Surviving Software Failures

A survey on the research context of "Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures" by F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou.

Johannes Pletzer, 9920310

Software and System Seminar, SS2007, University Of Salzburg, Austria

ABSTRACT

This paper is a survey on approaches to survive software failures at runtime. After introducing different types of bugs described in literature, various methods to survive them are presented and discussed regarding their effectiveness and feasibility.

1. Introduction

In the early years of computing hardware faults were mostly responsible for the failure of a computer system. As hardware got more and more reliable over the years, the top cause for failures today are software faults or the bugs which cause them. It is clear that ways to cope with bugs at runtime would increase the reliability of applications. This is especially desirable for server applications where downtime typically leads to decreased productivity and subsequently to financial loss.

It is vital to understand why software fails at all and therefore in chapter 2 the major types of software bugs are introduced. Subsequently chapter 3 presents different approaches for surviving software failures. Chapter 4 concludes with a short summary.

2. Types of Bugs

Gray [3] identified two major kinds of software bugs: The first type are "hard" software bugs that always cause a failure when a specific code region of an application is executed. These bugs are also called deterministic bugs or Bohrbugs, a name inspired by the simple atomic model by Niels Bohr. Because such bugs can be reproduced rather easily, Gray argues that most of them are found and fixed during software development. The second type of bugs are bugs that are non-deterministic and occur in specific situations and environmental conditions. They are for example related to synchronization problems in multithreaded software. Gray called them Heisenbugs in analogy to the Heisenberg Uncertainty Principle. Such bugs cause "soft" failures and are similar to transient hardware failures and can also be cured by similar approaches than those used to cope with hardware failures, namely reboot or retry. Unlike Bohrbugs, Heisenbugs often disappear when the problematic code region is executed again as environmental condition such as the order of thread execution might be different.

Another special kind of bugs are aging-related bugs. Aging refers to changes during the runtime of an application. It has been observed that certain bugs or faults are more likely to manifest themselves the longer an application runs continuously. This phenomenon is called software aging. A typical example would be memory leaks that cause an application to run out of memory after a certain amount of time.

While Gray's hypothesis [3] is that most software faults are caused by Heisenbugs, Chandra and Chen [6] suggest the contrary. In their study of faults in Apache, GNOME and MySQL they found 72-87% deterministic Bohrbugs and only 5-14% transient Heisenbugs. Also another study by Sullivan and Chillarege [7] yielded similar results. This discrepancy can be explained by today's feature-driven software culture which puts features before reliability and also the fact that Heisenbugs are less likely to be reported as they mostly cannot be reproduced.

3. Approaches to Survive Software Failures

The different approaches found in the literature for coping with software failures range from simply removing the bugs that cause them to rather unconventional strategies such as manufacturing values to cope with illegal memory reads. The following sections try to categorize them and to describe a few significant contributions in detail while pointing out notable drawbacks and limitations.

3.1. Removing Bugs

The safest and most effective way to survive software failures is obviously to prevent them in the first place, i.e. to make the code as dependable as possible. Among these "proactive approaches" are the use of safe languages such as Java and compilers and code analysis tools that aid in the production of code with as few bugs as possible. During development and testing one should not forget that fixing bugs in the operational phase is estimated to be by a factor of 5 to 100 times more expensive than fixing them before [9].

3.2. Reboot/Restart Approaches

Inspired by techniques for surviving transient hardware failures, which can normally be coped with by rebooting the system, some approaches try to apply this also for tolerating software faults.

Software Rejuvenation [11] bases on the assumption and also observation in some applications that the performance of software degrades during its runtime and that software failures are becoming more likely to occur. A well known example of a software fault caused by software aging is the failure of the Patriot missile defense system in 1991 [10] where the inaccuracies in time measurement increased over the runtime of the system. Software rejuvenation requires that an application is restarted periodically in order to restore a clean internal state. Research in this field focuses on determining the optimal rejuvenation frequency and time instant, e.g. during phases when a server is idle anyway in order to reduce the cost causes by the downtime during the reboot.

Microrebooting [5] tries to avoid the potentially very long reboot times introduced by software rejuvenation. Rather than rebooting the whole application, only small components are rebooted upon failure. Microrebooting also allows rejuvenating a complete application without ever shutting it down completely. Because microreboots are very fast they can be invoked upon the slightest hint of a failure and can be masked from end users for example by buffering requests for a short amount of time in which the microreboot can be performed. A significant disadvantage of the microrebooting approach however is that a redesign of an application is necessary so that is consists of numerous small, loosely coupled components which store all important state externally in dedicated state stores. Using such state stores can

As Bohrbugs will occur again after a (micro)reboot, the above approaches only are able to tackle Heisenbugs. Consequently, their effectiveness relies on the fraction of software faults that are actually caused by Heisenbugs - a number that is rather disputed as discussed above.

3.3. Design Diversity Approaches

Design diversity refers to having multiple different implementations of a given functionality or application. The idea is that independent programming teams won't make the same mistakes i.e. the different versions won't contain the same bugs.

N-Version Programming [4] refers to a methodology which requires two or more software versions that implement a given initial specification. The development of these versions should be completely independent and may also incorporate the use of different programming languages and algorithms. The different program versions are then executed by an execution environment which runs multiple versions of program blocks concurrently. The results of these blocks is compared at certain points in time called cross-check points. At such a point a generic decision algorithm is used to compare the results - also called comparison vectors - in order to obtain a consensus result. The result can for example be determined by majority voting. The result is then passed on to the next program blocks for which again multiple versions exist. Note that the partitioning of the whole application into software units that provide comparable output needs to be part of the initial specification as well.

Recovery Blocks [12] is another approach based on design diversity. It requires that an application consist of a collection of code blocks or functions called recovery blocks. Each recovery block contains a primary block, an acceptance test, and zero or more alternate blocks. The primary block contains an implementation of the desired functionality the block should execute. After execution the acceptance test determines whether the primary block produced correct results. If the acceptance test fails an alternate block is executed. This is repeated until either the acceptance test completes successfully or there are no more alternate blocks left.

Both design diversity approaches are very expensive as they require the development of multiple versions of an application. Therefore they are only used for systems which required exceptional high dependency such as flight control or train switching systems. They are however able to avoid a certain number of Bohrbugs as it is indeed not very likely that the independently developed versions contain the same bugs.

3.4. Rollback & Recovery Approaches

Rollback & recovery approaches base on the idea of rolling back an application to a recent checkpoint - a snapshot of the complete state of a program - and then retry execution from there again.

Lowell, Chandra and Chen [2] explored the limits of general application recovery by rollback and re-execution approaches. General recovery refers to recovery that is performed by the operating system and is transparent to the user and does not require any changes in the application software. The idea is that the operating system generates the illusion of failure-free operation. They identified two invariants that must be fulfilled in order to allow the successful recovery from software failures called Save-work and Lose-work. Save-work refers to the requirement that an application must store enough state so that the user is not exposed to a failure whereas Lose-work states that on the other hand sufficient state must be lost to prevent the repeated manifestation of a failure which makes recovery impossible. Their findings are that for stop failures - failures caused by application-external factors which make it stop abruptly for example because of a power failure - failure transparency is possible as only Save-work must be upheld. For propagation failures - where an erroneous state of the application is involved - the Save-work and Lose-work invariants often directly conflict and therefore recovering from such failures can only work with the help from the application itself.

Note that Bohrbugs cannot be survived by simple rollback & recovery approaches as they will manifest again during a repeated execution. An innovative re-execution methodology called Rx [13] tries to overcome this limitation. The idea is to rollback an application to a recent checkpoint upon a software failure and to re-execute it under a modified environment. This is inspired by allergy treatment in real life which includes removing the allergen from the environment. After passing the problematic code region the original environment is restored again as the environmental changes may introduce performance penalties. The environmental changes employed by Rx range from memory-related changes such as zero-filling newly allocated memory and padding allocated memory blocks to scheduling and message order changes and the dropping of user requests made to a server application. The memory management changes do overcome certain deterministic bugs - and therefore Bohrbugs - such as uninitialized reads and buffer overflows. The scheduling changes can tolerate certain Heisenbugs related to data race conditions. Dropping user requests may help against malicious requests made to a server while other requests are still processes correctly.

Rollback & recovery approaches rely on an efficient checkpointing algorithm as during runtime checkpoints must be made frequently all the time. Every single checkpoint needs to store the complete state of a running application. Discount checking [14] is such a low-overhead checkpointing method that is for example also used in the Rx system. It is built on reliable main memory and high-speed transactions. Reliable memory is provided by the Rio file cache [15] which protects memory from operating system crashes. Vista [16] builds on Rio to provide fast transactions which are used to allocate areas of persistent memory and to perform atomic, durable transactions on it. Transactions and checkpointing are equivalent as the interval between checkpoints is equivalent to the memory locations a transaction changes and taking a checkpoint is equivalent to committing the current transaction. Using additional optimizations Discount Checking manages to only slow down applications by 0.6%, even with very frequent checkpoints.

3.5. Speculative Approaches

Some recent approaches rely on speculative fixes for bugs that do no longer guarantee the original functionality of the application.

One such approach is the Reactive Immune System [1]. Its aim is to create "self-healing" software programs by localizing failures, detecting failures in the problematic code region and letting the function containing the region return an error value. This is done by running the faulty code region with an emulator and analyzing every instruction taking into consideration the reported cause of the fault, e.g. a division by zero. Then the function containing the fault is forced to return an error return value which is determined by some heuristics based on the return type of the function, e.g. -1 upon an integer return type. The approach relies on proper error handling of the application which according to the authors is mostly the case. Running the whole application in this "supervised" fashion would cause an enormous slowdown, but doing so for small code regions results in performance penalties of 30 to 100%.

Failure-oblivious computing [8] enables server applications to execute without memory corruption despite of erroneous memory access. It is a special C compiler that detects invalid memory accesses and replaces them with code that ignores invalid writes to memory and returns manufactured values for illegal read accesses. The idea is to avoid memory corruption in server applications triggered by malicious requests from clients and let the server's error handling logic deal with them which typically results in rejection of the problematic requests. This prevents the server from crashing and from entering dangerous execution paths and overall leads to increased availability and robustness. The major drawback of this approach and similar speculative approaches is that it is possible that because of the changes introduced by them the application may take an unanticipated and unintended execution path leading to unacceptable results. The authors argue that a software fault leads to such a behavior anyway and that nevertheless it is worth a try to attempt recovery by such methods.

4. Conclusion

The most effective way to survive software failures still seems to be preventing bugs in the first place and the use of "safer" languages such as Java. Note that for example the Bohrbugs that are potentially avoided in the Rx system and the failure-oblivious computing approach are all memory related. These bugs would not occur when using the Java programming language which has a much safer memory abstraction than for example the C programming language.

There seems to be a tendency to reference the right study that shows a specific distribution ratio of Heisenbugs and Bohrbugs so that the authors can argue that their approach to surviving failures is feasible. Probably it would be fairer to admit that most Bohrbugs cannot be survived safely and the frequency of them in applications varies significantly, also depending on how much effort was put on removing them during software development.

Nevertheless generic recovery techniques seem to be interesting for Heisenbugs and especially for server applications that demand high availability. With adequate rollback mechanisms and additional strategies such as dropping malicious user requests that crash the server they can help to keep up at least degraded operation with only a small time of service interruption.

References

[1] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In USENIX Annual Technical Conference, 2005.

[2] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation, 2000.

[3] J. Gray. Why do computers stop and what can be done about it? In Proc. Fifth Symposium on Reliability in Distributed Software and Database Systems, 1986.

[4] A. Avizienis. The Methodology of N-Version Programming. In Software Fault Tolerance, John Wiley & Sons Ltd, 1995

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot-- A Technique for Cheap Recovery. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), 2004.

[6] S. Chandra and P. M. Chen. Whither generic recovery from application faults? A case study using open-source software. In Proc. International Conference on Dependable Systems and Networks, New York, NY, 2000.

[7] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. Digest 21st International Symposium on Fault-Tolerant Computing, 1991.

[8] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing server availability and security through failure-oblivious computing. In Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI), 2004.

[9] B. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. IEEE Computer, 34(1):135--137, 2001.

[10] M. Grottke and K. Trivedi. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. IEEE Computer, pp. 107-109, February 2007.

[11] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In Proc. 25th International Symposium on Fault-Tolerant Computing, Pasadena, CA, 1995.

[12] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, B. Randell. A Program Structure for Error Detection and Recovery. International Symposium on Operating Systems, 1974.

[13] F. Qin, J. Tucek, J. Sundaresan, Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. Symposium on Operating Systems Principles, 2005.

[14] D. E. Lowell and P. M. Chen, Discount checking: Transparent, low-overhead recovery for general applications, Tech. Rep. CSE-TR-410-99, November 1998.

[15] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1996.

[16] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.