Heisenbugs and Bohrbugs: Why are they different?

March 8, 2003

Abstract

Jim Gray proposed a classification of bugs based on the type of failures they induce. Bohrbugs are bugs which always cause a failure when a particular operation is performed. Heisenbugs are bugs which may or may not cause a fault for a given operation. It has often been questioned as to whether these two bug categories are the same or different. The main reason for the argument is that if everything is the same, a bug should resurface and then by definition a Heisenbug should be the same as a Bohrbug.

We take the position that Heisenbugs and Bohrbugs are different types of bugs. As argument, we try to clarify the definition of Heisenbugs and Bohrbugs and explain what an "operation" really means in context of the definitions. Although Heisenbugs could resurface if the conditions which lead to the failure reappear, we show that it is very difficult to recreate the conditions which existed when the fault first happened.

After giving evidence in our favor, we explain how to debug software systems in the presence of Heisenbugs.

1 Introduction

J. Gray [1] put forth the hypothesis that bugs occurring in a computer system can be classified into two types: Heisenbugs and Bohrbugs. The basis for the classification was the ease with which the failure produced by the bug could be repeated. If a Bohrbug is present in the system, there would always be a failure on retying the operation which caused the failure. In case of a Heisenbug, the error could vanish on a retry.

Bohrbug was named after the Bohr atom. Just as the Bohr atom is solid, a Bohrbug is also solid and easily detectable by standard debugging techniques. The word Heisenbug comes from Heisenberg's Uncertainty Principle which states that it is fundamentally impossible to predict the position and momentum of a particle at the same time. If we try to apply standard debugging techniques such as cyclic debugging [5] to catch the error, we may find that the error has disappeared. Bohrbug is also called a permanent fault while Heisenbug is called as a transient or intermittent fault [2]. Note that inspite of the name, in both cases the fault is very much there at all times.

According to J.Gray, most industry software systems are released after design reviews, quality assurance, alpha, beta and gamma testing. They are mostly free from Bohrbugs which easily get caught as they are solid, but Heisenbugs may persist. As a result, most software failures in software systems are soft i.e. if the software system is restarted it would function correctly.

2 Software System and its environment

It is very important to understand what a software system and its environment consists of, before dealing with the subject of bugs. A process

consists of

1. program
2. data
3. stack

In addition, the process depends on a number of external factors for its faultless execution, to name a few:

1. Hardware
2. Operating System Code and Data structures
3. Other programs which it synchronizes with

The initial set of items form the software system, while the later set constitutes the environment of the system.

We should note that for a given program the state of the first set of items only depends on the input to a program. By input, we mean the parameters to a program or the data values accepted as input via I/O. Given a program and initial state of data and stack areas, for some input, the program should act so as to move the data and stack state to a predictable state. This can be easily inferred from the program correctness rules by [12].

However, we cannot assume that the evironmental factors are the same every time the program is run. The OS code might do process scheduling, memory management differently each time the program is run, the hardware might have changed, other programs might have changed their behavior.

3 The Argument

It has often been debated in academic circles as to whether Heisenbugs and Bohrbugs are the same. The fundamental reason for the debate is that the classification scheme uses failure as a means of classifying the fault. If we could get the failure to occur again, all Heisenbugs would be Bohrbugs.

We do not consider this line of argument to be correct and take the position that Heisenbugs and Bohrbugs are distinct. In section 4, we will explain the main arguments in favor of classifying Heisenbugs and Bohrbugs together as a single category of bugs. Sections 5, 6, 7, 8 refute those arguments and explain why we need the two categories. Conclusion and implications of our position are present in section 9.

4 Counterclaims

It is claimed that Heisenbugs are same as Bohrbugs since it is theoretically possible to recreate a failure caused due to a Heisenbug. If we can create the exact environment that was present during the execution of the program the first time and provide the same input, the failure should occur again and hence according to definition, the bug should be a Bohrbug.

This argument is based on the fact that computer systems are discrete hence there are only a finite number of states possible for a computer system. This effectively means that if we could get the computer back into that state which caused the error to happen, we could make the failure to reappear again.

Another argument is that for a Bohrbug to happen, we must perform the same operation on the program. The operation being external to the program is also an environmental factor. Therefore, both Heisenbug and Bohrbug can appear or disappear based on whether the environmental factors which caused the error are still present, so there is no difference between the two.

We do not agree with the above statements. The first argument is no good because although software systems are discrete there are many factors such as timing of asynchronous operations and clock drift that come into play so that it is not possible to recreate exactly the same conditions

once again. The second argument strongly depends on the definition of the two kinds of faults and we think the key to the argument is the word "operation". An operation is the input provided to the program. As we have told before, we consider input to be parameters to methods and data values received as input through IO. It should not be mixed up with any of the other environmental factors. We therefore think that Heisenbugs should be distinct from Bohrbugs.

5 Second Law of Thermodynamics

According to the "Second Law of Thermodynamics", the entropy of a closed system increases with time [6]. This means that the state of disorder in the system increases with time. If a program is executed once, there is an increase in the entropy of the system already. By system here, we mean the program plus its environment. The second time a program executes, even if we try the same operation, the disorder in the system is higher.

The entropy is a measure of the probability of a particular state. This means that the state in which the system exists is more probable than what it was earlier. The present system has less information in it as compared to the previous state. If we want to retain old information, we will need to record it separately. In case of a computer system, this would require special logs.

For example, if a disk drive has a disk platter which rotates and a drive head which moves over the head. It is quite possible that due to mechanical motion, the platter or the head has wear and tear and it loses mass and the operation of the disk will not be the same as before. Similarly, the keyboard, the mouse and other mechanical components may face slow wear and tear and change in behavior.

It may therefore be impossible to get the same hardware environment as before. Although, one could argue that we could replace the hardware, but the problem still exists. The new hardware need not be similar to the old hardware (when it was younger).

6 Asynchronous Operations

Although computer systems operate on the basis of a system clock, many operations in the computer are not synchronized to the system clock. Most of the peripheral devices use their own clock or are not clocked at all. This means that processes and the corresponding peripheral device operation may not be synchronized. This tremendously increases the number of states a computer system can be in, as the number of combinations of the state of the process and the state of the asynchronous operation can be large.

Also, in case of unclocked operations, the concept of discrete states cannot be used and there could be infinite possibilities. The time required to complete an unclocked operation is measured in pure real time (not in clock time) which is a continuous quantity. These asynchronous operations create uncertainties regarding the working of a computer system particularly in the scheduling part. These uncertainties in time can then translate to disappearing bugs.

6.1 Example 1 - Disk Drive Operation

The disk drive of a computer have variable seek time. If we run the program one time and then restart the system and rerun the program a second time, due to the variability of the seek time, for the same disk accesses coming in, we might get a different sequence of disk accesses.

Processes are normally blocked if they need a page from the disk. Since, the disk access sequence is not the same, the processes may remain blocked or unblocked for variable amount of time and so the processes may not all be scheduled in the same sequence as before. This means that a race condition that happened before may not happen now.

6.2 Example 2 - Network Characteristics

There are a large number of programs which use network connectivity. The traffic characteristic and load on the network cannot be duplicated in order to retest the system. Suppose packet 1 was originally lost once, in the second run also we must also lose it once. The delays encountered must be similar. On a lossy, best-effort networking infrastructure, this is a hard problem.

6.3 Example 3 - Software Clock Drift

The software clock of a computer system is kept up to date by use of high priority interrupts. Periodically, an interrupt is generated and the Interrupt Service Routine(ISR) for that Interrupt increments a register keeping track of the time. If the interrupt is masked or in general, the interrupts are disabled, the ISR will not be called and the ISR will not execute causing the clock to lose time [13].

This might affect applications which use current time to perform some work. For redoing an operation, we cannot just take the clock back and restart, clock drift will not allow the application to run the same way as before. A random number generator, for example, uses the current time as its seed. It is possible that during retry (even after pushing back the clock), a different seed was obtained since the clock time when srand() was called will not match the previous case and that will give an entirely different sequence of random numbers.

6.4 Example 4 - Processor Clock Drift

The processor clock depends on the crystal used. The crystals are normally precise in their frequency ratings but poor in accuracy. This means that the processor clock should be stable, however the frequency of these clocks strongly depend on the temperature and other environmental factors [13]. (environment here means the weather and other such conditions and is not the same as the the software environment mentioned earlier) Depending on temperature, the clock can run at a different rate and consequently scheduling could be altered, causing a previously present error to disappear.

We can give an example as to how asynchronous operations can affect the correctness of a system. Let us assume that a system has non-blocking read operation. It is possible that if we do a non-blocking read instead of a blocking one by mistake, the buffer may not filled in at all. However, if we give more time to the system (lot of processes are running), the buffer may actually be filled and the bug just disappears [14].

This section thus explains how clock frequency and asynchronous operations distort the image of a computer system as being truely discrete. In the next section, we look at effects caused by trying to debug the system.

7 Probe effects

Cyclic debugging techniques are normally used for debugging software. In this technique, on a failure, the code is attached to debugging utilities and a rerun is performed. However, the very fact that debugging tools have been attached to the program may cause the programs evironment to change. For example, if the debugger and the program runs in the same address space, the amount of memory [7] might be affected. Also, since an additional process is added, process scheduling might also be altered. This might lead to disappearance of the failure.

We should note that we cannot do anything to remove this kind of probe effect and to eliminate the uncertainty introduced due to asynchronous operations. Doing so would require 100% precise deadline scheduling which has not been performed to date. Deadline scheduling is needed since we want to strictly control how the programs run.

8 Classification Criteria

It is really interesting to know what is the distinguishing criteria between Heisenbugs and Bohrbugs. Classification of faults based on kind of failures that occur might not be the right classification. We need some intrinsic property of Heisenbugs and Bohrbugs which can separate the two.

Jim Gray gives the possible Heisenbugs. They could be:

1. Strange hardware conditions (rare and transient hardware failure)
2. Special Limit conditions (out of storage, counter overflow, lost interrupt, etc)
3. Race conditions

Carefully analyzing these faults, it is quite clear that Heisenbugs are caused due to interaction between conditions which occur outside the program being debugged and the program itself. The following can lead to Heisenbugs:

1. Uncertain environment (interrupt lost)
2. Wrong assumptions about the environment (memory is unlimited, so use and never free)
3. Wrong assumptions about the interaction between various subcomponents of a system (Program A and B use resource R, however when A is being written, we forget completely about B)

On the other hand Bohrbugs must be caused by wrong specifications about the system itself or mistakes in converting the specifications to a design. We can conclude this since a Bohrbug always recurs on retrying the operation. This is possible only if the designer or programmer coded up the system behavior incorrectly for that particular operation.

It could still be argued that since Bohrbugs are caused due to wrong code for some input while Heisenbugs are caused due to wrong code for some environmental conditions and since inputs to a program are part of the environment of the program, Heisenbugs and Bohrbugs are the same. This reasoning although correct only forces us to refine the definition of Heisenbugs and Bohrbugs.

Heisenbugs are errors in the program's assumption about the environment (except the input values) and the interaction between the subcomponents. Bohrbugs are errors in the program's actions based on its input values.

Thus we see a clear distinction between Heisenbug and Bohrbug.

9 Conclusion

These arguments clearly indicate that Heisenbugs and Bohrbugs are distict types of bugs. The difference is contingent on two important observations. The asynchronous nature of computer processes and the definition of what constitutes an operation.

The next question that should be answered is, in face of this uncertainty present, what strategies should we use to detect and eliminate bugs? We can take the analogy from Physics inorder to answer this question. On account of Heisenberg's uncertainty principle, we cannot predict the position and momentum of the particle at the same time. However, we can give a good probabilistic estimate of where the particle could be and what momentum it could have. Actually, In [8], the author makes a case for probabilistic modelling of software errors.

In case of a Heisenbug, although we cannot repeat the failure by retrying the same operation, we could get a good probabilistic estimate of where the error should be. We can perform a number of operations which exercise different components of the system. Errors happen some time or they do not, but by using a large sample space, we should get good probabilistic estimate of which component should be faulty. This idea

has been proposed and implemented in [9].

Some may find statistical arguments hard to swallow, so other strategies are in use against Heisenbugs. We could simulate execution of the software system [10]. In a simulated environment, everything is under control of the simulators. The environment is repeatable and some Heisenbugs can be caught. However such a simulator may not catch all errors because the environment posed by the simulator is somewhat different from the real world environment.

Another way of handling Heisenbugs is to introduce fault tolerance. In fault tolerance based schemes, the error is ignored and a new instance of the software system or component carries on the task. The idea is that if one instance of the system fails, the other could continue since its environment may not lead to a failure. Process pairs, N-version programming, etc [1] aid against Heisenbugs. Also, J. Gray[1] has given a rollback based scheme in which the occurrence of an error lead to retry after rollback. Since the environment must have changed the error, most probably the error would not repeat. [11] proposes a scheme in which debugging is combined with fault tolerance. If a fault occurs, a new error-free version of the component replaces the old component. The paper describes how to perform this while the system is online.

Newer debugging techniques and fault tolerant schemes thus aid programmers against Heisenbugs.

Acknowledgements

I am grateful to Rich Martin for his valuable advice on this paper. Many thanks to Neeraj Krishnan and Vishal Shah for discussing the topic with me.

References

[1] J.Gray, Why do computers stop and what can be done about them?, Tandem TR 85.7

[2] M.Hiller, Software Fault-Tolerance Techniques from a Real-Time Systems Point of View - an overview, TR 98-16 Department of Computer Engineering, Chamlers University of Technology, Sweden

[3] P.Bepari, Distributed Process Management Protocol: A Protocol for Transparent Remote Execution of Processes in a Cluster of Heterogeneous Unix Systems, M.Tech Thesis, Dept. of Computer Science and Engineering, IIT Kanpur, April 1999

[4] M.Ronsse, K. De Bosschere, RecPlay: A fully Integrated Practical Record / Replay System, ACM Transactions on Computer Systems, May 1999

[5] B.Miller, J.Choi, A Mechanism for Efficient Debugging of Parallel Programs, SIGPLAN 1988

[6] E.Lieb, J.Yngvason, The Physics and Mathematics of The Second Law of Thermodynamics, June 1997

[7] V.Paxson, A Survey of Support For Implementing Debuggers, CS262, University of California, Berkeley, 1990

[8] D.Hamlet, Foundations of Software Testing: Dependability Theory, SIGSOFT, 1994

[9] M.Chen et al, PinPoint: Problem Determination in Large, Dynamic Internet Services, International Conference on Dependable Systems and Networks, Washington D.C., 2002

[10] L.Albertsson, P.Magnusson, Using Complete System Simulation for Temporal Debugging Of General Purpose Operating Systems and Workloads, MASCOTS 2000

[11] D.Gupta, P.Jalote, Increase Software Reliability through Rollback and On-line fault repair, Department of Computer Science and Engineering, IIT Kanpur

[12] C.A.R.Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol 12, Issue 10, October 1969

[13] http://www.beaglesoft.com/Manual/page75.htm

[14] http://c2.com/cgi-bin/wiki?edit=HeisenBugs