2002 International Conference on Dependable Systems and Networks

Robust Software – No More Excuses

John DeVale
Intel Corporation
Intel Labs
devale@computer.org

Philip Koopman
Carnegie Mellon University
ECE Department
koopman@cmu.edu

Abstract

Software developers identify two main reasons why software systems are not made robust: performance and practicality. This work demonstrates the effectiveness of general techniques to improve robustness that are practical and yield high performance. We present data from treating three systems to improve robustness by a factor of 5 or more, with a measured performance penalty of under 5% in nearly every case, and usually under 2%.

We identify a third possible reason why software systems are not made robust: developer awareness. A case study on three professional development groups evaluated their ability to estimate the robustness of their software. Two groups were able to estimate their software's robustness to some extent, while one group had more divergent results. Although we can overcome the technical challenges, it appears that even experienced developers can benefit from tools to locate robustness failures and training in robustness issues.

1. Introduction

As our society becomes more dependent 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 dependent 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.

This is not an entirely unfamiliar problem. 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 [21] [27] [28], the effort to build robust systems was made.

Unfortunately, it is not unusual for system developers to pay too little attention to the need for building robust systems. Operating systems have shown to have at times poor robustness [24][12]. Commercial distributed computing frameworks such as CORBA client software tend to exhibit problems as well [31]. Even complex military distributed simulation frameworks have a lower, but significant, rate of robustness problems [9].

The past few years have seen a number of research efforts that measure, to some degree, some aspect of software fault tolerance or robustness [22][6][16][4]. Although none of these tools can reasonably be expected to be the ultimate authority on the measurement of system reliability, each can provide developers useful insight into some aspects of the system that arguably contribute to reliability.

Although the availability of these tools might lead one to expect that system developers would use them to steadily improve, this does not seem to be the case. While some groups have pursued using such tools to improve their systems, many more do not. We found it puzzling that some developers would not be interested in improving their products, and strove to understand the phenomenon.

In the course of the work done throughout the Ballista Project, we were afforded the opportunity to interact with a number of development groups. Some of the systems these groups were developing fall within the more traditional areas accustomed to the need for fault tolerance (military), while others were in areas whose need for fault tolerance and robust exception handling are only now being realized. These interactions provided some insight as to why some groups were reluctant to fix issues unless they could be traced directly to a total system failure.

Among the developers with whom we interacted, the primary concerns cited to justify not fixing all robustness problems found were: performance and practicality. Both of these issues bear close scrutiny. It is readily apparent that the need for any approach to address robustness issues must be easy to implement to reduce development cost. Additionally, run time performance must be fast on all platforms, and built solely on top of existing hardware without the need for architectural changes to enhance performance. This is largely because the majority of systems being built use commodity hardware and processors, which are usually manufactured with speed, cost, and quality in mind rather than supporting software fault tolerance.

To address the developer's concerns we demonstrate the effectiveness of general techniques to improve robustness that are practical and yield high performance. We present data from treating three systems to remove all detectable ro-

bustness failures, with a measured performance penalty of under 5% in nearly every case, and usually under 2%.

Although we can demonstrate solutions to the technical challenges involved in building robust systems, there remains the issue of familiarity, or awareness first explored by Maxion in [30]. Maxion concludes that through the exposure to a few simple ideas and concepts, student programming groups will produce more robust code. The obvious observation that can be made is that during their course work, the students in Maxion's study had not been effectively exposed to the types of exceptional conditions that occur in real world applications, or potential methods of dealing with them. One might wonder if such a result is indicative of any developer, or just students. Put another way, are professional development teams any more capable of building robust code than the control groups in Maxion's study?

To gain insight into this issue we present data from a case study performed using 3 professional development teams. The study attempts to determine how well the developers are able to predict the exception handling characteristics of their code. If their predictions were accurate, one might conclude that given the proper tools, and time they could build robust software systems. If they were not accurate, it is uncertain that they would be able to develop a robust system, even if they were asked to do so.

2. Previous Work

The literature 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; and how to do it quickly.

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 [17] [13]. Over the years two methods have come to dominate current implementations. These methods are the termination model and the resumption model [11].

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 [5]. Indeed, the implementation of resumption model semantics via signals in operating systems 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 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 [20].

At a higher level of abstraction, several formal frameworks for representing exception handling and recovery have been developed [18]. 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 [8]. Yet another approach is to use computational reflection to separate the exception handling code from the normal computational code [10].

2.2 Performing Exception Handling

Exception handling mechanisms can often make code generation and understanding 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 [26] [19] [15] [3]. More aggressive approaches go beyond simple compiler constructs build entire frameworks [14] [33] or language constructs [29].

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 [36]. 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. Once exceptions are generated, it can be difficult to recover from them in a robust fashion. The previous work discussed in this section largely focuses on generating, propagating, and handling exceptions as quickly as possible. That is complementary to the work presented herein. This work is mainly interested in developing methods of including enhanced error detection in software systems to detect incipient 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 [35], the authors present a hardware/software solution that can reduce delivery cost by an order of magnitude. In [38], the use of multithreading is explored to handle hardware exceptions such as TLB misses without squashing the main instruction thread. The work presented herein 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 processors using enhanced multiple branch prediction hardware[32] and block caches[2] can simply execute checks in parallel and speculatively bypass them with little or no performance cost.

In [37] 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% [1].

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. Methodology

One of the problems with simply allowing an exception to occur and cleaning up after it later is the questionable viability of any post-exception cleanup effort. The POSIX standard does not guarantee process state after signal delivery, and it is possible that the function in which the exception occurred was altering the system state in a way that is impossible or difficult to undo. Additionally, some processors on the market do not support precise exceptions. Given these issues, there is little guarantee that a portable program can recover from an exceptional condition in a graceful fashion.

For this reason our approach is to detect all possible exceptional conditions before any calculations are performed or actions are taken. This is accomplished through the rigorous validation of input data. Once the data has been validated, processing can continue as normal.

The result of this preemptive detection is that the process has the opportunity to respond gracefully to exceptional conditions before process state is altered. Addressing the issue of providing the mechanisms to handle exceptional conditions within applications is beyond the scope of this work, although we note that in many cases a simple retry can be effective. However, the main point of this work is to provide the opportunity to handle exceptions without incurring the overhead of a process restart, whereas in the past there was no portable way to do so in practice.

A generically applicable method for hardening a software interface is to harden each element of a function's incoming parameters. This corresponds to creating wrappers on a per-parameter basis. We have found that linking hardening code to data types is a scalable approach, and mirrors the abstraction of testing to data types within the software interface as is done in the Ballista testing service [6].

The main benefit from basing wrapper checks on data type is enhancing the modularity and reuse of checks. Checks can be completely encapsulated in a function call,

```
ReturnType function foo(dataTypeA a)
{
       if (!checkDataTypeA(a))
       {
              return ErrorCondition
       }
       .
       .
       .
       Perform normal calculations
       .
       .
       .
       Return result
}
```

Figure 1. Pseudocode illustrating entry checks for exceptional conditions.

and used as needed to harden against robustness failures. Upon entering any hardened function, a wrapper function is invoked for each of the incoming parameter values (see Figure 1). There are some instances where such an abstraction breaks, for instance when values may hold special context or exceptional values dependent on the functionality of the method being protected. Nonetheless, even when this occurs the function can be hardened with a slightly customized version of the generic hardening procedure at small cost. (This is analogous of creating custom parameter tests for application-specific data types based on inheritance from generic data types, as is done in the Ballista test harness.)

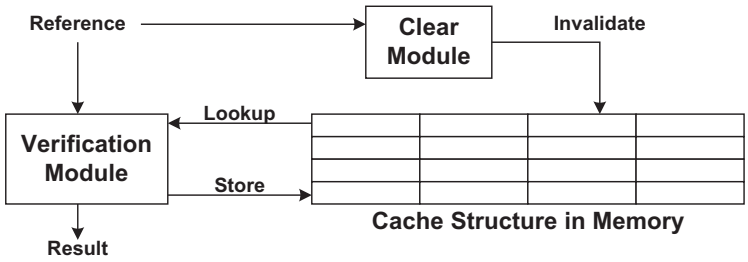

Figure 2. Software implemented direct mapped robustness check cache.

One of the problems encountered when fixing robustness issues due to memory problems (*e.g.* improperly allocated, or uninitialized memory) is lack of information accessible by user level programs. Although most systems and memory allocation schemes track the information needed to determine if a memory block is exceptional or not, the information is generally not exposed to the user. This fundamentally limits how well a developer can detect memory related exceptional conditions.

To remove this fundamental limitation we slightly modified malloc() and provided function hooks that could read the data needed to validate dynamically allocated memory at the user level. Such a technique could be used to store and retrieve other context dependent information pertaining to the memory block. For our purposes, it was sufficient to simply allow reading the information.

Making the checks practical is only part of the solution, since as it stands now, the overhead of our proposed method would be extremely high. The added cost of several procedure calls, each with several exceptional condition checks, would only be comparatively small for complex functions with long running times. But it is more often the case that the functions which need protection from exceptions are short and time critical. As such, even a few hundred nanoseconds for a full exception check might take an unacceptably long time.

To reduce the run-time cost of data validation, we propose the use of a *software-implemented* validation check cache (Figure 2). It is a direct mapped, memory resident, software managed cache. The address of the data structure to be validated is used to index into the cache. To repeat the emphasis, even though it looks like a typical hardware implementation technique, this is a purely software-implemented approach.

The cache is used to exploit the temporal locality of data accesses by caching the results (exceptional/non exceptional) of data validation so that in the event of a cache hit, the checks are completely bypassed. The operation is straightforward. After a data structure is successfully validated its address is entered into the cache. Any modification to the structure or memory block that would potentially cause it to become exceptional causes an invalidation of the cache entry. Although this invalidation is currently inserted by hand, one can envision a compile time system by which invalidations would be inserted automatically. Any time a block or structure to be validated has a cache entry, the checks can be bypassed.

Managing the cache in software and having it memory resident in the user process memory space results in configuration benefits. First, it allows the developer to determine how large to make the cache allowing balancing performance vs memory footprint. Additionally, it gives the developer the opportunity to have multiple caches for different purposes if desired.

Of course the primary benefit of a completely software solution is that it requires no hardware to operate, is portable, and can be implemented on any hardware platform. In the past, commercial processor vendors have exhibited a resistance to adding hardware features solely for the purpose of improving software reliability. Previous research in the area of detection of memory exceptions that require hardware has languished unimplemented by the processor community.

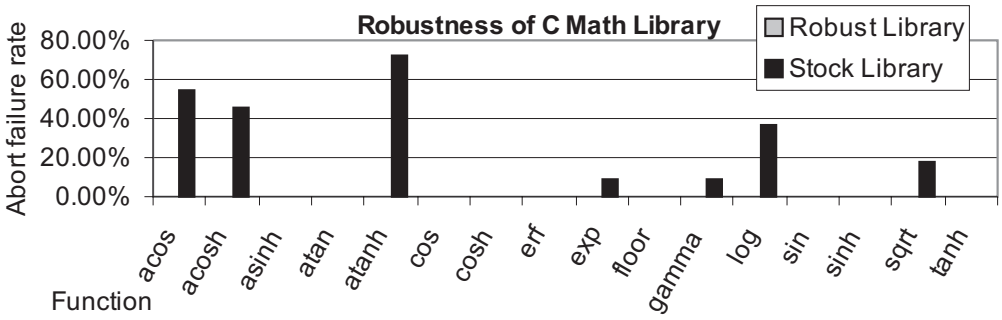

Figure 3. Measure robustness of math libraries before (stock) and after (robust) treatment. The robust version of the library has no measurable robustness failures.

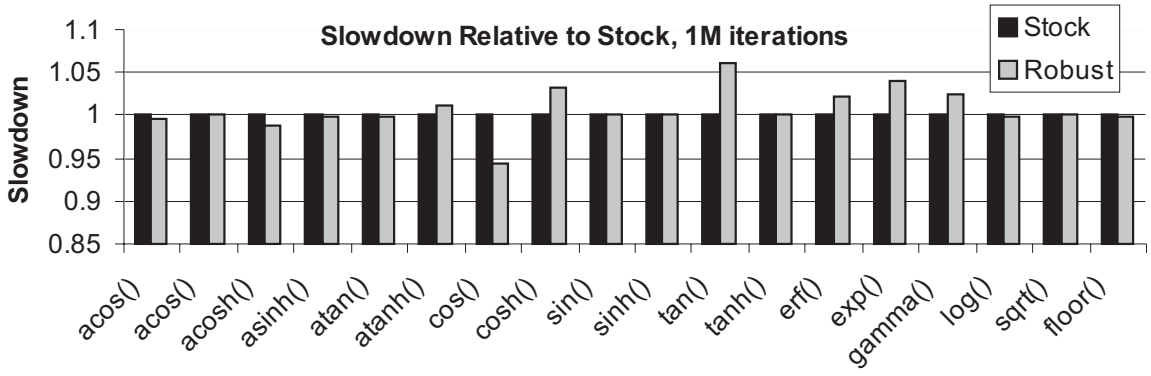

Figure 4. Slowdown of robust math library normalized to stock library performance.

4. Experimental Results

As a result of the feedback from developers, we choose to look at several software systems to determine if the robustness failures could be removed in a practical fashion without excessive sacrifices in performance.

This research occurred over the course of several years, and thus some of the experiments were carried out on systems which are currently out of date. However, the fact that even older systems could be improved with these techniques indicates that there has been no sudden reversal of situation that makes this approach viable, but rather a steady accumulation of hardware capabilities over time. The important point to be made is that the techniques presented are a general, practical applicability of a methodology to improve robustness while yielding high performance, and not simply improvements of any specific module under treatment.

During our initial investigations with Ballista, we noticed a number of robustness failures in the math libraries in FreeBSD [24]. As an initial study, we chose to address these issues and determine how robust the libraries could be made, and at what cost. The performance of the libraries was measured on a Pentium-133 with 64 MB of main memory running FreeBSD 3.0

Figure 3 shows the measured robustness of the math library before and after treatment. Figure 4 shows the performance of the hardened libraries normalized to that of the stock libraries. Performance was measured by iteratively calling each function with valid data 1,000,000 times. The results reported are the average of 10 complete runs. Because the math libraries are largely stateless, the use of a check cache was forgone, and all checks were performed for each function invocation. In a few instances, the robust libraries performance was better than stock. These are instances, most notably in cos(), where the pre-existing tests for exception conditions could be folded into the added checks and made more efficient.

Function	Average Overhead
memchr	22.3
memcpy	2.1
memcmp	7.1
memset	4.9
memmove	5.7
sem_init	10.9
sem_destroy	7.3
sem_getvalue	9.8
sem_post	7.4
sem_wait	6.2
sem_trywait	6.4

Table 1. Average overhead (in ns) for hardening Linux.

Our next investigation was the analysis and hardening of the Safe Fast I/O library [25]. The library has been retreated with techniques developed since [7] to remove all detectable robustness failures. Summary performance results are shown in Figure 5. The average failure rate of the methods within the untreated SFIO library was 4.5%.

The third system we investigated was elements of the Linux API. Although the LINUX developers have been systematically removing robustness failures over the course of the last several releases, a few critical areas such as memory manipulation and process synchronization

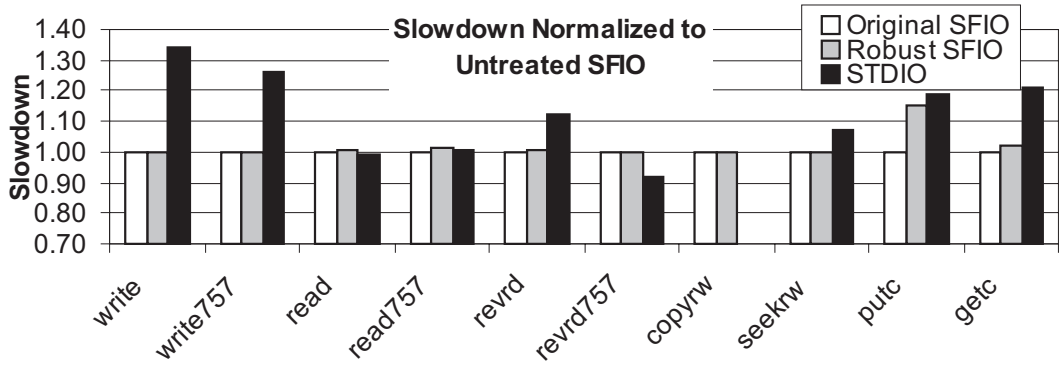

Figure 5. Performance of IO Benchmarks from [25]. Results represent slowdown normalized to untreated SFIO.

have been left in an unhardened state. Figure 6 shows the before and after treatment robustness of the functions selected for hardening.

Table 1 shows the average actual overhead incurred for making the functions robust in nanoseconds. Although the most common case is a hit in the cache reducing the checks to an index lookup and a jump, the absolute overheads do vary. This is due to differences in how well the microprocessor can exploit the instruction level parallelism in the treated function. Memchr() is highly optimizable, and makes more complete use of processor resources. Because of this the processor has only a few unused resources with which to execute the exception checking code in parallel with useful computation, and the paid overhead is slightly higher.

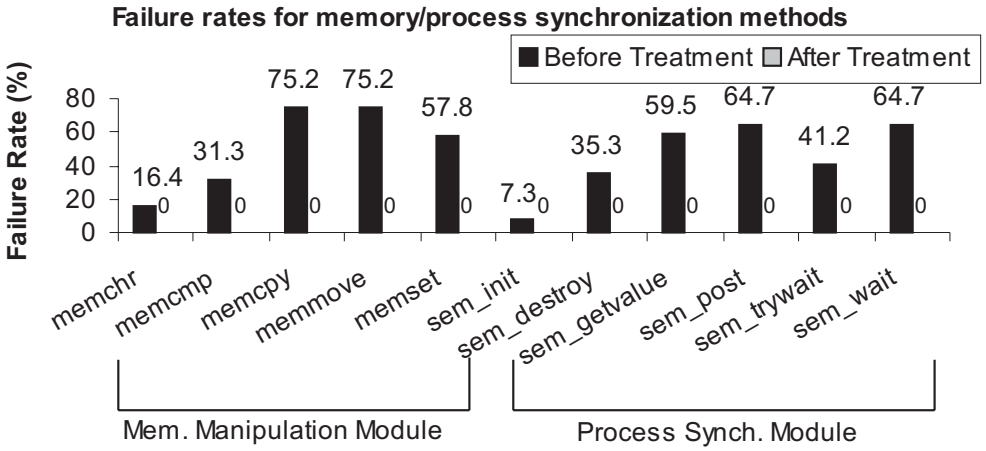

Figure 6. Before and after robustness of LINUX functions selected for treatment. After treatment, no function had detectable robustness failures.

The average performance penalties for iterative calls to the treated functions are located in Figures 7 and 8. Overhead for process synchronization is given in Figure 9, as measured by a synthetic application benchmark that simply obtained semaphore locks, enqueued an integer value, dequeued an integer value, and released the locks. No processing occurred other than the single enqueue/dequeue operation.

In the worst case, two of the memory functions exhibit relatively large slowdowns for small buffer operand sizes. These two functions, memchr() and memset() are highly optimizable and leave fewer unused processor resources than typical functions do. Although the overhead is 19% and 11% respectively, it represents less than 25 nanoseconds. Sem_getvalue, which consists of a single load instruction suffers the worst penalty of 37%.

Although it may seem untoward to benchmark memory functions iteratively, it is logical considering that the memory functions are often used to move data in isolation from other computation. As such, it made the most sense to test them in isolation (iteratively) to obtain the worst case overhead performance data. Used in combination with other instructions and algorithms, the cost will be less as the overhead is amortized over larger amounts of processing.

In the case of process synchronization however, the functions are always used within the scope of some larger computational module. For this reason we measured their performance within the context of a trivial synthetic application. Only the most basic computation occurs inside of the programming structures that obtain and release semaphore locks. As with the iterative testing of the memory functions, this yields an extremely pessimistic value for overhead. Any application that performs non-trivial computation within a critical section (protected by semaphores) will experience much less relative overhead. Note that performance does begin to degrade due to conflict misses as the number of objects contending for cache entries approaches the cache size.

5. Analysis

All robustness failures detectable by the Ballista testing harness in the three test systems could be removed. It cannot be concluded that the systems are now completely robust, because Ballista does not test every possible robustness failure. However, the tests used attained broad coverage by testing combinations of basic data type attributes, so they attained reasonably good coverage. Furthermore, if new failures become detectable, they can be fixed using the standard techniques outlined here at low performance cost.

While there is some variation in performance penalty, only the most heavily optimizable functions running on short data sets exhibit large slowdowns. Absolute overhead

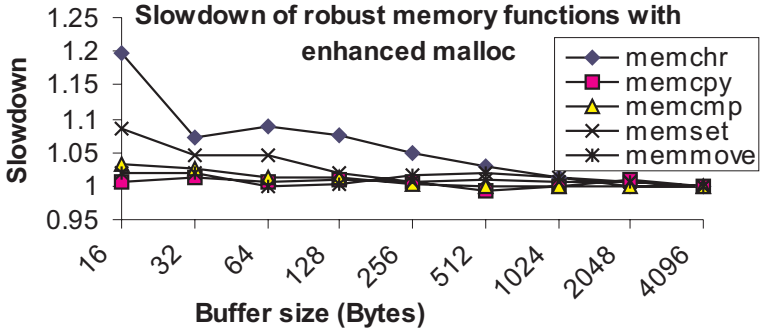

Figure 7. Performance of hardened memory functions for varying input sizes.

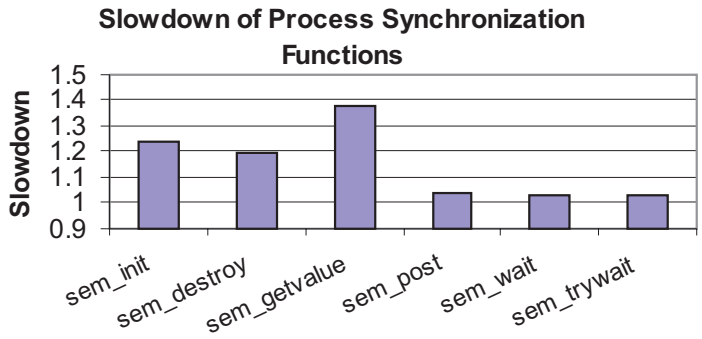

Figure 8. Performance of hardened process synchronization functions.

was measured at 25 nanoseconds or less on a dual pIII 600 system. Nearly all of the hardened functions suffer performance penalties of less than 5%, and most were less than 2%.

The overhead that is incurred is reduced to (in the case of a robustness check cache hit) that of executing a few integer instructions to calculate the cache index, a load, a compare, and a branch. This small number of instructions usually represents only a small fraction of the running time of the protected code, thus yielding high performance. Even in the case of short, speed critical functions much of the latency of the cache lookup is hidden through hardware exploitable parallelism.

In the event of a cache miss, the full suite of checks must be performed, and as indicated in Figure 9, this cost can be substantial. Of course, these penalties can be minimized through optimizing both the size of the cache and the number of caches on a per-application basis. It is also possible to use multiple caches, providing a developer the opportunity to finely tune the operation of the system.

It is notable that even functions that consist of only a few hardware instructions can be hardened with a relatively small speed penalty. It is at first counterintuitive that doubling (in some cases) the number of instructions results in only a 20% speed reduction. This result is easily explained through the realization that although modern processors can extract a great deal of instruction level parallelism from some of these functions, there is still a fair amount of unused processor resources available for concurrent use by the exception detection code.

This data shows that a range of speed critical services/functions can be enhanced to be extremely robust with a conservatively attainable speed penalty of <5% for the synthetic application benchmarks. Iterative benchmarks show worst case slowdowns in pessimistic scenarios of 37%, but usually <5%. While it is not clear what an "average" case is, the synthetic benchmarks show that even the most lightweight 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 for the Linux functions treated, and can be found in Table 1. 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 overhead increases to an average of 50 ns for a 1/4 full cache, increasing to 90 ns for a cache at capacity. The base overhead increase is due to the necessity of adding a more complex indexing scheme capable of indexing arbitrarily sized caches. 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 10-25 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 [2], multiple branch prediction [32] and branch predication can be expected to effectively reduce exception checking 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 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.

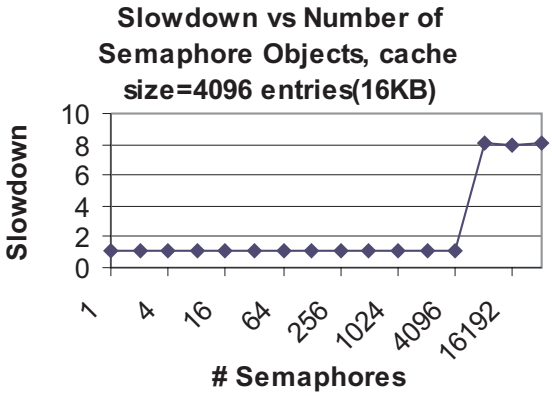

Figure 9. Performance of cached hardening for varying number of active semaphores.

6. Case Study

Robustness is not a concept addressed in many programming, or software engineering classes [30]. Too often the idea of testing software is linked to the idea that a system

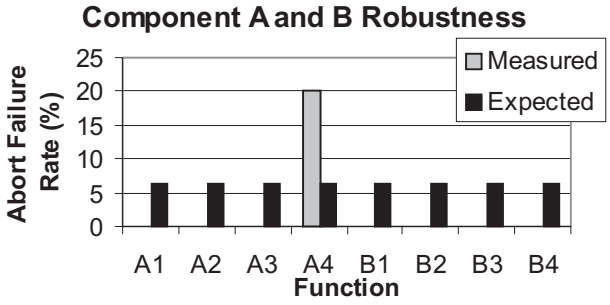

Figure 10. Measured vs. Expected robustness of components A & B.

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 resultanting from exceptional inputs or conditions, and exception detection and handling code tends to be the least tested and least well understood part of the entire software system [5].

While up to two-thirds of all system crashes can be traced to improperly handled exceptional conditions [5], the reason such failures occur is uncertain. 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 usual litany of excuses. While some of those possibilities have been discredited, others are a matter of opinion and can probably never be resolved. But perhaps there is another, more fundamental mechanism at work.

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 [34]. Thus the root cause of a robustness failure is often just that the exception checking and handling code is either designed or implemented incorrectly (commission), or simply omitted (omission).

In a 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 [30]. Maxion provided material to the experimental groups with information on exceptions, exception conditions, and a mnemonic to jump-start their thinking on the topic as well as help them to remember exception checking. He was able to show significant improvement in the exception handling characteristics of the treatment group software when compared to a control group.

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 section examines how well experienced developers understand the exception handling characteristics of their code, not necessarily how robust the code was. We look at a series of Java components written by various corporate development groups within one of Carnegie Mellon's large corporate research partners that is historically known for producing robust software. Data was collected on how the developers thought their code would respond to exceptional conditions, and contrasted with the robustness as measured by Ballista.

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 [30]. Only minor changes were required, in order to better fit the object and safety models that are inherent to Java.

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.

Expected failure rates were calculated from the data provided by the programming teams. The method used to compute expected failure rates was a simple correlation between which types of failures the developers expected to be handled improperly, and the incidence rate of the failures in the test cases. For example, let us consider hypothetical developer group X. Group X predicts that method foo() will suffer an underflow exception(and none others) under the right conditions. 10% of the test cases run against foo() would put it in those conditions. Thus we would report that Group X estimates a 10% failure rate for function foo(). Such an abstraction is necessary since the groups have no a priori knowledge of the test cases.

Results for testing components A and B can be found in Figure 10. Overall average failure rates were 5% and 0% respectively. Teams 1 & 2 did fairly well in accurately classifying their code. One method suffers from an abort failure due to invalid data (memory reference), and some

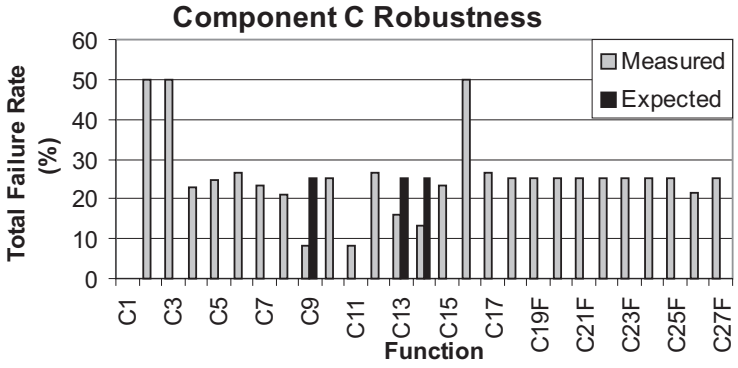

Figure 11. Measured vs. Expected robustness of component C. F suffix indicate methods that perform data formatting operations rather than performing calculations.

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 that were in fact caught by the language).

Figure 11 contains the test results for component C, with an overall average failure rate was 24.5%. 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.

Teams 1 and 2 had much better knowledge of 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.

Team 3 seemed to have more difficulty determining how robust their code was. 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 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.

7. Conclusions

We have determined that generic robustness hardening wrappers can be used to handle all detected robustness failures in several different software interfaces. Moreover, via the use of a software caching technique for robustness checks that maps well onto modern processor architectures, the amortized cost of exception checking can be made quite low, often only a percent or two of execution time. Thus, the scalable exception testing approach used by the Ballista robustness testing system has formed the basis of a scalable, generic, low-cost robustness hardening wrapper approach. Although in most cases we removed all the failures detectable by our system, this is by no means a claim that there are no residual failures. Rather it is an argument toward the effectiveness of techniques used to preemptively detect and correct conditions that would induce a failure without suffering large performance penalties or programmatic complexity. Based on this result, there seems to be little reason for developers to shun building robust systems based on performance concerns.

Further, we have measured the ability of a set of experienced, professional developers to accurately classify the exception handling abilities of their software systems. 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. This suggests that an effective way to improve robustness is to train developers in understanding and applying known robustness improvement techniques.

8. Acknowledgments

Support for this research was provided via an IBM Research Fellowship and a grant from AT&T Shannon Research Laboratory. Special thanks to P. Santhanam at IBM T. J. Watson Research Center for the joint research partnership.

9. References

[1] 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

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

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

[4] 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.25, no.2

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

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

[7] DeVale, J.; Koopman, P., "Performance Analysis of Exception Handling in IO Libraries," *Intl. Conf. on Dependable Systems and Networks*, 2001

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

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

[10] 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

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

[12] 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

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

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

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

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

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

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

[19] 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

[20] *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

[21] 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

[22] 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

[23] 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

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

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

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

[27] Leveson, N.G.; Turner, C.S., "An investigation of the Therac-25 accidents," *IEEE Computer*, Vol 26, No. 7

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

[29] 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

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

[31] Pan, J.; Koopman, P.; Huang,V.; Gruber, R., "Robustness Testing and Hardening of CORBA ORB Implementations," *International Conference on Dependable Systems and Networks*, 2001

[32] 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

[33] Alexandre Romanovsky, "An exception Handling Framework for N-Version Programming in Object-Oriented Systems," *3rd International Symposium on Object-Oriented Real-Time Distributed Computing*, 2000

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

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

[36] Vo, K-P., Wang, Y-M., Chung, P. & Huang, Y., "Xept: a software instrumentation method for exception handling," *The Eighth Intl. Symposium on Software Reliability Engineering*, 1997

[37] 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

[38] Ziles, C.B., Emer, J.S., Sohi, G.S., "The use of multithreading for exception handling," *32nd Annual International Symposium on Microarchitecture*, 1999