A Tool for Investigating Empirical Data Structure Robustness

David J. Taylor
James P. Black

Research Report CS-82-02

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

January, 1982

ABSTRACT

Research in robust data structures can be done both by theoretical analysis of properties of abstract implementations and by empirical study of real implementations. Empirical study requires a support environment for the actual implementation. In particular, if the response of the implementation to errors is being studied, a mechanism must exist for artificially injecting appropriate kinds of errors. This paper discusses techniques used in empirical investigations of data structure robustness, with particular reference to a tool developed for this purpose at the University of Waterloo.

1. INTRODUCTION

The study of robust representations of data structures has been going on at the University of Waterloo for a number of years. It is convenient to think of the activity as having three main parts: general theoretical analysis, theoretical analysis of particular structures, and empirical studies. The empirical studies are intended to investigate the response of a robust representation to "random" damage, in contrast to the theoretical analyses, which consider worst-case results.

The theoretical analyses can largely be performed using only pencil and paper, but the empirical studies require that actual update, detection, and possibly correction, procedures be implemented. In principle, this implementation could take place in any convenient programming environment. There is, however, a practical problem: it is necessary to be able to introduce random damage into the structures being tested. This can be done in a variety of ways, but it is clearly preferable to build the "damaging" mechanisms only once and then make use of them in a variety of environments. For this reason, a tool (or rather, a small collection of inter-related tools) has been built to aid in performing the empirical studies. This tool has been used in obtaining previously reported empirical results [2, 4, 6]. The purpose of this paper is not to report results obtained using the tool, but to describe the tool itself. The intention is to give a more detailed description of how empirical results have been obtained, and to provide information which may be useful to others who are interested in investigating data structure robustness.

This tool, called IOSYS, was originally constructed to help answer such questions as: given a detection procedure which can detect any one or two errors, what is the probability that it will detect a set of three errors? However, one can also use IOSYS to increase confidence in the correctness of the procedure's implementation by running a number of one and two error cases. This is particularly useful with some correction procedures, which are so complex that extensive testing seems essential. Thus, IOSYS has also proved useful for testing fault tolerant software.

In the following sections, we describe the facilities provided by IOSYS and the underlying requirements of robust data structure testing which led to the creation of those facilities.

2. BASIC FACILITIES

IOSYS is, in a sense, an input/output system with certain added facilities. Because the "added facilities" were the whole purpose of the implementation, an effort was made to keep the usual I/O facilities as simple as possible.

Primarily as background for the following section, this section provides a brief description of the facilities provided by IOSYS when viewed as a standard input/output system.

Only one kind of file is provided by IOSYS: a fixed-length record, direct access file. Several files may be in use simultaneously with a different record length for each, but variable length records within a file are not permitted. This is a fairly restrictive environment but matches the requirements for our experimentation quite well.

Program access to files is provided by read and write functions. Files are simply identified internally by consecutive non-negative integers, so the read and write routines have a file number, a record number, and a buffer address as parameters.

During the "setup" phase of running a program under IOSYS, commands are entered from the terminal to attach the program to existing files in the "real" file system or to create new, empty files. Because many experiments are very I/O-intensive, IOSYS will normally keep small files in main storage rather than on disk. If desired, any file can be kept on disk, no matter how small the file, by explicitly requesting disk allocation. During execution of the user program there is no difference, except in timing, between a file kept in main storage and one which is really on disk.

To aid in debugging and monitoring programs, commands are provided to display records at the terminal and to activate a trace of input/output requests made during program execution.

The preceding is not a complete description of the facilities available, but does provide sufficient background for purposes of this paper.

3. MANGLING FACILITIES

In order to test the fault tolerance of software or data structures, it is necessary to have a mechanism which provides controlled simulation of the effects of faults. When studying robust data structures, the faults of interest are those which ultimately damage stored representations of data structures. Thus, a reasonable approach to the simulation of faults is to damage stored data.

In order to avoid building extensive knowledge of the data structures being used into the fault simulation mechanism, update routines can be used to guide the mechanism. Records (nodes) being accessed or updated seem logical candidates for being damaged. This consideration is the main reason that IOSYS deals with data structures which are (or at least, appear to be) on external storage: accesses and updates are forced to go through read and write function calls, thus making it easy for the fault simulation mechanism to observe such activity.

Records could be damaged (erroneously modified) when they are read or written, but since records which are being modified by an update routine seem more likely in practice to be modified erroneously, records are damaged only as they are written. At present two basic forms of damage are possible: modification of a single word in the record and refusal to write the record.

This facility for introducing damage is referred to as the "mangler." It exists as part of the IOSYS write function and is driven by a random number generator and parameters entered by the user. Since mangling is a central feature of IOSYS, a fairly detailed description of the implementation is provided here. First, there is a global control which allows the mangler to be turned on and off. Second, there is a mangle probability associated with each file. If the probability is zero, that file will not be mangled. If the mangler is on, the probability value specifies the chance of mangling on each write call.

When a mangle is to occur, the "mangle type" for the file determines what happens. Three mangle types change a single word, and two cause the record not to be written at all. The first three mangle types differ in how the word to be modified is selected: one uses a uniform distribution over all the words in the record, one uses a distribution skewed toward the beginning of the record, and one calls a user exit routine to obtain a list of "mangleable" words, from which a uniform random selection is made. Note that the last mangle type may be used to implement any arbitrary probability distribution, by selecting a single word according to this distribution and then reporting it as the only mangleable word. In each of these three cases, the record is modified by adding a value to the selected word. The value is selected uniformly from a user-specified range symmetric around zero. Adding a small quantity tends to produce more "subtle" changes than making an arbitrary replacement of the word.

The fourth mangle type simply refuses to write the record, if the probability test is satisfied. The fifth type ("crash" mangling) is intended to simulate a system crash during updating. In this case, IOSYS makes a transition from "up" to "crashed" with the specified probability. Once in "crashed" state, all writes to the file (and any other file with crash mangling specified) are refused. It is simpler to simulate crashes in this way than by attempting to abort the actual execution of an update routine. Naturally, an IOSYS call is provided to "uncrash" the system, in order to proceed to another experiment iteration, once the effects of the simulated crash have been analysed.

To allow mangling activity to be monitored, when a mangle takes place a message may be displayed at the terminal or a user exit routine may be called. The user program may request any combination of these.

4. OPTIONAL FACILITIES

The facilities described in the two previous sections are part of IOSYS itself. Modifying or replacing any of them must be done very carefully. There are other routines which work in conjunction with IOSYS, which are supplied so that code will not have to be duplicated in multiple application programs. These routines are intended to be used optionally, as required, and can be replaced by similar, user-written routines or ignored altogether, as desired.

There are presently three such packages of routines. One package provides a simple-minded free space management for IOSYS files. The other two are of greater interest.

The second package provides a "mangle table" capability. Although, in some sense, the damage done to a file must be kept secret from much of the program (for example, error detection routines clearly must not make use of such information), it is frequently important to keep a record of the mangles which have taken place. Because keeping and using such a mangle table is a non-trivial task, a package of routines is provided to do such things as: set up a mangle table, record a new mangle, find out the "true" (unmangled) value of a field, print the mangle table, and so on.

We want to make the distribution of mangles over the nodes of a structure realistic. One approach to this is to use the set of records written by an update routine as candidates for mangling rather than selecting records completely at random. This means that mangles to individual records are not independent, which seems desirable. However, using an update routine with the mangler active introduces a serious problem. If the update routine makes use of a field which has already been mangled it could propagate the damage in an unknown way, go into an infinite loop, or cause an abort. Designing update routines which will not do any of these things is an interesting problem, but in order to avoid solving the problem before performing any experiments, we wanted to use less robust update routines.

This leads to a three-file cluster for each logical file used by the program. One file is an unchanging master copy used to refresh the other two files. One of the other files is the target of actual updates, but is not mangled. The remaining file is mangled but not updated. Whenever the update routine writes a record to the update file, the corresponding record is read and rewritten in the mangle file. Since the mangler is active on this file, the write operation may result in a mangle. For efficiency, the complete update file does not really exist: only the modified records are kept in this file, a bit vector being used to indicate which records have been modified. A small set of routines is provided to handle this, so that the facility can be made conveniently available to the various different experiment programs.

5. EXPERIMENTS

While it is not the purpose of this paper to report the results of specific experiments performed using IOSYS, it is appropriate to describe the kinds of experimentation and testing which it has supported.

The first type of experiment for which IOSYS has been used is empirical detectability estimation. This is the kind of experiment alluded to in the introduction: if we know that some sets of three errors cannot be detected we would like to estimate the probability that such a set cannot be detected. To perform such an experiment we need to insert three errors, run a detection procedure, and repeat a large number of times. For any robust data structure with detectability greater than one, it appears that the probability of introducing an undetectable set of errors is essentially zero.

Errors could be inserted completely at random, but to produce a more realistic distribution of errors, a delete routine is used to select nodes to be written, and the mangler is engaged so that some writes cause erroneous modification of the record being written. To allow all of this to be done safely and efficiently, the three-file technique described in Section 4 is used.

A second type of experiment is a "connectedness check." In this kind of experiment, the objective is to determine empirically the probability of losing all access paths to any node (thus disconnecting the structure instance) for a given number of erroneous changes. As in the first type of experiment, a delete routine is used to guide error insertion. Then a "connectedness checker" is run which attempts to find an access path to each node. This process is repeated some large number of times.

In practice, the first two kinds of experiments are run in conjunction with each other. That is, some number of errors is inserted then the resulting damaged instance is checked for both apparent correctness and connectedness. A typical example is an experiment of 3000 iterations on a double-linked list: 16341 deletes were used overall to insert 3 mangles on each iteration, causing 1058 disconnections, but producing no undetectable errors.

A third type of experiment is intended to determine the effects of error propagation. In this kind of experiment, a script of insert and delete commands is executed with the mangler engaged. Detection and correction routines are also invoked periodically during the script execution. No measures are taken to make mangles invisible to update routines, so new errors may be introduced due to updates, and update routines may be blocked from performing any action because of encountering an error.

The objective in this kind of experiment is to determine the percentage of errors detected, percentage corrected, etc. To do this, a "mangle table" must be constructed containing data on mangles and corrections. Because of error propagation, data in this table cannot always be relied upon. Therefore, a copy of the file as it should be at the end of the script run (produced with the mangler turned off) is compared with the file actually obtained. Any differences not accounted for by the mangle table are noted. If some inserts or deletes were blocked, the number of differences noted can be very large, because a record by record comparison is not appropriate unless all updates were performed. In such cases, significant human effort is required to determine the actual number of errors detected and corrected. However, in many cases, the comparison finds no unnacounted-for differences and error detection and correction statistics produced by the program can be used immediately.

IOSYS has also been used for testing fault tolerant software. Of course, the various detection and correction routines used in the experiments described above are always used in "trial runs" with the mangler active, in order to remove implementation bugs. It is also possible to use IOSYS for testing complicated routines, such as some correction routines, whose behaviour cannot be characterised theoretically. An example is the single error correction algorithm for CTB-trees [3]. While it is known that any single error to a CTB-tree can be corrected, this particular algorithm is so complicated that proving anything significant about it seems impossible. Therefore, it was implemented and tested, first by hand insertion of "interesting" errors. This resulted in finding a number of bugs, and appropriate modifications were made to the algorithm. When no more bugs were found by hand insertion of errors, the mangler was used to create a large number of single error test cases. These cases, even though produced completely at random, made apparent a number of bugs not previously encountered, which were then fixed. Although we cannot guarantee that any single error will be corrected on the basis of this test, we are now much more confident that the correction routine will function properly.

6. SUMMARY AND HISTORY

Finally, a brief summary of its history and possible future developments may help to put IOSYS in perspective. EXSYS [1] was the first instance of attempting to investigate empirical robustness of data structures. To perform EXSYS experiments, *ad hoc* mechanisms were added to the code in order to produce the necessary random damage. The result was a workable but not very convenient system. Because of the effort which would be required to perform a conversion, the current version of EXSYS still uses these *ad* hoc mechanisms.

When other empirical testing was contemplated, it seemed clear that a more general, flexible tool was required. Therefore, the first version of IOSYS was implemented, on a Honeywell 6050, in a locally designed language, Eh. The tool proved very useful, and has been modified and extended, in order to make it more powerful and useful. For various local reasons, including dropping of support for Eh, the tool was also moved to UNIX*, and translated into C.

IOSYS has now been used to perform a large number of experiments on different data structures. Although creating, modifying, and maintaining IOSYS has taken a significant effort, the benefit in simplifying experimentation has easily compensated for this effort. It is our intention to go on using IOSYS for further experiments. We think IOSYS presently has a good set of facilities for our purposes, but some extensions will likely be required to meet future needs.

One direction in which work is presently proceeding is the creation of "interchangeable" storage structure implementations. These would not only have a standard interface to lower-level routines (in IOSYS), but would also have a standard interface with the higher-level routines which use them. One benefit of this will be that a single

*UNIX is a trademark of Bell Telephone Laboratories.

interactive command interpreter could be used with an arbitrary storage structure. (At present, each storage structure has its own slightly different command interpreter.) More importantly, having a standard interface to storage routines will allow them to be combined easily into more complex storage structures, such as compound structures [5].

It is hoped that the material presented here helps to explain how we have performed robust data structure experiments and will provide useful assistance to others who are investigating data structure robustness or testing fault tolerant software.

ACKNOWLEDGEMENTS

The development of IOSYS was possible only with the able assistance of several programmers: Bert Bonkowski, Mark Ingram, Ralph Hill, and Adrian Pepper. The design of IOSYS was also influenced by helpful discussions with David Morgan.

BIBLIOGRAPHY

1. Black, J. P., D. J. Taylor, and D. E. Morgan. A case study in fault tolerant software. Software--Practice and Experience, vol. 11, no. 2 (February 1981). pp145-157.

2. Black, J. P., D. J. Taylor, and D. E. Morgan. A compendium of robust data structures. Digest of Papers, The Eleventh Annual International Symposium on Fault-Tolerant Computing, Portland, Maine, June 24-26, 1981. pp129-131.

3. Black, J.P., D. J. Taylor, and D. E. Morgan. A robust B-tree implementation. Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, California. pp63-70.

4. Taylor, D. J., J. P. Black, and D. E. Morgan. Redundancy in data structures: Improving software fault tolerance. IEEE Transactions on Software Engineering, vol. 6, no. 6 (November 1980). pp585-594.

5. Taylor, D. J., J. P. Black, and D. E. Morgan. Redundancy in data structures: Some theoretical results. IEEE Transactions on Software Engineering, vol. 6, no. 6 (November 1980). pp595-602.

6. Taylor, D. J. Robust Data Structure Implementations for Software Reliability. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, 1977.