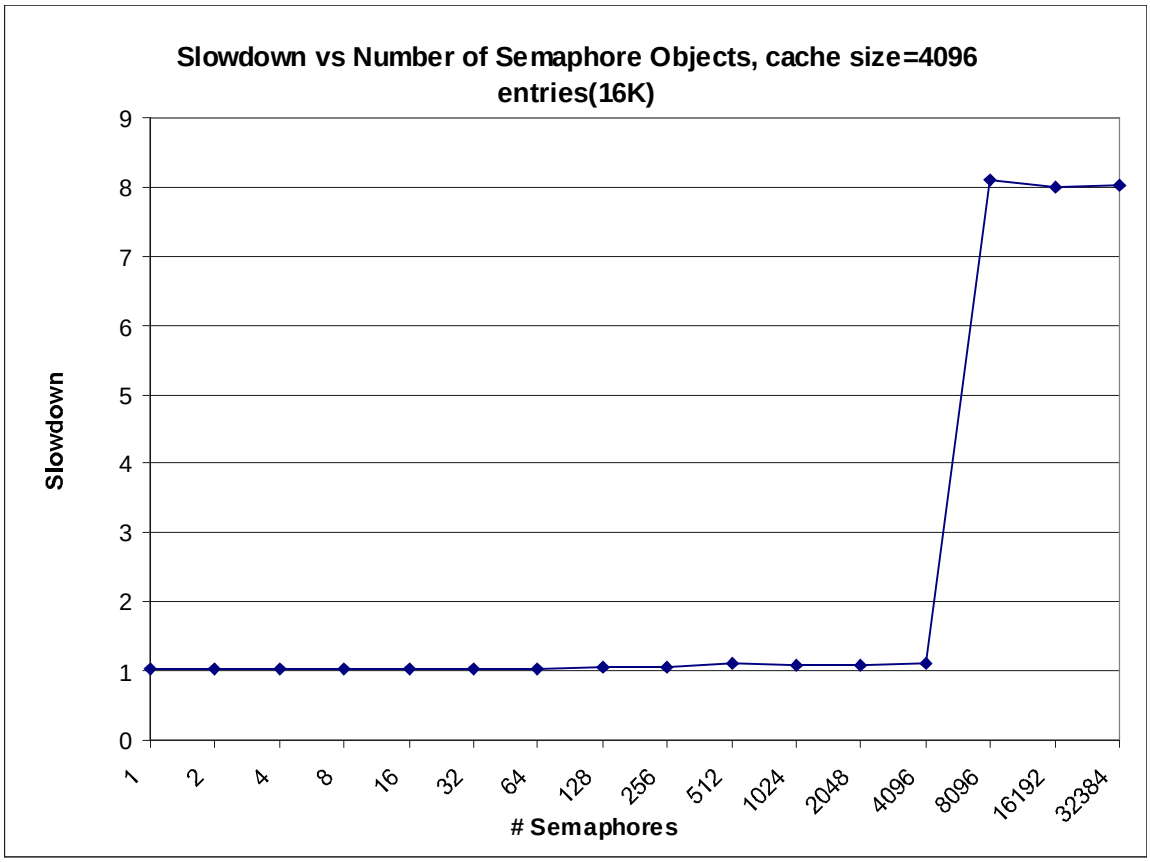


Figure 26. Slowdown of synthetic benchmark using robust process synchronization functions with large cache size

7.4 Conclusion

This data shows that a range of speed critical OS services/functions can be enhanced to be extremely robust with a conservatively attainable speed penalty of 5% for our light weight synthetic application benchmark. Iterative benchmarks show worst case slowdowns of 3-37%.

While it is not clear what an "average" case is, the synthetic benchmarks show that even the lightest weight application penalties approach the lower worst case bound. A software system that performs any non-trivial computation will likely see a near zero overhead.

The actual overhead (in nanoseconds) was determined, and can be found in figures 27 and 28. Overall the average absolute overhead was 9 nanoseconds for the process synchronization functions and 8 nanoseconds for the memory functions using a small cache size. Small cache synthetic application overhead was an average of 20 ns (for a full cache). For a large cache, the

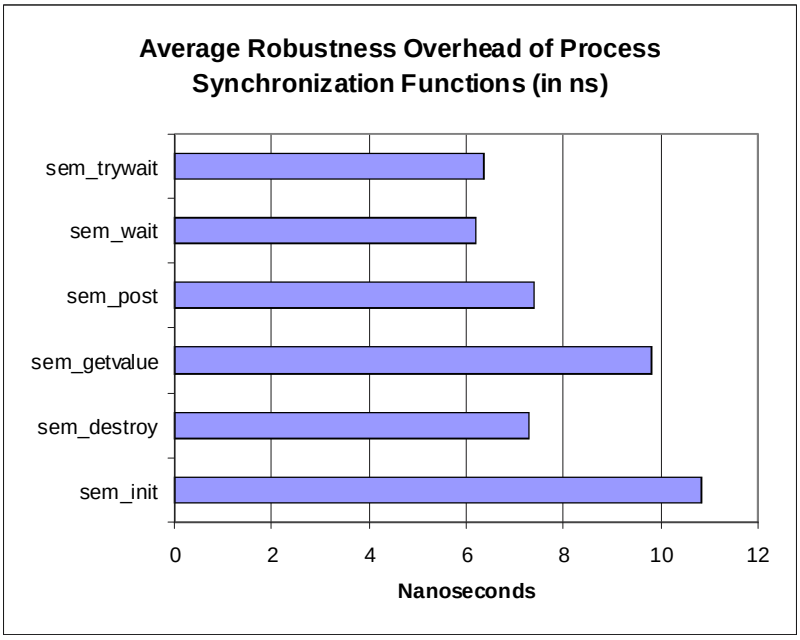


Figure 27. Absolute robustness overhead for process synchronization functions in nanoseconds

overhead increases to an average of 50 ns for a 1/4 full cache, increasing to 107ns for a cache at capacity. Note that the measured overhead for the synthetic benchmark actually includes the overhead of a pair of calls, one to `sem_wait()`, and one to `sem_post()`.

This overhead is on the order of about 10 cycles per protected call, and is representative of not only the index calculation and the branch resolution, but also the wasted fetch bandwidth. The microprocessor used on the test platform is incapable of fetching past branches, thus even correctly predicting the branch induces some penalty.

Advances in architecture such as the block cache[black99], multiple branch prediction[racvik00] and branch predication will effectively reduce the overhead to near zero. The robustness checks will be performed completely in parallel with useful computation, and predicated out. Fetch bandwidth will be preserved by the block cache and the multiple branch predictors. Thus only code with the highest degree of parallelism that utilizes 100% of the hardware resources will likely have any drop off in performance. This level of parallelism is seldom seen, and usually occurs only in tightly optimized loops of computational algorithms. Of

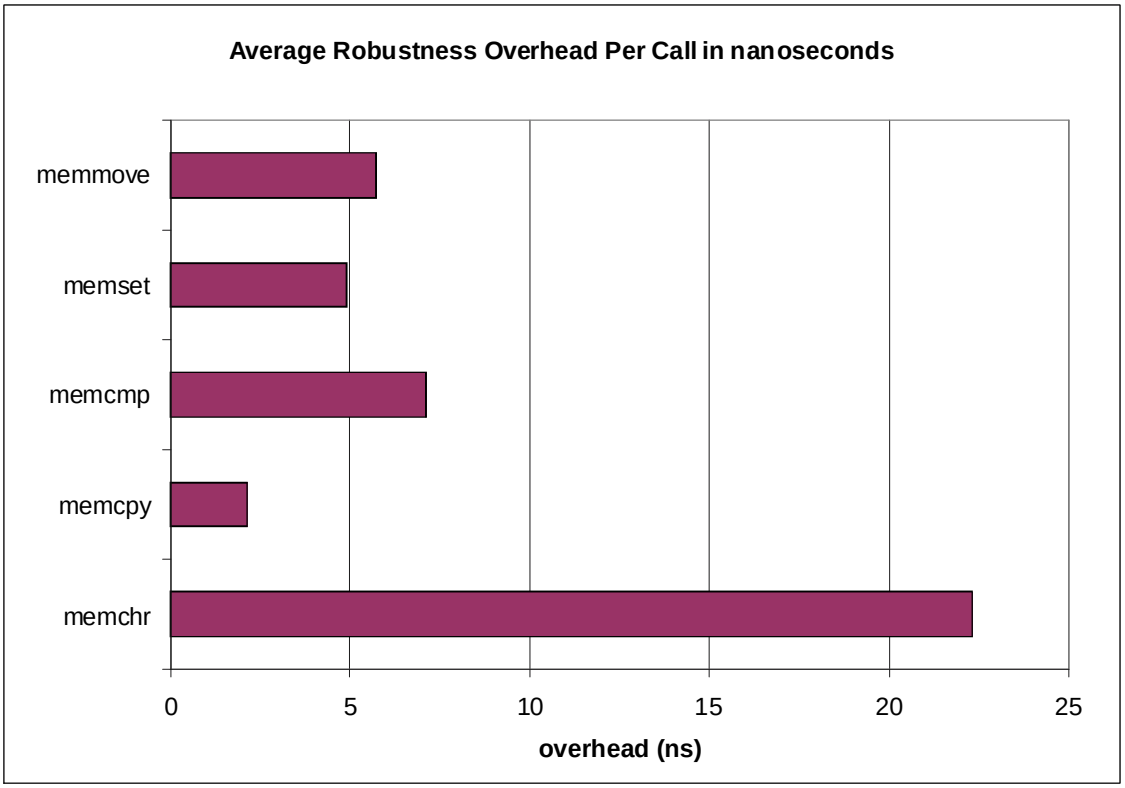


Figure 28. Absolute robustness overhead for memory functions in nanoseconds

course code sections such as those only need robustness checks upon method entry, and perhaps not even then if the method is guaranteed to receive known valid data.

8 Understanding Robustness

This chapter examines how well experienced developers understand the exception handling characteristics of their code. We look at the results from the testing of a DOD simulation framework known as HLA-RTI, developed in C++ with the explicit goal of having no undefined/generic exceptions possible. We then look at a series of Java components from the IBM component library. Data was collected on how the developers thought their code would respond to exceptional conditions, and contrasted with the robustness as measured by Ballista. The results indicate that industrial developers cannot predict the exception handling characteristics of their software systems, and may not have the ability to write robust software systems without better training in the area of exception handling, and exceptional conditions.

Robustness is not a concept addressed in many programming, or even software engineering classes[maxion98]. Too often the idea of testing software is linked to the idea that a system has been successfully tested when you can be reasonably sure it provides the correct output for normal input. Unfortunately, this philosophy overlooks the entire class of failures resultant from exception inputs or conditions, and exception detection and handling code tends to be the least tested and least well understood parts of the entire software system[christian95].

While up to two-thirds of all system crashes can be traced to improperly handled exceptional conditions[christian95], the reason such failures occur is not certain. Several possibilities exist, including the classics of “Too hard” to check for, “Too slow” to be robust, “That could never happen”, “That was from a third party application”, and one of any number of the litany of excuses.

Simple human error is yet another possibility. The system developer/designer may have intended to handle exceptions, and simply made a mistake. Almost all errors fall into one of two categories - errors of commission and errors of omission [swain83]. Thus the exception checking and handling code is either designed or implemented incorrectly (commission), or simply omitted (omission).

In his thorough treatment of this topic, Maxion posits that most exception handling failures in his test groups were errors of omission due to simple lack of knowledge and exposure to

exceptions, exceptional conditions, and exception handling[maxion98]. Maxion provided material to the groups with information on exceptions, exception conditions, and a mnemonic to jump-start their thinking on the topic, and help them to remember exception checking. Maxion was able to show significant improvement in the exception handling characteristics of the treatment group software when compared to the control group. This process is known as priming.

Although it clearly demonstrates that ordinary students do not understand robustness and exception handling, the obvious question with regard to Maxion's work is how well professional developers understand robustness, and the exception handling characteristics of their code. This is an important issue to address, because before we can succeed in helping developers create robust software systems, we need a better insight into why robust systems are not being built today.

This chapter examines how well experienced developers understand the exception handling characteristics of their code. We look at the results from the testing of a DOD simulation framework known as HLA-RTI, developed in C++ with the explicit goal of having no undefined/generic exceptions possible. We then look at a series of Java components written by various corporate development groups within IBM Research. Data was collected on how the developers thought their code would respond to exceptional conditions, and contrasted with the robustness as measured by Ballista.

8.1 The DOD High Level Architecture Run Time Infrastructure

The DOD High Level Architecture Run Time Infrastructure (HLA RTI) is a standard architecture for distributed simulation systems. It was developed by the US Department of Defense to facilitate model and simulation reuse and interoperability. Simulations run across a disparate network, and may include disparate components from a variety of vendors. For this reason the framework was desired to be completely robust[DOD98]. The HLA was adopted as the Facility for Distributed Simulation Systems 1.0 by the Object Management Group (OMG) in November 1998, and was approved as an open standard through the Institute of Electrical and Electronic Engineers (IEEE) - IEEE Standard 1516 - in September 2000.

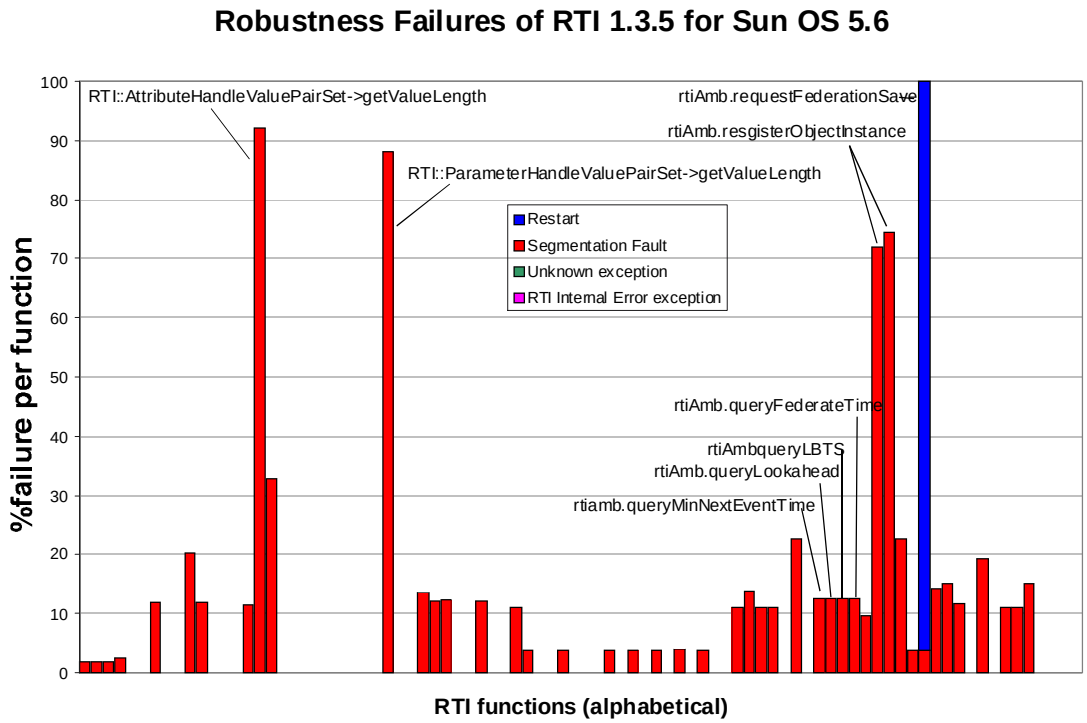


Figure 29. Robustness failure rate for RTI 1.3.5 under SunOS 5.6. Overall Average failure rate = 10.0% from [fernsler99].

As robustness is an explicit design goal, it can be argued that the developers had every reason to identify and handle all possible exceptional conditions. Thus any remaining exception handling failures must be the result of human error, rather than a design decision. Without access to the source code and extensive review it as unlikely that any determination can be made as to if the errors were of commission or omission. However, assuming that the standard practice of code reviews and adequate testing were accomplished, it is likely that the majority of the failures occurred due to errors of omission.

The RTI was extensively tested across a range of versions and operating systems[fernsler99]. Figure 29 and Figure 30 give the resultant failure rates of RTI version 1.3.5 on Solaris and Digital Unix respectively. 86 RTI functions were tested, and roughly half had failures. Though the failure rates are low when compared to most operating systems, there were several functions with large failure rates.

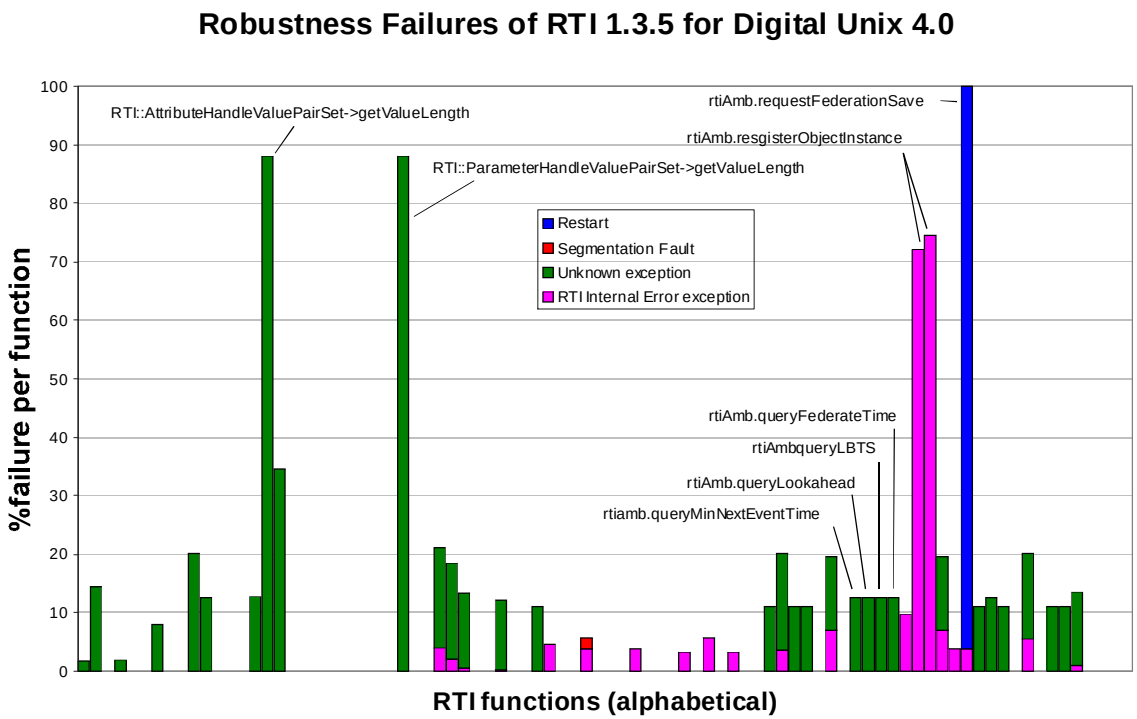


Figure 30. Robustness failure rate for RTI 1.3.5 under Digital Unix 4.0. Overall Average failure rate = 10.1% from [fernsler99].

Failures were diverse, ranging from problems in the mutex locking algorithms to infinite loops, or unwind crashes in the exception handling code. Given the broad failures, and the assumption that these were all due to errors of omission, it seems likely that there were at least several classes of exceptional conditions the designers/developers of the RTI did not think about or anticipate.

8.2 Commercial Java Objects

While the RTI results show that although the specification mandated robustness the developers didn't quite deliver, it fails to help address the question of how well the developers understood the characteristics of their code. Since the spec required robust operation, we assume the developers thought the system was robust. This is not necessarily the case however, and is mostly speculation.

To better address this issue, three production Java components were selected, and their developers reported on the expected robustness of the component. The components were then

feof()		Unknown	Not Robust					Semi-Robust				Robust		N/A
				Silent (No indication of Error)	Hindering - Unknown	Hindering - Wrong	Abort	Unhelpful Exception	Exception, leaves system in unknown state	Exception - leaves system in known, but unrecoverable state	Exception - Leaves system in known, recoverable state			
Computational/Concurrency														
	Div by 0													
	Out of Domain													
	Overflow/Underflow													
	MT Unsafe Response													
Hardware														
	Not Enough Disk													
	Resource Unreachable													
	Corrupt Memory			X										
	Resource Exhaustion													
I/O - File														
	File DNE						X							
	File Permissions Wrong													
	File Corrupt													
	File Moved/deleted													
	Invalid Filename						X							
	File Already Exists													
	File Locked													
Library Function (Shared Object)														
	Library N /A													
	Incorrect Version													
	Incorrect Parameters													
Data Input														
	Empty Data File													
	Incorrect Delimiter													
	Data Invalid			X										
Return Values and ARGS														
	Data Values Invalid													
	Wrong # of Arguments													
	Wrong Type of Arguments													
External														
	Wrong Command Line													
	Wrong Response to Prompt													
	No Response to Prompt													
	Workflow Overload													
Null Ptr/Memory														
	Null Ptr						X							
	Invlaid Ptr						X							
	Points to Invalid Data			X										
	Insufficient Memory													
	Allocation Error													
	Buffer Overflow													

Figure 31. Sample report form for feof()

tested with a Java version of Ballista written specifically for this purpose. The results were then be compared with their expected robustness.

8.2.1 Self Report Format

In order to allow a development group to report on their component's expected response to exceptional conditions, a taxonomy of failures was first developed. This taxonomy borrows heavily from that developed by Maxion in [maxion98]. Only minor changes were required, in order to better fit the object and safety models that are inherent to Java. The major categories were retained, and the order was maintained to preserve the original mnemonic structure, "CHILDREN". While this was not essential for this specific study, it was done for continuity, and out of respect for the original work.

Figure 31 is an example of the form used to report the expected robustness of a method within a component. This specific example is for feof(), and represents its true response as measured in earlier Ballista work. Categories are listed in rows, with expected response to exceptional conditions reported in the columns by checking the appropriate box. Any row without a check in a column is by default assumed to mean inapplicable (N/A).

8.2.2 Self Report Data

Once the form was finalized, it was built into a Lotus Notes database, and placed in service by the IBM Center for Software Engineering Research in Yorktown Heights, NY. The data was collected by IBM Research over the period of several weeks.

Tables 6, 7, and 8 contain the expected response of the components, rated by the development teams.

Condition	Response
Div by 0	NR
Out of domain	R
Overflow/Underflow	R
Data invalid	R
Data values invalid	R
Wrong num of args	R
Wrong type of args	R
Null pointer	R
Pointer to invalid data	R
Insufficient memory	R

Table 6. Expected robustness response for Component C

Method	Condition	Resonse
B1	Overflow/Underflow	R
	Data invalid	SR
	Data values invalid	SR
B2	Overflow/Underflow	R
	Data invalid	SR
	Data values invalid	SR
B3	Overflow/Underflow	S
	Data invalid	S
B4	Overflow/Underflow	S
	Data invalid	S

Table 7. Expected robustness response for component B

Method	Condition	Response
A1	Overflow/Underflow	S
	Data invalid	R
A2	Overflow/Underflow	S
	Data Input	R
A3	Overflow/Underflow	S
	Data Input	R
A4	Overflow/Underflow	S
	Data Input	R

Table 8. Expected robustness response for component A

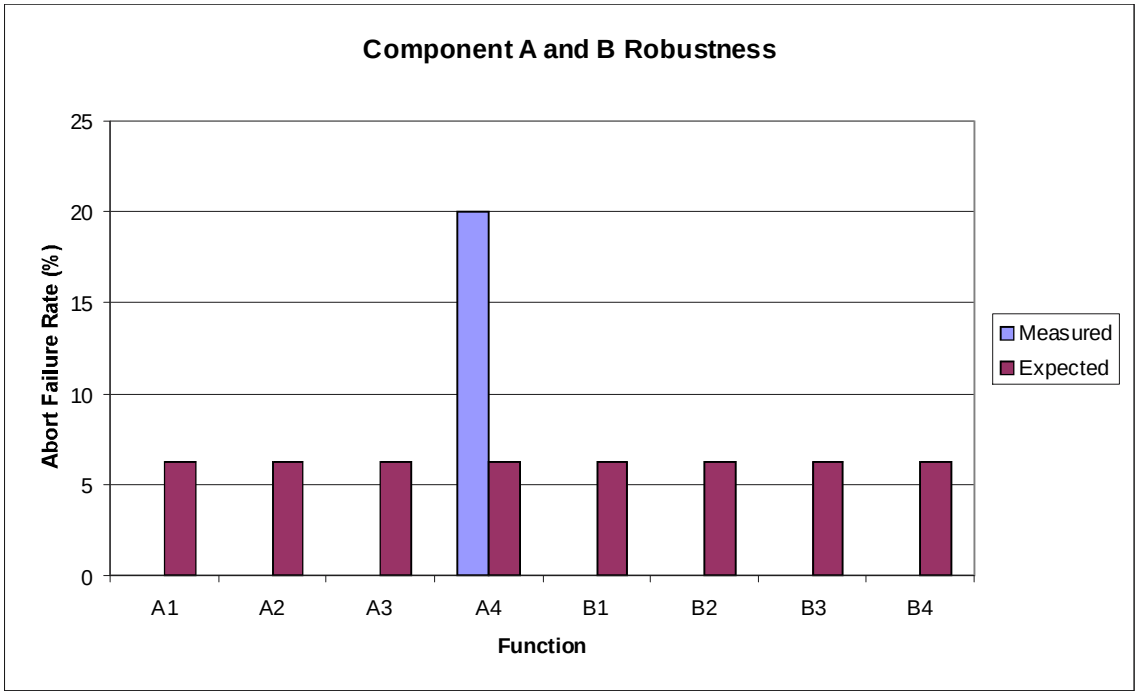


Figure 32. Robustness failure rates for components A & B

Component C consists of 38 methods, all of which had the same expected robustness. In the response column, R stands for a robust response, SR for semi robust, NR for not robust, and S for silent failure (A sub-category of not robust). Semi-robust means that the response is some error code or language supported exception that leaves the program in an indeterminate state.

8.2.3 Test Results

Three components comprising 46 discrete methods were rated by the development teams and tested using Ballista. The components, labeled A, B and C were written by teams 1, 2 and 3 respectively.

Results for testing components A and B can be found in Figure 32. Overall average failure rates were 5% and 0% respectively. One method suffers from an abort failure due to invalid data (memory reference), and some conditions marked as semi-robust actually could not occur due to language constraints placed on the testing system that made it impossible to create corrupt data for certain base level data types(both teams anticipated divide by zero failures).

Figure 33 contains the test results for object constructors of component C, and Figure 34 contains the test results for all other component C methods. Overall average failure rate was

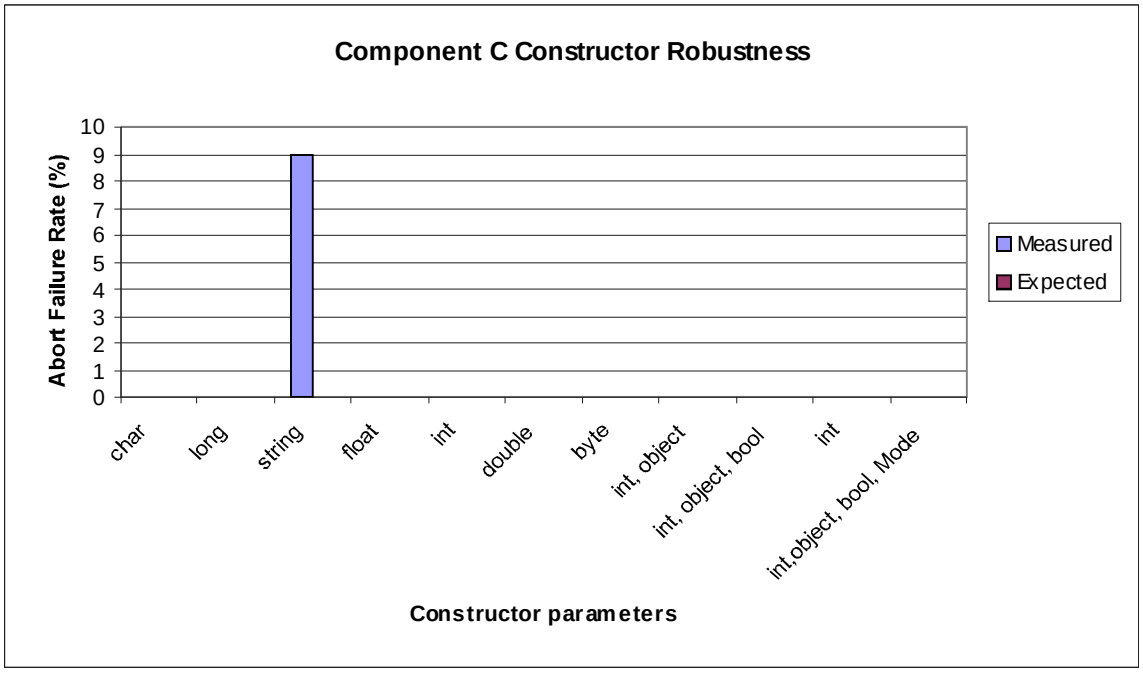


Figure 33. Component C Constructor abort failure rates by parameter

10.2%. The methods are separated into two classifications, those that do calculation, and those that merely return object data in a specific form or data type.

8.3 Analysis

The RTI development team delivered fairly solid code, with an overall failure rate of approximately 10% for the most recent versions examined. The Solaris version suffered from a high rate of abort failures, apparently due to an underlying problem with the C++ language exception system under Solaris. The Digital Unix version suffered a similar failure profile, converting nearly all of the abort failures to "Unknown exception" failures.

It is likely that the RTI team thought they had handled all possible exceptions correctly. This conclusion is drawn from anecdotal evidence developed from reading the specification and conversations with project engineers. When our testing results were presented, the failures were fixed, and suitable tests were added to the RTI regression test suite.

Teams 1 and 2 closely estimated how their system would respond to exceptional conditions. With the exception of a single failed pointer check instance, their expected robustness matched the measured robustness of the systems. This excludes conditions that could not be generated due to language constraints to check the presence of failures as a result of invalid base data types.

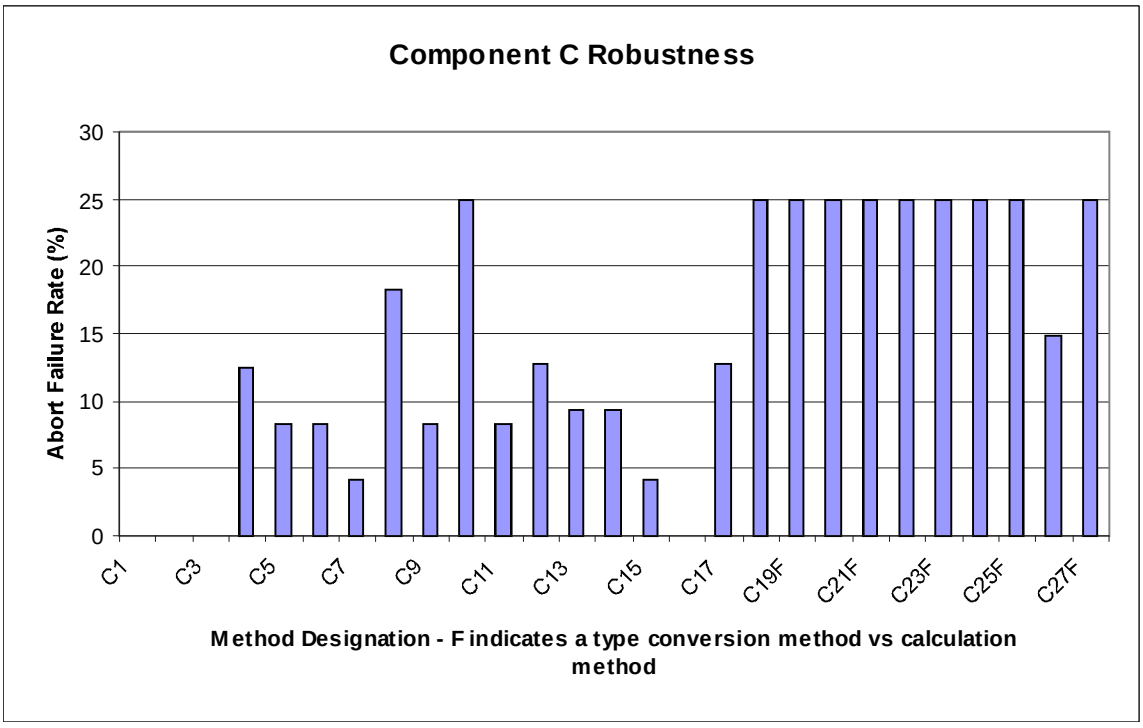


Figure 34. Component C Abort Failure rate

As is evident from the test data, component C suffered abort failures in roughly 60% of its methods. Team 3 indicated that the only failures would be resultant from divide by zero, and that all other exceptional conditions would be handled correctly. In fact, they suffered from several failures common in most software, including memory reference/data corruption issues, and failing to handle legal, but degenerate data conditions. The most prevalent exceptional conditions not handled correctly were caused by values at the extreme end of their legal ranges.

Although not normally detectable using the Ballista tool, a number of silent errors were detected within component C. These were detected when resultant values from method calls that returned without error caused a subsequent robustness failure during test cleanup. Consider the following example:

- An attempt to allocate memory failed, but returned a non-null value without an error condition being indicated.
- The system attempts to free the memory for use.
- An abort failure is the result.

Beyond the obvious observation that we can make about the robustness of the memory release method, we can also conclude that the allocation method suffered a silent failure. In essence it did not flag an error, and allowed it to propagate through the system where it could cause damage later during program execution. In this example, it manifested as an abort failure. But potentially, it could have been much worse if it simply caused the system to to the wrong thing - say deploy the airbags when no crash had occurred while the vehicle was traveling at speed along an interstate highway.

Figure 35 contains the silent failure rate for component C. The input conditions that induce the silent failures were examined for each case. The failures were all caused by conditions which caused abort failures in other methods. Silent failures are usually indicative of incomplete range testing for algorithmic bounds. In this case, the fact that the silent failures seem to be occurring instead of abort failures suggests that the development team did not have a cohesive strategy for addressing exceptional conditions, and any approach was uneven and ad-hoc at best.

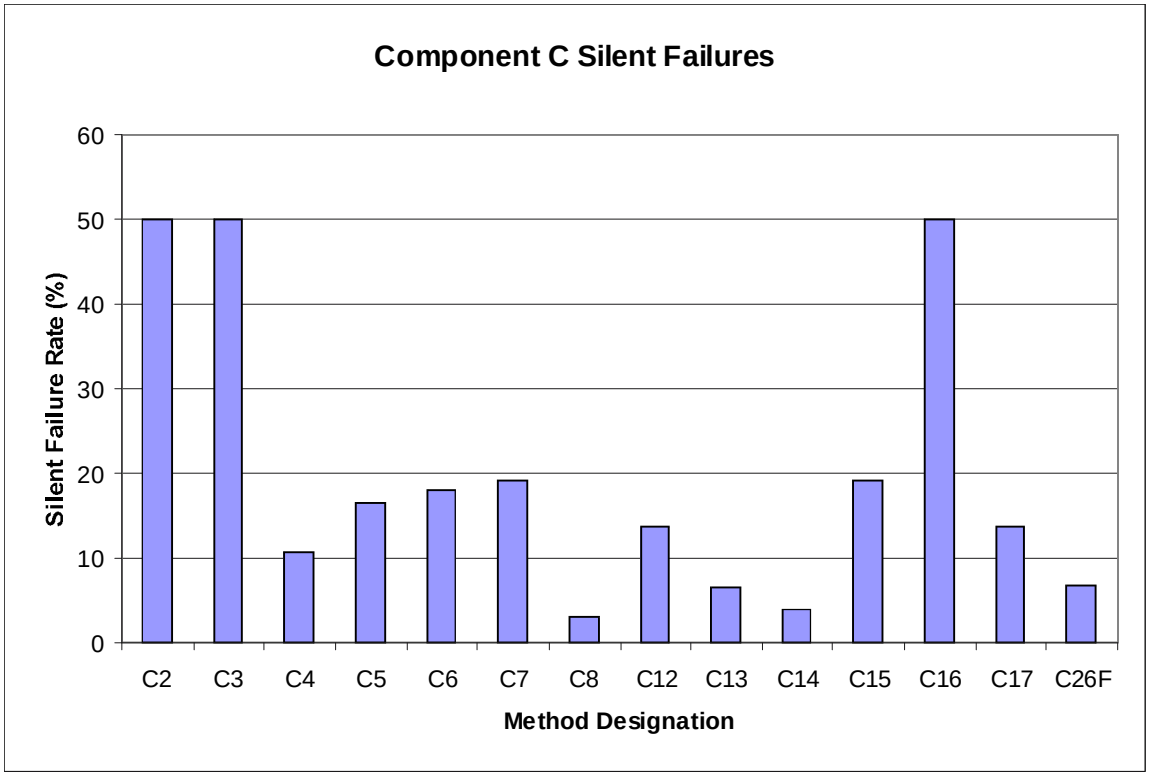


Figure 35. Component C silent failure rates

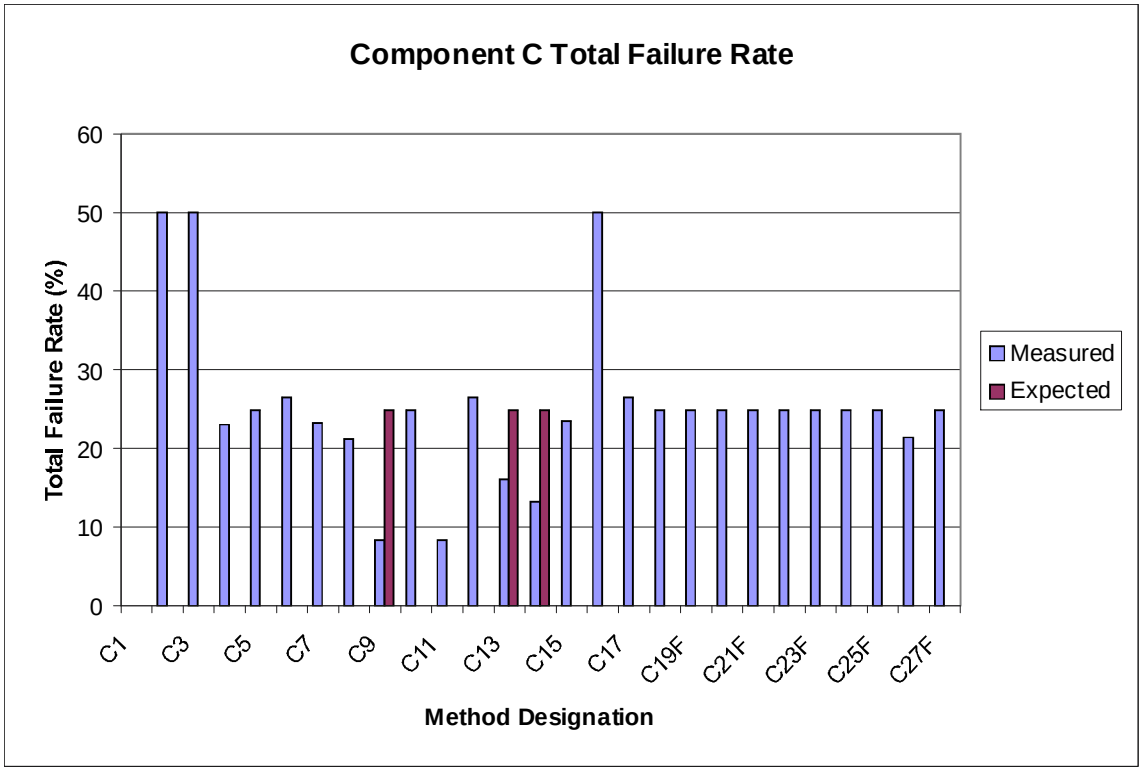


Figure 36. Total failure rate for component C

Figure 36 contains the total and expected failure rates for component C's non-constructor methods. Average measured failure rate is 24.5%.

There were three methods computationally susceptible to the expected failure for divide by zero. The expected failure rates were calculated by determining the percentage of tests that would result in a divide by zero, and was 25%. All other functions are listed as a 0% expected failure rate, since they are not susceptible to the only expected failure.

Component C did not suffer from any of the expected divide by zero failures. So even though group 3 anticipated *some* failures, they were not the ones that occurred.

8.4 Conclusions

We present the results of testing systems written by 4 distinct programming groups ranging in complexity from a small utility module to full blown distributed simulation frameworks. In each case we know how the development teams thought their systems would respond to exceptional conditions, either by inference in the case of the RTI team, or by survey as was the case with the

3 corporate development groups. Of these groups, 2 were able to classify how well their code would respond to exceptional conditions to a reasonable extent. The other two overestimated the robustness of their systems, one to a significant degree.

This data seems to provide evidence to support the conclusion that Maxion's work [maxion98] is applicable to professional development teams. Perhaps contrary to popular wisdom, even hard won industry experience is no guarantee that a team possesses the knowledge required to build a robust system capable of handling exception conditions in a graceful manner.

Although the anecdotal results presented here cannot be considered conclusive, they provide useful evidence and may help to shape the direction of future experimentation in this area. Some outstanding questions include:

- Was the individual responsible for reporting expected robustness in a position to have a good feel for the code base? Were they very junior, or perhaps management too far removed to really know?
- Some Java IDEs build exception handling wrappers around applications that mask exceptions. It is possible that teams using such environments (such as Visual Age Java) are in fact testing for these conditions, but they are being caught and ignored by the application level exception handler.

8.5 Acknowledgments

Special thanks to IBM Corporation for implementing my idea for developer self-reporting, collecting the data on their teams, and then allowing me to use it in this dissertation. Particularly Melissa Buco for writing the Lotus Notes database for data reporting and collection, and Peter Santhanam for going to the development teams and getting their cooperation in this matter.

9 Conclusions

This work has focused on gaining a better understanding of exception detection and handling. Specifically, it established a quantitative relationship between performance cost and robust exception handling across a wide range of software systems that is arguably general in scope, and determined that the performance penalty associated with robust code is low. It measured the ability of a set of professional developers to accurately classify the exception handling abilities of their software systems, and determined that for this particular case study, Maxion's hypothesis that developers without specific training on the topic might not fully grasp exceptional conditions seems to hold. It developed generic models of common exception failures, and developed generically applicable methods to fix them.

9.1 Contributions

This work makes the following contributions:

- ***Tractability*** : Proves that common robustness problems can be fixed, contrary to popular wisdom
- ***Generality*** : Presents a new methodology that can be used to address robustness problems without sacrificing performance
- ***Speed*** : Establishes that the performance cost of making a software system robust can be made negligible
- ***Developer Understanding*** : Provides a case study using data collected from corporate development groups supporting the assertion that Maxion's hypothesis can hold true for professional developers

9.1.1 Tractability and Generality

Over the course of developing the work that preceded and is included in this dissertation, it has become clear that most robustness failures stem from memory issues (for instance improperly referenced memory). A smaller number of failures manifest from structure state (for instance file status and permissions). Fewer still are a result of algorithmic boundary conditions (for instance attempting to calculate the inverse sine of 2).

The dominant form of these failures (memory) can be addressed generically across any application by using the checkMem() function developed as a part of this work. This function, and its associated structures can greatly reduce the number of robustness failures, without significant performance loss. It can be easily included in any system, and provides robust, recoverable exception detection and handling. Further, the structure is easily extensible to provide context aware exception checking for arbitrary structures.

9.1.2 Speed

Three distinct software system domains were protected using the hardening techniques developed in this work, and benchmarked for performance. Cost was on average <1% for hardened math functions, <2% for hardened IO functions, and <5% for OS service primitives. With the exception of the IO benchmarks, the benchmarks used stressed the actual calls being tested rather than building them into a realistic application. For this reason we believe the costs reported to be conservative, and suspect the true cost to a system implementation will be lower in real application code. Additionally, newer micro architectures will hide the latency of the checks better, as processors with block caches, predication, and multiple branch prediction become prevalent.

9.1.3 Developer Understanding

This work analyzed data collected from three distinct development groups reporting on how they expected their software systems to respond to exceptional conditions. The systems were then tested, and the measured robustness was related to the expected values. Additionally, one group with a design mandate for robustness was included in the testing portion of the analysis, with no self report data.

Only 2 of the 3 sel-reporting groups were able to classify the robustness of their software systems to a reasonable degree. The RTI group, with a mandate to build robust code, left a 10% abort failure in their system. Group 3 anticipated a nearly 100% robust system, when in reality it had an average failure rate of 17.7% overall.

We conclude that although this particular experiment was not scientifically rigorous in its experimental method, it provides a useful data point in validating the extension of Maxion's hypothesis to professional development teams.

9.2 Future Work

Although this work addresses some of the critical outstanding issues regarding robustness, and building robust software systems, there is much work that still needs to be done. We look at several outstanding issues directly related to this work.

9.2.1 Architectural Improvements

We assert that advanced microarchitectural features will result in faster determination of data as exceptional or normal. Unfortunately, the bulk of this work was completed just prior to new systems becoming available. Out assertion that the performance cost decreases as architectural enhancements become available needs to be validated.

Additionally, an in depth look at the robustness checking code should be done at the microarchitectural level. Such an investigation, using simulator tools such as SimpleScalar will provide better insight as to how to better structure the software to take maximum advantage of the hardware features. By natural extension, it will help identify potential enhancements to architecture that will speed robustness checks.

9.2.2 Compiler Support

We have shown that our techniques are generally applicable to any software. The next logical step is to build specific support for robustness checks into the compiler. This will allow for maximum protection with a minimum of effort while hiding the complexity of the checking structure.

9.2.3 Controlled Study

We presented a case study of how well professional developer understand the robustness response of their software systems. Although Maxion performed a careful scientific study with university programming teams, there is still no good scientific study involving professional teams. Such a study would be a huge undertaking, but would establish with certainty the need for

better education with respect to exception conditions and exception handling. Further, it would either validate or discredit the hypothesis that developers learn enough about exception handling in current educational processes to reliably design and develop robust software systems.

9.2.4 Tool Integration

The tools and techniques presented here are tenuously joined through scripts or manual manipulation. It is likely that wide scale adoption of these techniques will only occur if they are built into a consolidated, coherent, cross platform toolset. Such an undertaking, while large in scope, would enable a broader base of practitioners to use the methods presented here.

9.2.5 Detailed Performance Evaluation

While the benchmarks used within this work to evaluate the performance impact present a pessimistic upper bound, the actual overhead is unknown. Further, the implications of changing the memory footprint of a system through adding tags within allocated memory and a software cache are not clear. Research into this area could give further insight as to how the performance of robust code could be further enhanced.

10 References

[Austin94] Austin, T.M.; Breach, S.E.; Sohi, G.S., "Efficient detection of all pointer and array access errors," *Conference on Programming Language Design and Implementation* (PLDI) ACM SIGPLAN '94

[Avizienis85] Avizienis, A., "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol.SE–11, no.12, p. 1491-501

[Barton90] Barton, J., Czeck, E., Segall, Z., Siewiorek, D., "Fault injection experiments using FIAT," *IEEE Transactions on Computers*, 39(4): 575–82

[Beizer95] Beizer, B., *Black Box Testing*, New York: Wiley, 1995

[Black99] Black, Bryan, Bohuslav Rychlik and John Paul Shen, "The block-based trace cache," *Proceedings of the 26th annual International Symposium on Computer Architecture* ISCA 1999

[Buhr00] Buhr, Peter, Mok, Russell, "Advanced Exception Handling Mechanisms," *IEEE Transactions on Software Engineering*, vol. 26, number 9

[Carreira98] Carreira, J.; Madeira, H.; Silva, J.G., "Xception: a technique for the experimental evaluation of dependability in modern computers," *IEEE Transactions on Software Engineering*, vol.24, no.2 p. 125-36

[Carrette96] Carrette, G., "CRASHME: Random input testing," (no formal publication available) ple.delphi.com/gjc/crashme.html accessed July 6, 1998

[Cristian95] Cristian, F., "Exception Handling and Tolerance of Software Faults," In: *Software Fault Tolerance*, Michael R. Lyu (Ed.). Chichester: Wiley, 1995. pp. 81–107, Ch. 4

[Czeck86] Czeck, E., Feather, F., Grizzaffi, A., Finelli, G., Segall, Z. & Siewiorek, D., "Fault-free performance validation of avionic multiprocessors," *Proceedings of the IEEE/AIAA 7th Digital Avionics Systems Conference*, Fort Worth, TX, USA; 13-16 Oct. 1986, pp. 803, 670-7

[Dahmann97] Dahmann, J., Fujimoto, R., & Weatherly, R, "The Department of Defense High Level Architecture," *Proceedings of the 1997 Winter Simulation Conference*, Winter Conference Board of Directors, San Diego, CA 1997

[DeVale99] DeVale, J., Koopman, P., Guttendorf, D., "The Ballista Software Robustness Testing Service," *16th International Conference on Testing Computer Software*, 1999. pp. 33–42

[Dingman95] Dingman, C., "Measuring robustness of a fault tolerant aerospace system", *25th International Symposium on Fault-Tolerant Computing*, June 1995. pp. 522-7

[Dingman97] Dingman, C., *Portable Robustness Benchmarks*, Ph.D. thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 1997

[DoD98] U.S. Department of Defense, *High Level Architecture Run Time Infrastructure Programmer's Guide, RTI 1.3 Version 5*, Dec. 16, 1998, DMSO/SAIC/Virtual Technology Corp

[Dony90] Dony, C., "Improving Exception Handling with Object-Oriented Programming," *14th International Conference on Computer Software and Applications*, 1990

[Edelweiss98] Edelweiss, N.; Nicolao, M., "Workflow modeling: exception and failure handling representation," *18th International Conference of the Chilean Society of Computer Science*, 1998

[Fernsler99] Fernsler, K. & Koopman, P., "Robustness Testing of a Distributed Simulation Backplane," *10th International Symposium on Software Reliability Engineering*, November 1-4, 1999

[Garcia99] Garcia, A.F., Beder, D.M., Rubira, C.M.F., "An Exception handling mechanism for developing dependable object-oriented software based on a meta-level approach," *10th International Symposium on Software Reliability Engineering*, 1999

[Garcia00] Garcia, A.F., Beder, D.M., Rubira, C.M.F., "An exception handling software architecture for developing fault-tolerant software," *5th International Symposium on High Assurance System Engineering*, 2000

[Gehani92] Gehani, N., "Exceptional C or C with Exceptions," *Software – Practice and Experience*, 22(10): 827-48

[Ghosh99] Ghosh, A.K.; Schmid, M., "An approach to testing COTS software for robustness to operating system exceptions and errors," *Proceedings 10th International Symposium on Software Reliability Engineering* ISRE 1999

[Goodenough75] Goodenough, J., "Exception handling: issues and a proposed notation," *Communications of the ACM*, 18(12): 683–696, December 1975

[Govindarajan92] Govindarajan, R., "Software Fault-Tolerance in Functional Programming," *16th International Conference on Computer Software and Applications*, 1992

[Griffin00] Griffin, J.L.; Schlosser, S.W.; Ganger, G.R.; Nagle, E.F., "Modeling and performance of MEMS-based storage devices," *International Conference on Measurement and Modeling of Computer Systems* ACM SIGMETRICS '2000

[Hagen98] Hagen, C., Alonso, G., "Flexible Exception Handling in the OPERA Process Support System," *18th International Conference on Distributer Computing Systems*, 1998

[Hastings92] Hastings, R.; Joyce, B., "Purify: fast detection of memory leaks and access errors," *Proceedings of the Winter 1992 USENIX Conference*

[Hill71] Hill, I., "Faults in functions, in ALGOL and FORTRAN," *The Computer Journal*, 14(3): 315–316, August 1971

[Hof97] Hof, M., Mossenbock, H.,Pirkelbauer, P., "Zero-Overhead Exception Handling Using Metaprogramming," *Proceedings of the 24th Seminar on Current Trends in Theory and Practice of Informatics*, 1997

[Hofstede99] Hofstede, A.H.M., Barros, A.P., "Specifying Complex Process Control Aspects in Workflows for Exception Handling," *6th International Conference on Advanced Systems for Advanced Applications*, 1999

[Hull88] Hull, T.E., Cohen, M.S., Sawchuk, J.T.M., Wortman, D.B., "Exception Handling in Scientific Computing," *ACM Transactions on Mathematical Software*, Vol. 14, No 3, September 1988

[IEEE85] *IEEE standard for binary floating point arithmetic*, IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, 1985

[IEEE90] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12–1990, IEEE Computer Soc., Dec. 10, 1990

[IEEE93] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) Amendment 1: Realtime Extension [C Language]*, IEEE Std 1003.1b–1993, IEEE Computer Society, 1994

[Jones96] Jones, E., (ed.) *The Apollo Lunar Surface Journal, Apollo 11 lunar landing*, entries 102:38:30, 102:42:22, and 102:42:41, National Aeronautics and Space Administration, Washington, DC, 1996

[Kanawati92] Kanawati, G., Kanawati, N. & Abraham, J., "FERRARI: a tool for the validation of system dependability properties," *1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*. Amherst, MA, USA, July 1992, pp. 336–344

[Koopman97] Koopman, P., Sung, J., Dingman, C., Siewiorek, D. & Marz, T., "Comparing Operating Systems Using Robustness Benchmarks," *Proceedings Symposium on Reliable and Distributed Systems*, Durham, NC, Oct. 22–24 1997, pp. 72–79

[Koopman99] Koopman, P., DeVale, J.,"Comparing the Robustness of POSIX Operating Systems," *28th Fault Tolerant Computing Symposium*, June 14-18, 1999, pp. 30-37

[Koopman00] Koopman, P.; DeVale, J., "The exception handling effectiveness of POSIX operating systems," *IEEE Transactions on Software Engineering*, vol.26, no.9 p. 837-48

[Korn91] Korn, D & Vo, K.-P., "SFIO: safe/fast string/file IO," *Proceedings of the Summer 1991 USENIX Conference*, 10-14 June 1991, pp. 235-56

[Kropp98] Kropp, N., Koopman, P. & Siewiorek, D., "Automated Robustness Testing of Off-the-Shelf Software Components," *28th Fault Tolerant Computing Symposium*, June 23–25, 1998, pp. 230–239

[Lee83] Lee, P.A., "Exception Handling in C Programs," *Software Practice and Experience*. Vol 13, 1983

[Leveson93] Leveson, N.G., Turner, C.S., "An investigation of the Therac-25 accidents," *IEEE Computer*, Vol 26, N/o. 7

[Lions96] Lions, J.L. (chairman) *Ariane 5 Flight 501 Failure: report by the inquiry board*, European Space Agency, Paris, July 19, 1996

[Lippert00] Lippert, Martin, Lopes, Cristina, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming," *Proceedings of the 2000 International Conference on Software Engineering*, 2000

[Maes87] Maes, P., "Concepts and experiments in computational reflection," *Conference on Object Orientated Programming, Systems, Languages and Applications*, OOPSLA 1987

[Maxion98] Maxion, R.A.; Olszewski, R.T., "Improving software robustness with dependability cases," *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, FTCS 1998

[Marick95] Marick, B., *The Craft of Software Testing*, Prentice Hall, 1995

[Miller90] Miller, B., Fredriksen, L., So, B., "An empirical study of the reliability of operating system utilities," *Communication of the ACM*, (33):32–44, December 1990

[Miller98] Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A. & Steidl, J., "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," *Computer Science Technical Report 1268*, Univ. of Wisconsin-Madison, May 1998

[Mukherjee97] Mukherjee, A., Siewiorek, D.P., "Measuring software dependability by robustness benchmarking," *IEEE Transactions on Software Engineering*, June 1997

[Musa96] Musa, J., Fuoco, G., Irving, N. & Kropfl, D., Juhlin, B., "The Operational Profile", in: Lyu, M. (ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill/IEEE Computer Society Press, Los Alamitos CA, 1996, pp. 167–216

[Numega01] Numega. BoundsChecker. http://www.compuware.com/products/numega/bounds/, accessed 10/12/2001

[OMG95] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995

[Ostrand88] Ostrand, T.J., Balcer, M. J., "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, 31(6), 676-686

[Para01] Parasoft. Insure++. http://www.parasoft.com/products/insure/index.htm, accessed 10/12/2001

[Pu95] Pu, C.; Autrey, T.; Black, A.; Consel, C.; Cowan, C.; Inouye, J.; Kethana, L.; Walpole, J.; Ke Zhang, "Optimistic incremental specialization: streamlining a commercial operating system," *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SIGOPS 1995

[Rakvic00] Rakvic, Ryan, Black,Bryan, & Shen, John, "Completion time multiple branch prediction for enhancing trace cache performance," *The 27th Annual International Symposium on Computer architecture*, ISCA 2000

[Rational01] Rational. Purify. http://www.rational.com/products/pqc/index.jsp, accessed 10/12/2001

[Romanovsky00] Romanovsky, A., "An exception handling framework for N-version programming in object-oriented systems," *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2000

[Rotenberg96] Rotenberg, Eric, Bennett, Steve, & Smith, James E., "Trace cache: a low latency approach to high bandwidth instruction fetching," *Proceedings of the 29th annual IEEE/ACM International Symposium on Computer Architecture*, ISCA 1996

[Schuette86] Schuette, M., Shen, J., Siewiorek, D. & Zhu, Y., "Experimental evaluation of two concurrent error detection schemes," *Digest of Papers. 16th Annual International Symposium on Fault-Tolerant Computing Systems*, Vienna, Austria; 1-4 July 1986, pp. 138-43

[Sedgewick92] Sedgewick, Robert. *Algorithms in C++*. Addison-Wesley, 1992

[Shelton00] Shelton, C.P.; Koopman, P.; Devale, K., "Robustness testing of the Microsoft Win32 API," *Proceedings of the International Conference on Dependable Systems and Networks.* DSN 2000

[Siewiorek93] Siewiorek, D., Hudak, J., Suh, B. & Segal, Z., "Development of a benchmark to measure system robustness," *23rd International Symposium on Fault-Tolerant Computing*, June 1993. pp. 88-97

[Srivastava94] Srivastava, A., Eustace, A., *ATOM: A system for building customized program analysis tools*, Research Report WRL–94/2, Digital Western Research Laboratory, Palo Alto, CA, 1994

[Swain83] A.D. Swain and H.E. Guttmann, "Handbook of Human Reliability Analysis with Emphasis on Muclear Power Plant Applications," *Technical Report NUREG/CR-1278, U.S. Nuclear Regulatory Commission*, 1983

[Thekkath94] Thekkath, C., Levey, H., "Hardware and Software Support for Efficient Exception Handling," *Sixth International Conference on Architectural Support for Programming Languages*, October 1994

[Tsai95] Tsai, T., & R. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," *Proceedings Eighth International Conference. on Modeling Techniques and Tools for Computer Performance Evaluation*, Heidelberg, Germany, Sept. 20–22 1995, Springer-Verlag, pp. 26-40

[Uhlig95] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., Emer, J., "Instruction fetching: Coping with code bloat," *Proceedings 22nd Annual International Symposium on Computer Architecture*, 22–24 June 1995, ACM: New York, pp. 345–56

[Vo97] Vo, K-P., Wang, Y-M., Chung, P. & Huang, Y., "Xept: a software instrumentation method for exception handling," *The Eighth International Symposium on Software Reliability Engineering*, Albuquerque, NM, USA; 2-5 Nov. 1997, pp. 60–69

[Wilkin93] Wilken, K.D.; Kong, T., "Efficient memory access checking," *The Twenty-Third International Symposium on Fault-Tolerant Computing*, FTCS-23

[Wilkin97] Wilken, K.D.; Kong, T., "Concurrent detection of software and hardware data-access faults," *IEEE Transactions on Computers*, vol.46, no.4 p. 412-24[2]Beizer, B., Black Box Testing, New York: Wiley, 1995

[Zilles99] Zilles, C.B.; Emer, J.S.; Sohi, G.S., "The use of multithreading for exception handling," *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999

Appendix A

Complete robustness failure rates(%) for Linux kernel 2.2.16.

Call	Rate	Call	Rate	Call	Rate	Call	Rate	Call	Rate
abs	0.0	feof	1.4	isxdigit	8.0	rmdir	0.3	strncat	55.8
access	0.2	ferror	1.4	labs	0.0	sched_get_priority_max	0.0	strncmp	31.3
acos	0.0	fflush	1.4	ldexp	0.0	sched_get_priority_min	0.0	strncpy	76.7
alarm	0.0	fgetc	1.6	ldiv	3.7	sched_getparam	4.3	strpbrk	31.7
asctime	20.8	fgetpos	48.7	link	5.6	sched_getscheduler	0.0	strrchr	9.5
asin	0.0	fgets	3.3	localtime	8.3	sched_setparam	4.3	strspn	31.2
atan	0.0	fileno	0.7	log	0.0	sched_setscheduler	3.9	strstr	33.9
atan2	0.0	floor	0.0	log10	0.0	sem_close	5.9	strtod	19.5
atof	9.2	fmod	0.0	longjmp	78.6	sem_destroy	35.3	strtok	40.4
atoi	9.2	fopen	10.5	lseek	0.0	sem_getvalue	59.5	strtol	7.3
atol	9.2	fpathconf	0.0	malloc	0.0	sem_init	7.3	strtoul	7.3
calloc	0.0	fprintf	18.0	mblen	7.7	sem_open	0.0	strxfrm	44.6
ceil	0.0	fputc	1.5	mbstowcs	32.4	sem_post	64.7	sysconf	0.0
cfgetispeed	0.0	fread	33.4	mbtowc	15.9	sem_trywait	41.2	tan	0.0
cfgetospeed	0.0	free	76.5	memchr	16.4	sem_unlink	0.3	tanh	0.0
cfsetispeed	0.0	freopen	8.7	memcmp	31.3	sem_wait	64.7	tcdrain	0.0
cfsetospeed	0.0	frexp	36.4	memcpy	75.2	setbuf	3.3	tcflush	0.0
chdir	0.3	fscanf	38.8	memmove	75.2	setgid	0.0	tcgetattr	0.1
chmod	0.0	fseek	1.5	memset	57.8	setjmp	0.0	tcgetpgrp	0.0
chown	0.2	fsetpos	59.7	mkdir	0.0	setlocale	2.4	tcsendbreak	0.0
clearerr	1.4	fstat	21.1	mkfifo	0.0	setpgid	0.0	tcsetattr	0.0
close	0.0	fsync	0.0	mktime	20.8	setuid	0.0	time	12.5
closedir	42.5	ftell	1.6	mlock	0.0	setvbuf	26.3	times	4.5
cos	0.0	ftruncate	0.1	mlockall	0.0	sigaction	1.4	tmpnam	50.8
cosh	0.0	fwrite	4.4	mmap	0.0	sigaddset	6.8	tolower	0.0
creat	0.0	getc	1.6	modf	15.4	sigdelset	6.8	toupper	0.0
ctermid	12.3	getcwd	47.1	mprotect	0.0	sigemptyset	22.2	ttyname	0.0
ctime	8.3	getenv	9.2	msync	0.0	sigfillset	16.7	umask	0.0
difftime	0.0	getgrgid	0.0	munlock	0.0	sigismember	4.5	uname	0.4
div	4.0	getgrnam	8.7	munmap	0.0	siglongjmp	78.6	ungetc	1.5
dup	0.0	getgroups	21.6	open	0.0	sigpending	0.0	unlink	0.3
dup2	0.0	getpwnam	8.7	opendir	18.1	sigprocmask	0.2	utime	0.0
execl	14.8	getpwuid	0.0	pathconf	16.3	sigsetjmp	0.0	wait	0.0
execle	14.8	gmtime	8.3	perror	7.7	sin	0.0	waitpid	0.0
execlp	38.9	isalnum	8.0	pow	0.0	sinh	0.0	wcstombs	21.1
execv	2.0	isalpha	8.0	printf	34.0	sprintf	89.3	wctomb	8.6
execve	19.1	isatty	0.0	putc	1.5	sqrt	0.0	write	0.0
execvp	18.9	iscntrl	8.0	putchar	0.0	srand	0.0	strchr	9.6
exp	0.0	isdigit	8.0	puts	20.0	sscanf	48.7	strcmp	31.2
fabs	0.0	isgraph	8.0	readdir	80.8	stat	7.3	strcoll	31.2
fchmod	0.0	islower	8.0	realloc	57.3	strcat	55.6	strcpy	47.2
fclose	2.3	isprint	8.0	remove	0.3	strcspn	31.2		
fcntl	0.0	ispunct	8.0	rename	5.6	strerror	0.0		
fdatasync	0.0	isspace	8.0	rewind	1.6	strftime	58.9		
fdopen	6.7	isupper	8.0	rewinddir	47.9	strlen	9.2		

Appendix B

```
//------------------------------------------------------------------------
// b_ptr_sigactionLINUX.tpl : Ballista Datatype Template for signal
// action pointer
//
// Copyright (C) 1998-2001  Carnegie Mellon University
//
// This program is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License
// as published by the Free Software Foundation; either version 2
// of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
// 02111-1307,USA.

name structSigactionPtr b_ptr_sigaction;

parent b_ptr_buf;

includes
[
 {
  #define structSigactionPtr struct sigaction*
  #include <signal.h>
  #include "b_ptr_buf.h"
 }
]

global_defines
[
 {
  struct sigaction sigaction_temp;
  void foo_handler1(int a){
 }
 void foo_action1(int sig, siginfo_t * b, void * c){
 }
 }
]

dials
[
  enum_dial SA_HANDLER : NULL, SIG_DFL, SIG_IGN, USR_FUNC, SIG_ERR;
  enum_dial SA_MASK : EMPTY, FULL, SIGABRT, SIGSEGV, SIGINT, SIGILL,
ZERO, MAXINT;
```

```
  enum_dial SA_FLAGS : SA_NOCLDSTOP_SET, SA_SIGINFO_SET, SA_ONSTACK,
SA_RESTART, SA_ALL, NO_EXTRA, SA_ZERO, SA_MAXINT;
  enum_dial SA_SIGACTION : ACTION_NULL, ACTION_USR_FUNC;
]

access
[
{
  sigaction_temp.sa_flags = 0;
  sigaction_temp.sa_mask.__val[0] = 0;
}

  NULL
  {
    sigaction_temp.sa_handler = NULL;
  }
  SIG_DFL
  {
    sigaction_temp.sa_handler = SIG_DFL;
  }
  SIG_IGN
  {
    sigaction_temp.sa_handler = SIG_IGN;
  }
  USR_FUNC
  {
    sigaction_temp.sa_handler = foo_handler1;
  }
  SIG_ERR
  {
    sigaction_temp.sa_handler = SIG_ERR;
  }

  EMPTY
  {//no signals blocked
    if((sigemptyset (&sigaction_temp.sa_mask))!=0)
    {
      FILE* logFile = NULL;

      if ((logFile = fopen ("/tmp/templateLog.txt","a+")) == NULL)
      {
        exit(99);
      }
      fprintf (logFile, "b_ptr_sigaction - sigemptyset at EMPTY failed.
Function not tested\n");
      fclose(logFile);
      exit(99);
    }
  }
  FULL
  {//all signals blocked.
    if((sigfillset (&sigaction_temp.sa_mask))!=0)
    {
      FILE* logFile = NULL;
```

```
      if ((logFile = fopen ("/tmp/templateLog.txt","a+")) == NULL)
      {
        exit(99);
      }
      fprintf (logFile, "b_ptr_sigaction - sigfullset at FULL failed.
Function not tested\n");
      fclose(logFile);
      exit(99);
    }
  }
  SIGABRT
  {
    sigaction_temp.sa_mask.__val[0] = SIGABRT;
  }
  SIGSEGV
  {
    sigaction_temp.sa_mask.__val[0] = SIGSEGV;
  }
  SIGINT
  {
    sigaction_temp.sa_mask.__val[0] = SIGINT;
  }
  SIGILL
  {
    sigaction_temp.sa_mask.__val[0] = SIGILL;
  }
  ZERO
  {
    sigaction_temp.sa_mask.__val[0] = 0;
  }
  MAXINT
  {
    sigaction_temp.sa_mask.__val[0] = MAXINT;
  }

  SA_NOCLDSTOP_SET, SA_ALL
  {
    sigaction_temp.sa_flags |= SA_NOCLDSTOP;
  }
  SA_SIGINFO_SET, SA_ALL
  {
    sigaction_temp.sa_flags |= SA_SIGINFO;
  }
  SA_ONSTACK, SA_ALL
  {
    sigaction_temp.sa_flags |= SA_ONSTACK;
  }
  SA_RESTART, SA_ALL
  {
    sigaction_temp.sa_flags |= SA_RESTART;
  }
  SA_ZERO
  {
```

```
    sigaction_temp.sa_flags |= 0;
  }
  SA_MAXINT
  {
    sigaction_temp.sa_flags |= MAXINT;
  }
  SA_ALL
  {
    sigaction_temp.sa_flags |= SA_RESTART | SA_NODEFER |  SA_RESETHAND
|  SA_NOCLDWAIT;
  }

  ACTION_NULL
  {
    sigaction_temp.sa_sigaction = NULL;
  }
  ACTION_USR_FUNC
  {
    sigaction_temp.sa_sigaction = foo_action1;
  }

{
  _theVariable = &sigaction_temp;
}
]

commit
[
]

cleanup
[
]
```

```
/*
   b_ptr_sigaction.cpp   Generated by the Ballista(tm) Project data
object compiler
   Copyright (C) 1998-2001  Carnegie Mellon University

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 2
   of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

   File generated Tuesday, October 02 at 04:09 PM EDT

TITLE
   b_ptr_sigaction.cpp
*/

//-------------------------------------------------------------------------

#include <errno.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <stream.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include "b_ptr_sigaction.h"

//-------------------------------------------------------------------------


struct sigaction sigaction_temp;
void foo_handler1(int a){
}
void foo_action1(int sig, siginfo_t * b, void * c){
}
//-------------------------------------------------------------------------

CLASSNAME::CLASSNAME()
{
  //DIAL DECLARATIONS HERE
```

```
  //generated

   strcpy(_b_ptr_sigactionTYPENAME,CLASS_STRING);
strcpy(_b_ptr_sigactionNULL,"b_ptr_sigaction_NULL");
strcpy(_b_ptr_sigactionSIG_DFL,"b_ptr_sigaction_SIG_DFL");
strcpy(_b_ptr_sigactionSIG_IGN,"b_ptr_sigaction_SIG_IGN");
strcpy(_b_ptr_sigactionUSR_FUNC,"b_ptr_sigaction_USR_FUNC");
strcpy(_b_ptr_sigactionSIG_ERR,"b_ptr_sigaction_SIG_ERR");
strcpy(_b_ptr_sigactionEMPTY,"b_ptr_sigaction_EMPTY");
strcpy(_b_ptr_sigactionFULL,"b_ptr_sigaction_FULL");
strcpy(_b_ptr_sigactionSIGABRT,"b_ptr_sigaction_SIGABRT");
strcpy(_b_ptr_sigactionSIGSEGV,"b_ptr_sigaction_SIGSEGV");
strcpy(_b_ptr_sigactionSIGINT,"b_ptr_sigaction_SIGINT");
strcpy(_b_ptr_sigactionSIGILL,"b_ptr_sigaction_SIGILL");
strcpy(_b_ptr_sigactionZERO,"b_ptr_sigaction_ZERO");
strcpy(_b_ptr_sigactionMAXINT,"b_ptr_sigaction_MAXINT");
strcpy(_b_ptr_sigactionSA_NOCLDSTOP_SET,"b_ptr_sigaction_SA_NOCLDSTOP_S
ET");
strcpy(_b_ptr_sigactionSA_SIGINFO_SET,"b_ptr_sigaction_SA_SIGINFO_SET")
;
strcpy(_b_ptr_sigactionSA_ONSTACK,"b_ptr_sigaction_SA_ONSTACK");
strcpy(_b_ptr_sigactionSA_RESTART,"b_ptr_sigaction_SA_RESTART");
strcpy(_b_ptr_sigactionSA_ALL,"b_ptr_sigaction_SA_ALL");
strcpy(_b_ptr_sigactionNO_EXTRA,"b_ptr_sigaction_NO_EXTRA");
strcpy(_b_ptr_sigactionSA_ZERO,"b_ptr_sigaction_SA_ZERO");
strcpy(_b_ptr_sigactionSA_MAXINT,"b_ptr_sigaction_SA_MAXINT");
strcpy(_b_ptr_sigactionACTION_NULL,"b_ptr_sigaction_ACTION_NULL");
strcpy(_b_ptr_sigactionACTION_USR_FUNC,"b_ptr_sigaction_ACTION_USR_FUNC
");
}

//------------------------------------------------------------------------
b_param *b_ptr_sigaction::b_ptr_sigactionNULL()
{
   return &_b_ptr_sigactionNULL;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSIG_DFL()
{
   return &_b_ptr_sigactionSIG_DFL;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSIG_IGN()
{
   return &_b_ptr_sigactionSIG_IGN;
}

b_param *b_ptr_sigaction::b_ptr_sigactionUSR_FUNC()
{
   return &_b_ptr_sigactionUSR_FUNC;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSIG_ERR()
{
```

```
   return &_b_ptr_sigactionSIG_ERR;
}

b_param *b_ptr_sigaction::b_ptr_sigactionEMPTY()
{
   return &_b_ptr_sigactionEMPTY;
}

b_param *b_ptr_sigaction::b_ptr_sigactionFULL()
{
   return &_b_ptr_sigactionFULL;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSIGABRT()
{
   return &_b_ptr_sigactionSIGABRT;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSIGSEGV()
{
   return &_b_ptr_sigactionSIGSEGV;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSIGINT()
{
   return &_b_ptr_sigactionSIGINT;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSIGILL()
{
   return &_b_ptr_sigactionSIGILL;
}

b_param *b_ptr_sigaction::b_ptr_sigactionZERO()
{
   return &_b_ptr_sigactionZERO;
}

b_param *b_ptr_sigaction::b_ptr_sigactionMAXINT()
{
   return &_b_ptr_sigactionMAXINT;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSA_NOCLDSTOP_SET()
{
   return &_b_ptr_sigactionSA_NOCLDSTOP_SET;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSA_SIGINFO_SET()
{
   return &_b_ptr_sigactionSA_SIGINFO_SET;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSA_ONSTACK()
```

```
{
   return &_b_ptr_sigactionSA_ONSTACK;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSA_RESTART()
{
   return &_b_ptr_sigactionSA_RESTART;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSA_ALL()
{
   return &_b_ptr_sigactionSA_ALL;
}

b_param *b_ptr_sigaction::b_ptr_sigactionNO_EXTRA()
{
   return &_b_ptr_sigactionNO_EXTRA;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSA_ZERO()
{
   return &_b_ptr_sigactionSA_ZERO;
}

b_param *b_ptr_sigaction::b_ptr_sigactionSA_MAXINT()
{
   return &_b_ptr_sigactionSA_MAXINT;
}

b_param *b_ptr_sigaction::b_ptr_sigactionACTION_NULL()
{
   return &_b_ptr_sigactionACTION_NULL;
}

b_param *b_ptr_sigaction::b_ptr_sigactionACTION_USR_FUNC()
{
   return &_b_ptr_sigactionACTION_USR_FUNC;
}

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
------
//type name return method

b_param *CLASSNAME::typeName()
{
  return &_b_ptr_sigactionTYPENAME;
}

//-------------------------------------------------------------------------
------
//-------------------------------------------------------------------------
------
int CLASSNAME::distanceFromBase()
```

```
{
  return CLASSPARENT::distanceFromBase() +1;
}


//-------------------------------------------------------------------------
------
void CLASSNAME::typeList(b_param list[], int num)
{
  strcpy(list[num],(char *) typeName());
  CLASSPARENT::typeList(list,num+1);
}
//-------------------------------------------------------------------------
  void *CLASSNAME::access(b_param data[])
{
  if (strcmp(data[0],(char *)typeName())!=0)
    return CLASSPARENT::access(data);

  //ACCESS CODE
b_ptr_sigaction_null = 0;
b_ptr_sigaction_sig_dfl = 0;
b_ptr_sigaction_sig_ign = 0;
b_ptr_sigaction_usr_func = 0;
b_ptr_sigaction_sig_err = 0;
b_ptr_sigaction_empty = 0;
b_ptr_sigaction_full = 0;
b_ptr_sigaction_sigabrt = 0;
b_ptr_sigaction_sigsegv = 0;
b_ptr_sigaction_sigint = 0;
b_ptr_sigaction_sigill = 0;
b_ptr_sigaction_zero = 0;
b_ptr_sigaction_maxint = 0;
b_ptr_sigaction_sa_nocldstop_set = 0;
b_ptr_sigaction_sa_siginfo_set = 0;
b_ptr_sigaction_sa_onstack = 0;
b_ptr_sigaction_sa_restart = 0;
b_ptr_sigaction_sa_all = 0;
b_ptr_sigaction_no_extra = 0;
b_ptr_sigaction_sa_zero = 0;
b_ptr_sigaction_sa_maxint = 0;
b_ptr_sigaction_action_null = 0;
b_ptr_sigaction_action_usr_func = 0;
   int dataPTR =0;


   dataPTR++;
if (strcmp(data[dataPTR],_b_ptr_sigactionNULL)==0)
      b_ptr_sigaction_null = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSIG_DFL)==0)
      b_ptr_sigaction_sig_dfl = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSIG_IGN)==0)
      b_ptr_sigaction_sig_ign = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionUSR_FUNC)==0)
      b_ptr_sigaction_usr_func = 1;
```

```
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSIG_ERR)==0)
      b_ptr_sigaction_sig_err = 1;
  else
    {
      cerr<<"Error: Unknown setting for the "
          <<"SA_HANDLER"
          <<" dial of the data object "
          <<CLASS_STRING
          <<". "<<endl
          <<"The offending string is : "
          <<data[dataPTR]
          <<endl;
      exit(1);
    }

   dataPTR++;
if (strcmp(data[dataPTR],_b_ptr_sigactionEMPTY)==0)
      b_ptr_sigaction_empty = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionFULL)==0)
      b_ptr_sigaction_full = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSIGABRT)==0)
      b_ptr_sigaction_sigabrt = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSIGSEGV)==0)
      b_ptr_sigaction_sigsegv = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSIGINT)==0)
      b_ptr_sigaction_sigint = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSIGILL)==0)
      b_ptr_sigaction_sigill = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionZERO)==0)
      b_ptr_sigaction_zero = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionMAXINT)==0)
      b_ptr_sigaction_maxint = 1;
  else
    {
      cerr<<"Error: Unknown setting for the "
          <<"SA_MASK"
          <<" dial of the data object "
          <<CLASS_STRING
          <<". "<<endl
          <<"The offending string is : "
          <<data[dataPTR]
          <<endl;
      exit(1);
    }

   dataPTR++;
if (strcmp(data[dataPTR],_b_ptr_sigactionSA_NOCLDSTOP_SET)==0)
      b_ptr_sigaction_sa_nocldstop_set = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSA_SIGINFO_SET)==0)
      b_ptr_sigaction_sa_siginfo_set = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSA_ONSTACK)==0)
      b_ptr_sigaction_sa_onstack = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSA_RESTART)==0)
      b_ptr_sigaction_sa_restart = 1;
```

```
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSA_ALL)==0)
      b_ptr_sigaction_sa_all = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionNO_EXTRA)==0)
      b_ptr_sigaction_no_extra = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSA_ZERO)==0)
      b_ptr_sigaction_sa_zero = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionSA_MAXINT)==0)
      b_ptr_sigaction_sa_maxint = 1;
  else
    {
      cerr<<"Error: Unknown setting for the "
          <<"SA_FLAGS"
          <<" dial of the data object "
          <<CLASS_STRING
          <<". "<<endl
          <<"The offending string is : "
          <<data[dataPTR]
          <<endl;
      exit(1);
    }

   dataPTR++;
if (strcmp(data[dataPTR],_b_ptr_sigactionACTION_NULL)==0)
      b_ptr_sigaction_action_null = 1;
   else if (strcmp(data[dataPTR],_b_ptr_sigactionACTION_USR_FUNC)==0)
      b_ptr_sigaction_action_usr_func = 1;
  else
    {
      cerr<<"Error: Unknown setting for the "
          <<"SA_SIGACTION"
          <<" dial of the data object "
          <<CLASS_STRING
          <<". "<<endl
          <<"The offending string is : "
          <<data[dataPTR]
          <<endl;
      exit(1);
    }

 sigaction_temp.sa_flags = 0;
 sigaction_temp.sa_mask.__val[0] = 0;

if (b_ptr_sigaction_null==1)
{

 sigaction_temp.sa_handler = NULL;

}

if (b_ptr_sigaction_sig_dfl==1)
{

 sigaction_temp.sa_handler = SIG_DFL;
```

```
}

if (b_ptr_sigaction_sig_ign==1)
{

 sigaction_temp.sa_handler = SIG_IGN;

}

if (b_ptr_sigaction_usr_func==1)
{

 sigaction_temp.sa_handler = foo_handler1;

}

if (b_ptr_sigaction_sig_err==1)
{

 sigaction_temp.sa_handler = SIG_ERR;

}

if (b_ptr_sigaction_empty==1)
{

 if((sigemptyset (&sigaction_temp.sa_mask))!=0)
 {
 FILE* logFile = NULL;

 if ((logFile = fopen ("/tmp/templateLog.txt","a+")) == NULL)
 {
 exit(99);
 }
 fprintf (logFile, "b_ptr_sigaction - sigemptyset at EMPTY failed.
Function not tested\n");
 fclose(logFile);
 exit(99);
 }

}

if (b_ptr_sigaction_full==1)
{

 if((sigfillset (&sigaction_temp.sa_mask))!=0)
 {
 FILE* logFile = NULL;

 if ((logFile = fopen ("/tmp/templateLog.txt","a+")) == NULL)
 {
 exit(99);
 }
```

```
 fprintf (logFile, "b_ptr_sigaction - sigfullset at FULL failed.
Function not tested\n");
 fclose(logFile);
 exit(99);
 }

}

if (b_ptr_sigaction_sigabrt==1)
{

 sigaction_temp.sa_mask.__val[0] = SIGABRT;

}

if (b_ptr_sigaction_sigsegv==1)
{

 sigaction_temp.sa_mask.__val[0] = SIGSEGV;

}

if (b_ptr_sigaction_sigint==1)
{

 sigaction_temp.sa_mask.__val[0] = SIGINT;

}

if (b_ptr_sigaction_sigill==1)
{

 sigaction_temp.sa_mask.__val[0] = SIGILL;

}

if (b_ptr_sigaction_zero==1)
{

 sigaction_temp.sa_mask.__val[0] = 0;

}

if (b_ptr_sigaction_maxint==1)
{

 sigaction_temp.sa_mask.__val[0] = MAXINT;

}

if (b_ptr_sigaction_sa_nocldstop_set==1 || b_ptr_sigaction_sa_all==1)
{

 sigaction_temp.sa_flags |= SA_NOCLDSTOP;
```

```
}

if (b_ptr_sigaction_sa_siginfo_set==1 || b_ptr_sigaction_sa_all==1)
{

 sigaction_temp.sa_flags |= SA_SIGINFO;

}

if (b_ptr_sigaction_sa_onstack==1 || b_ptr_sigaction_sa_all==1)
{

 sigaction_temp.sa_flags |= SA_ONSTACK;

}

if (b_ptr_sigaction_sa_restart==1 || b_ptr_sigaction_sa_all==1)
{

 sigaction_temp.sa_flags |= SA_RESTART;

}

if (b_ptr_sigaction_sa_zero==1)
{

 sigaction_temp.sa_flags |= 0;

}

if (b_ptr_sigaction_sa_maxint==1)
{

 sigaction_temp.sa_flags |= MAXINT;

}

if (b_ptr_sigaction_sa_all==1)
{

 sigaction_temp.sa_flags |= SA_RESTART | SA_NODEFER | SA_RESETHAND |
SA_NOCLDWAIT;

}

if (b_ptr_sigaction_action_null==1)
{

 sigaction_temp.sa_sigaction = NULL;

}

if (b_ptr_sigaction_action_usr_func==1)
```

```
{

 sigaction_temp.sa_sigaction = foo_action1;

}

 _theVariable = &sigaction_temp;
   return &_theVariable;
}

//---------------------------------------------------------------------
  int CLASSNAME::commit(b_param tname)
{
  if (strcmp(tname,(char *)typeName())!=0)
    return CLASSPARENT::commit(tname);
  //COMMIT CODE HERE
//generated
   return 0;
}

//---------------------------------------------------------------------
  int CLASSNAME::cleanup(b_param tname)
{
  if (strcmp(tname,(char *)typeName())!=0)
    return CLASSPARENT::cleanup(tname);

  //CLEANUP CODE
//generated
   return 0;
}

//---------------------------------------------------------------------
int CLASSNAME::numDials(b_param tname)
{
  if (!strcmp(tname,(char *)typeName()))
    return NUMBER_OF_DIALS;
  else return CLASSPARENT::numDials(tname);

}


//---------------------------------------------------------------------
------
int CLASSNAME::numItems(b_param tname,int dialNumber)
{
  if (strcmp(tname,(char *)typeName())!=0)
    return CLASSPARENT::numItems(tname,dialNumber);
  switch (dialNumber)
    {
      //NUMITEMS SWITCH CASES HERE
      //generated

   case 1:
      return 5;
```

```
      break;

   case 2:
      return 8;
      break;

   case 3:
      return 8;
      break;

   case 4:
      return 2;
      break;
//-----------------------------------------------------------------------
------
      //end generated

    default:
      cerr<<"Error, invalid dial number passed to "
          <<CLASS_STRING<<"::numItems\n"
          <<"Please check declaration files.  Dial number passed was "
          <<dialNumber<<endl;
      exit(1);
    }
  return 0;
}

//-----------------------------------------------------------------------
------
b_param *CLASSNAME::paramName(b_param tname,
                                    int dialNumber,
                                    int position)

{
  if (strcmp(tname,(char *)typeName())!=0)
    return CLASSPARENT::paramName(tname,dialNumber,position);

  switch (dialNumber)
    {
      //PARAMNAME SWITCH CASES HERE
      //generated

case 1:
 switch (position)
   {
      case 1:
         return b_ptr_sigactionNULL();
         break;
      case 2:
         return b_ptr_sigactionSIG_DFL();
         break;
      case 3:
         return b_ptr_sigactionSIG_IGN();
         break;
```

```
      case 4:
         return b_ptr_sigactionUSR_FUNC();
         break;
      case 5:
         return b_ptr_sigactionSIG_ERR();
         break;

      default:
          cerr<<"Error, invalid position number passed to "
              <<CLASS_STRING<<"::paramName\n"
              <<"Please check declaration files.  Dial number passed
was"
              <<dialNumber<<" position "<<position<<".\n";
          exit(1);
   }
   break;

case 2:
 switch (position)
   {
      case 1:
         return b_ptr_sigactionEMPTY();
         break;
      case 2:
         return b_ptr_sigactionFULL();
         break;
      case 3:
         return b_ptr_sigactionSIGABRT();
         break;
      case 4:
         return b_ptr_sigactionSIGSEGV();
         break;
      case 5:
         return b_ptr_sigactionSIGINT();
         break;
      case 6:
         return b_ptr_sigactionSIGILL();
         break;
      case 7:
         return b_ptr_sigactionZERO();
         break;
      case 8:
         return b_ptr_sigactionMAXINT();
         break;

      default:
          cerr<<"Error, invalid position number passed to "
              <<CLASS_STRING<<"::paramName\n"
              <<"Please check declaration files.  Dial number passed
was"
              <<dialNumber<<" position "<<position<<".\n";
          exit(1);
   }
   break;
```

```
case 3:
 switch (position)
   {
      case 1:
         return b_ptr_sigactionSA_NOCLDSTOP_SET();
         break;
      case 2:
         return b_ptr_sigactionSA_SIGINFO_SET();
         break;
      case 3:
         return b_ptr_sigactionSA_ONSTACK();
         break;
      case 4:
         return b_ptr_sigactionSA_RESTART();
         break;
      case 5:
         return b_ptr_sigactionSA_ALL();
         break;
      case 6:
         return b_ptr_sigactionNO_EXTRA();
         break;
      case 7:
         return b_ptr_sigactionSA_ZERO();
         break;
      case 8:
         return b_ptr_sigactionSA_MAXINT();
         break;

      default:
          cerr<<"Error, invalid position number passed to "
              <<CLASS_STRING<<"::paramName\n"
              <<"Please check declaration files.  Dial number passed
was"
              <<dialNumber<<" position "<<position<<".\n";
          exit(1);
   }
   break;

case 4:
 switch (position)
   {
      case 1:
         return b_ptr_sigactionACTION_NULL();
         break;
      case 2:
         return b_ptr_sigactionACTION_USR_FUNC();
         break;

      default:
          cerr<<"Error, invalid position number passed to "
              <<CLASS_STRING<<"::paramName\n"
              <<"Please check declaration files.  Dial number passed
was"
```

```
            <<dialNumber<<" position "<<position<<".\n";
          exit(1);
   }
   break;
    default:
      cerr<<"Error, invalid dial number passed to "
          <<CLASS_STRING<<"::paramName\n"
          <<"Please check declaration files.  Dial number passed was "
          <<dialNumber<<endl;
      exit(1);

    }
  return NULL;
}

/*
   b_ptr_sigaction.h   Generated by the Ballista(tm) Project data
object compiler
   Copyright (C) 1998-2001  Carnegie Mellon University

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 2
   of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

   File generated Tuesday, October 02 at 04:09 PM EDT

TITLE
   b_ptr_sigaction.h
*/

//include control
#ifndef B_PTR_SIGACTION_H
#define B_PTR_SIGACTION_H
#include <errno.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <stream.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "bTypes.h"
```

```
#define structSigactionPtr struct sigaction*
#include <signal.h>
#include "b_ptr_buf.h"
#define CLASSTYPE structSigactionPtr
#define CLASSNAME b_ptr_sigaction
#define CLASS_STRING "b_ptr_sigaction"
#define CLASSPARENT b_ptr_buf
#define NUMBER_OF_DIALS 4


class CLASSNAME:public CLASSPARENT
{
private:
  //CLASS DIAL SETTING STRING VARIABLES
  b_param _b_ptr_sigactionTYPENAME;
b_param _b_ptr_sigactionNULL;
int b_ptr_sigaction_null;
b_param _b_ptr_sigactionSIG_DFL;
int b_ptr_sigaction_sig_dfl;
b_param _b_ptr_sigactionSIG_IGN;
int b_ptr_sigaction_sig_ign;
b_param _b_ptr_sigactionUSR_FUNC;
int b_ptr_sigaction_usr_func;
b_param _b_ptr_sigactionSIG_ERR;
int b_ptr_sigaction_sig_err;
b_param _b_ptr_sigactionEMPTY;
int b_ptr_sigaction_empty;
b_param _b_ptr_sigactionFULL;
int b_ptr_sigaction_full;
b_param _b_ptr_sigactionSIGABRT;
int b_ptr_sigaction_sigabrt;
b_param _b_ptr_sigactionSIGSEGV;
int b_ptr_sigaction_sigsegv;
b_param _b_ptr_sigactionSIGINT;
int b_ptr_sigaction_sigint;
b_param _b_ptr_sigactionSIGILL;
int b_ptr_sigaction_sigill;
b_param _b_ptr_sigactionZERO;
int b_ptr_sigaction_zero;
b_param _b_ptr_sigactionMAXINT;
int b_ptr_sigaction_maxint;
b_param _b_ptr_sigactionSA_NOCLDSTOP_SET;
int b_ptr_sigaction_sa_nocldstop_set;
b_param _b_ptr_sigactionSA_SIGINFO_SET;
int b_ptr_sigaction_sa_siginfo_set;
b_param _b_ptr_sigactionSA_ONSTACK;
int b_ptr_sigaction_sa_onstack;
b_param _b_ptr_sigactionSA_RESTART;
int b_ptr_sigaction_sa_restart;
b_param _b_ptr_sigactionSA_ALL;
int b_ptr_sigaction_sa_all;
b_param _b_ptr_sigactionNO_EXTRA;
int b_ptr_sigaction_no_extra;
```

```
b_param _b_ptr_sigactionSA_ZERO;
int b_ptr_sigaction_sa_zero;
b_param _b_ptr_sigactionSA_MAXINT;
int b_ptr_sigaction_sa_maxint;
b_param _b_ptr_sigactionACTION_NULL;
int b_ptr_sigaction_action_null;
b_param _b_ptr_sigactionACTION_USR_FUNC;
int b_ptr_sigaction_action_usr_func;
  //TYPE VARIABLE TO SAVE VALUE FOR DESTRUCTION
  CLASSTYPE _theVariable;

 public:
  //CLASS DIAL SETTING STRING ACCESS METHODS
b_param *b_ptr_sigactionNULL();
b_param *b_ptr_sigactionSIG_DFL();
b_param *b_ptr_sigactionSIG_IGN();
b_param *b_ptr_sigactionUSR_FUNC();
b_param *b_ptr_sigactionSIG_ERR();
b_param *b_ptr_sigactionEMPTY();
b_param *b_ptr_sigactionFULL();
b_param *b_ptr_sigactionSIGABRT();
b_param *b_ptr_sigactionSIGSEGV();
b_param *b_ptr_sigactionSIGINT();
b_param *b_ptr_sigactionSIGILL();
b_param *b_ptr_sigactionZERO();
b_param *b_ptr_sigactionMAXINT();
b_param *b_ptr_sigactionSA_NOCLDSTOP_SET();
b_param *b_ptr_sigactionSA_SIGINFO_SET();
b_param *b_ptr_sigactionSA_ONSTACK();
b_param *b_ptr_sigactionSA_RESTART();
b_param *b_ptr_sigactionSA_ALL();
b_param *b_ptr_sigactionNO_EXTRA();
b_param *b_ptr_sigactionSA_ZERO();
b_param *b_ptr_sigactionSA_MAXINT();
b_param *b_ptr_sigactionACTION_NULL();
b_param *b_ptr_sigactionACTION_USR_FUNC();
  //CLASS CONSTRUCTOR
  CLASSNAME();

public:
  //Mandatory Methods
  b_param *typeName();           //returns the type of parameter
  virtual void *access(b_param data[]);
  virtual int commit(b_param tname);
  virtual int cleanup(b_param tname);

  virtual int numDials(b_param tname);
  virtual int numItems(b_param tname,int dialNumber);
  virtual b_param *paramName(b_param tname,int dialNumber, int
position);

  virtual int distanceFromBase();
  virtual void typeList(b_param list[],int num);
```

```
};


#endif      //CLASSNAME_H
```