Performance Evaluation of Exception Handling in I/O Libraries

John DeVale   Philip Koopman
Carnegie Mellon University
Department of Electrical and Computer Engineering
Institute for Complex Engineering Systems
5000 Forbes Ave   Pittsburgh, PA 15213
devale@cmu.edu koopman@cmu.edu

Abstract

Lack of data quantifying the performance cost of implementing good exception handling often causes developers to skimp on exception handling based on its overestimated perceived cost. In an effort to remedy this problem we provide performance data on the cost of building good exception handling into software. We use the Safe Fast IO library as a basis for this study. SFIO improves robustness by a factor of 3 to 10 over STDIO without sacrificing performance. We were able to improve the robustness of the critical SFIO functions by another factor of 5, thus 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.

1. Introduction

Recent advances in the ability to measure software robustness have revealed that it is common for software to be non-robust when presented with exceptional parameter values. For example, both Unix and Windows operating systems and their C libraries tend to have significant robustness failure rates, with C library functions often being less graceful at handling exceptions than system calls [8][15].

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[15]. While the evidence does not prove causality, in many cases overall system failures tended to be caused by modules with poor overall exception handing characteristics [15][7].

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 [12]. Others eschew it because of perceived performance problems and development difficulty. Neither of these need be the case. As Maxion points out in [12], even a small amount of effort applied to raising the awareness of the importance of solid exception handling can result in significant improvements. Additionally, there are now several research and commercial tools to help developers detect potential robustness weaknesses [10][6][3]. But, beyond the issue of finding and correcting robustness problems, we believe that in general developers greatly overestimate the performance penalty of making software highly robust and use this as a reason to avoid robustness improvement.

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 [9]. 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 IO 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.

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 paper 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.

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 greater than STDIO (Figure 1), 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

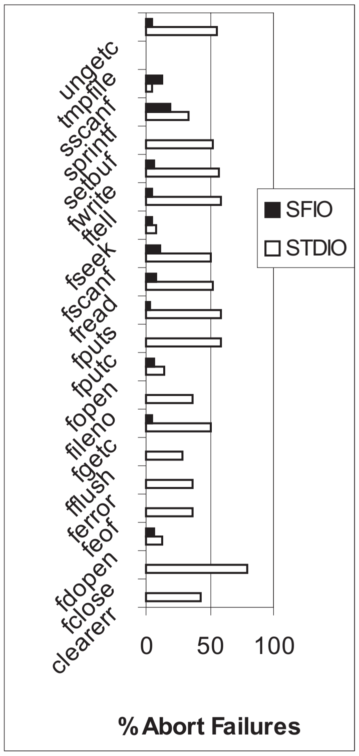

Figure 1. Robustness failure rates for SFIO, STDIO compared for 20 functions with direct functional equivalence, as measured on the Linux test system. SFIO failure rates on Digital Unix were lower for some SFIO functions and are addressed later in section 2

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 2 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.

While validating file parameters was fairly straightforward, validating the buffers and data structures was more difficult. 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.

We used the mechanisms described in [11] 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 2 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[4], 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% across the 8 modified functions.

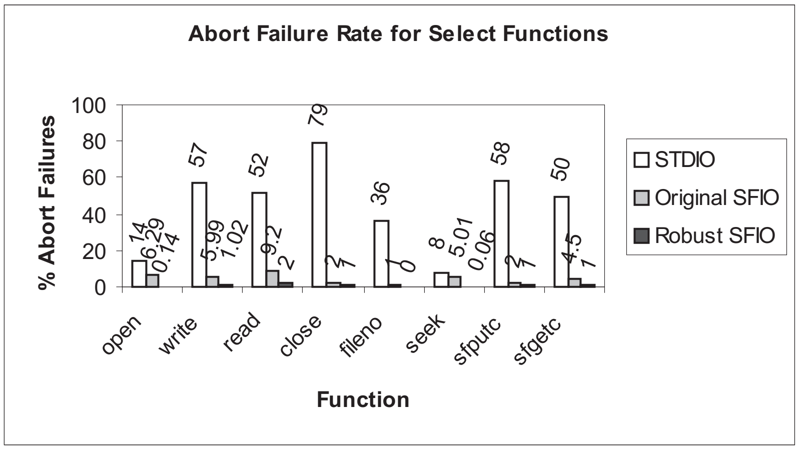

Figure 2. Abort Failure Rate for Select SFIO Functions under Linux

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 [17][1][18]). However, many of these techniques have a hefty performance penalty without their proposed architectural support to identify "bad data" situations. Thus, further robustness improvements will become practical only when they are supported by future generations of microprocessor hardware.

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 1) as described by the authors of the original SFIO [9]. The results presented are the averages from 10 benchmark runs and are presented in figure 3 (execution time variance across runs was minimal). 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.

Table 1 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 run 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 ques-

		File Size	
Benchmark Name	Description	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	reads random blocks, writes to position 0	1000MB	2000MB
Write	Writes	1000MB	2000MB

Table 1. Benchmark Descriptions

tions, 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 a 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 Linux is most commonly used as a workstation OS and as such is optimized more for latency and less for raw throughput. Finally, commodity PC hardware is extremely cost sensitive and tends to sacrifice 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 completed too quickly for accurate measurement.

In order to avoid large penalties for exception context initialization and checking each time through a tight IO loop, we applied a variation of optimistic incremental specialization [13]. We cached the most recent set of checks, and tested to see if the current data set was recently validated for a call that was non-destructive to its value. In such a case, we allowed program execution to bypass the exceptional condition checks.

Table 2 gives complete user and system level performance data for the original SFIO and the final robust SFIO with incremental specialization . Total process time is broken down into the user and system components as measured by libc function call time().

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 [5], 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.

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 opera-

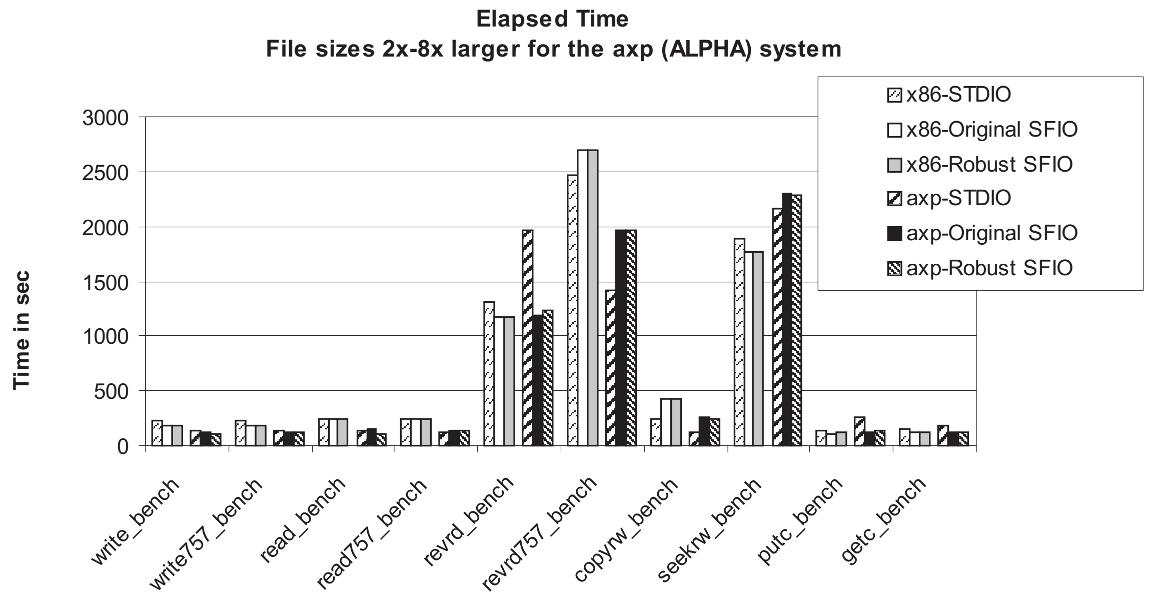

Figure 3. Benchmark Execution Time in Seconds - Programs from [9]

tions 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. After all, the time spent during IO wait can be used to perform other useful work in a multi-tasking system.

Figure 4 shows 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 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 increased robustness checks.

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 this penalty, 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[16]. 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.

Processors that use a trace cache[16], 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 [14] and block caches [2]. 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 processors incorporate 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.

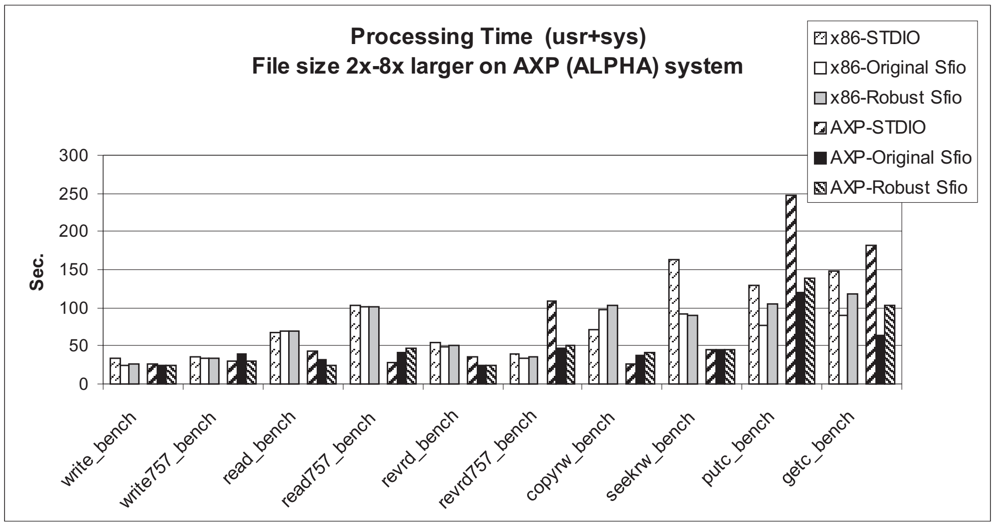

Figure 4. Processing Time

5. Conclusions

We used the Ballista robustness testing tool to find 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.

6. Acknowledgments

This work funded in part by DARPA under contract DABT 63-96-C-0064 (the Ballista project), and a fellowship from IBM Corporation. Intel based systems used in the research were provided by an equipment grant from Intel Corporation.

7. References

[1] Austin, T.M.; Breach, S.E.; Sohi, G.S., “Efficient detection of all pointer and array access errors,” *Conf. on Programming Language Design and Implementation* (PLDI) ACM SIGPLAN ‘94

[2] Bryan Black, Bohuslav Rychlik and John Paul Shen, “The block-based trace cache,” *Proceedings of the 26th annual Intl. Symp. on Computer Architecture*

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

[4] Fernsler, K.; Koopman, P., “Robustness testing of a distributed simulation backplane,” *Proc.10th Intl. Symp.on Software Reliability Engineering* ISRE 1999

[5] Griffin, J.L.; Schlosser, S.W.; Ganger, G.R.; Nagle, E.F., “Modeling and performance of MEMS-based storage devices,” *Intl. Conf. on Measurement and Modeling of Computer Systems* ACM SIGMETRICS ‘2000

[6] Hastings, R.; Joyce, B., “Purify: fast detection of memory leaks and access errors,” *Proceedings of the Winter 1992 USENIX Conf.*

[7] Koopman, P.; DeVale, J., “Comparing the robustness of POSIX operating systems,” *Twenty-Ninth Annual Intl. Symp.on Fault-Tolerant Computing*

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

[9] Korn, D.G.; Vo, K.-P., “SFIO: safe/fast string/file IO,” *Proceedings of the Summer 1991 USENIX Conf.*

[10] Kropp, N.P.; Koopman, P.J.; Siewiorek, D.P., “Automated robustness testing of off-the-shelf software components,” *Twenty-Eighth Annual Intl.Symp. on Fault-Tolerant Computing* FTCS 1998

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

[12] Maxion, R.A.; Olszewski, R.T., “Improving software robustness with dependability cases,” *Twenty-Eighth Annual Intl.Symp. on Fault-Tolerant Computing*

[13] 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 Symp. on Operating systems principles*, SIGOPS 1995

[14] Ryan Rakvic, Bryan Black and John Paul Shen, “Completion time multiple branch prediction for enhancing trace cache performance,” *The 27th Annual Intl.Symp. on Computer architecture* ISCA 2000

[15] Shelton, C.P.; Koopman, P.; Devale, K., “Robustness testing of the Microsoft Win32 API,” *Proceedings of the Intl.Conf. on Dependable Systems and Networks*. DSN 2000

[16] Eric Rotenberg, Steve Bennett and James E. Smith, “Trace cache: a low latency approach to high bandwidth instruction fetching,” *Proceedings of the 29th annual IEEE/ACM Intl.Symp. on Computer Architecture* ISCA 1996

[17] Wilken, K.D.; Kong, T., “Efficient memory access checking,” *The Twenty-Third Intl.Symp. on Fault-Tolerant Computing* FTCS-23

[18] Wilken, K.D.; Kong, T., “Concurrent detection of software and hardware data-access faults,” *IEEE Trans. on Computers*, vol.46, no.4 p. 412-24