M.I.T. Laboratory for Computer Science

Request for Comments No. 246
April 4, 1983

Checkpoint Debugging Thesis Proposal

by Wayne C. Gramlich

The following thesis proposal has been submitted to the EECS dept.

1. Introduction

Recent experience with distributed systems at the Laboratory for Computer Science has indicated that conventional interactive debuggers are not very effective for debugging many programs in a distributed system. The programs in a distributed system that seem to be the most difficult to debug are programs that have real-time constraints in addition to being run without continuous human supervision. These programs are called servers. An example of a server is a printer server that takes files from various nodes on the distributed system and prints them on a printer. Examples of other servers are file servers, phone directory servers, and data encryption servers. The real-time constraint makes it difficult to debug a server because when the programmer stops the server to examine some internal state, the subsequent behavior of the server may be affected. The lack of human supervision means that when a failure is encountered there is no one around to debug the program. Thus, it is necessary that new debugging techniques need to evolve to deal with debugging distributed systems programs, particularly servers.

This thesis proposes to investigate a debugging methodology called *checkpoint debugging.* Basically, checkpoint debugging works by taking regular checkpoints of a program. A checkpoint consists of a fixed part and incremental part. The fixed part of a checkpoint consists of a single consistent snapshot of the relevant program state. The incremental part of a checkpoint consists of a sequential recording of all program input since the time of the program snap-shot. When a program failure occurs, it is possible to use the checkpoint information to deterministically repeat the failure as many times as necessary to locate the program failure. This is done by going to a previous checkpoint, loading the fixed part, and reexecuting the program using the incremental part for program input. The major advantage of checkpoint debugging is that it converts a large class of non-deterministic failures (i.e. non-repeatable) into deterministic failures (i.e. repeatable). It turns out that it is much easier for a user to locate and correct a deterministic failure than a non-deterministic failure.

2. A model of debugging

It is important to realize that software is debugged by *people.* A wide range of tools have been developed to aid the programmer in locating software errors, but it is still the programmer who is ultimately responsible for the location and correction of program errors. The remainder of this section will develop a fairly simple model of how programmers debug software. It can not be

over-emphasized that this debugging model is very simple and and is only meant to identify the important aspects of debugging.

It is useful to define some terminology before continuing. A *failure* has occurred when the actual program output differs from the expected program output. A *bug* is an incorrect piece of software (or hardware) that can cause a failure. It is useful to further sub-divide failures into deterministic and non-deterministic failures. A *deterministic failure* is a failure that can be repeated upon demand by the user. Conversely, a *non-deterministic failure* is a failure that is not repeatable by the user. Failures that occur in programs whose output depends only upon file input (i.e. batch-oriented programs) are almost always deterministic failures. Failures that occur in real-time or interactive systems are frequently, but not always, non-deterministic failures. Whether a failure is deterministic or non-deterministic is independent of whether the program is deterministic or non-deterministic. In particular, a program that is basically non-deterministic can still encounter a deterministic failure. Conversely, a deterministic program can encounter a non-deterministic failure in the presence of non-deterministic hardware failures. However, in the absence of hardware failures, a 100% deterministic program can not have a non-deterministic failure.

The debugging model consists of four steps that are repeated until the program is "debugged".

1. The program failure must be detected.

2. Some person must locate the cause of the failure.

3. Some person must alleviate the cause of the failure.

4. An attempt should be made to verify that the failure will not occur again.

These steps have to be performed in order with the possible exception that sometimes the last step can be skipped. A good programmer will rarely skip the last step. This model does not disallow parallelism. Thus, there is nothing to prevent the user from trying to correct many failures simultaneously. Each of these steps will be discussed in more detail in the following paragraphs. Particular emphasis will be placed on the differences between debugging programs that experience deterministic failures as opposed to non-deterministic failures.

Detecting a failure is most often accomplished by a person, although failures are sometimes detected by special hardware or software. The person most often notices the failure by observing

that the program output is either incorrect or incomplete. Sometimes inadequate performance or response time is considered to be a failure. It is frequently the case that the person that detects the failure is not the person who wrote the program in the first place. System software such as compilers, linkers, and editors are typical examples of programs where failures are detected by the user and not the programmer. In these cases, information concerning the failure must be reported to the software maintainer. It is not infrequent that inadequate information is forwarded to the maintainer. The detection of a non-deterministic failure can be significantly harder to detect than a deterministic failure. This is due to the inability of the user to reproduce the failure upon demand.

Once the failure has been detected it is necessary for the software maintainer to isolate the cause for the failure. Generally speaking, the cause of the failure can be either bad input or a program bug. More rarely, the failure is caused by faulty hardware or a faulty system program, such as the compiler, linker, or operating system. Failure isolation is performed by repeatedly making a hypothesis as to the failure cause and then performing one or more experiments on the hypothesis to prove or disprove the validity of the hypothesis. Once a hypothesis has been validated, the programmer has isolated a cause for the failure.

There is no set rule as to how the programmer will go about forming and testing a hypothesis. Sometimes the hypothesis is generated by an examination of some combination of the program text, the program input, and the program output. Other times, the programmer has no idea and examines some fraction of the intermediate state of the program. In this case, the hypothesis is "maybe these variables have something to do with the failure". The hypothesis is tested in variety of ways. It is frequently useful to examine the intermediate state of the program when testing a hypothesis. This can be done with either a debugger or the addition of "print" statements to the code.

The programmer's approach to isolating a failure will vary considerably depending upon whether the failure is deterministic or non-deterministic. It is relatively easy for the programmer to test a hypothesis for a deterministic failure by simply rerunning the program. It is much more difficult to test a hypothesis for a non-deterministic failure since there is no guarantee the failure will occur again. Non-deterministic failures that occur very infrequently are currently very difficult to isolate.

Some combination of logs, traps, and post-mortem dumps are usually used to track down a

non-deterministic failure. A *log* is a file that is generated by "print" statements that the programmer has added to the program being debugged. These "print" statements display some portion of the intermediate program state that the programmer thinks might aid in failure isolation. When the failure occurs, the programmer can examine the end of log to see whether the log contains any anomalous information. A *trap* is a piece of code that the programmer adds to a program to help detect a failure. The primary purpose of a trap is to try to detect a failure at a earlier time than it would otherwise be detected. A *post-mortem dump* is a file that contains a raw unprocessed picture of the program as it was when a failure was detected. The analysis of *post-mortem dump* is an art-form, since the programmer must sift through an enormous amount of information to find the cause for the failure. Also, there is no guarantee that the dump will contain enough information to determine the cause for the failure. None of these techniques are terribly satisfactory for finding non-deterministic failures. These reasons tend to make finding the cause of a non-deterministic failure more difficult than finding the cause for a deterministic failure.

After the programmer has isolated the cause for the failure, the programmer must fix the cause for the failure. As with failure isolation, there is not set procedure the programmer follows. If the failure was caused by bad input, the programmer will tend to fix the failure by either changing the input (if permissible) and/or by changing the program to accept the unmodified input. If the failure was caused by faulty hardware, the hardware is either fixed or the program is changed to work around the faulty hardware. In the most common case, where the failure was caused by an incorrect piece of code, the code is modified. While there are plenty of tools available to help the user perform a program modification, there are no tools to aid the programmer in deciding what needs to be modified.

After the cause for a failure has been fixed, it is desirable to verify that the fix actually works and does not introduce more errors. For deterministic failures, this is fairly easy. The programmer need only rerun the modified program with the same input and verify that the failure no longer occurs. For non-deterministic failures, the programmer does not have this ability. At best, the programmer can simply install the modified program and see whether the failure ever occurs again. If the failure does not occur for a very long time, the programmer can be reasonably assured, but not guaranteed, that the cause for the failure has been fixed. Thus, testing a fix for a non-deterministic failure is much more difficult than testing a fix for a deterministic failure.

The fundamental conclusion that should be drawn from the simple debugging model is that debugging a program that experiences non-deterministic failures is substantially harder than debugging a program that experiences deterministic failures. This is true in the detection, isolation, and testing phases of debugging a program.

3. Checkpoint Debugging

The major advantage of checkpoint debugging is that it converts a large class of non-deterministic failures into deterministic failures. Since deterministic failures are easier to deal with than non-deterministic failures, the programmer's efficiency in dealing with program failures should be enhanced via checkpoint debugging. Indeed, some non-deterministic failures that would otherwise never be found due to their low frequency of occurrence will probably be caught the first time using checkpoint debugging. It would be nice to assert that programmer will find more bugs with checkpoint debugging than without checkpoint debugging. It is quite possible that an experienced programmer using post-mortem dumps and logging information will find more program bugs than an inexperienced programmer using checkpoint debugging. However, an experienced programmer using checkpoint debugging can always find at least as many (and hopefully more) bugs as can be found using post-mortem dumps and logging information. This is because the programmer always has the option with checkpoint debugging of executing the program until failure and then resorting to post-mortem dumps.

The question arises of how often do non-deterministic failures really occur? In general, almost all real-time programs have the potential of encountering non-deterministic failures. It should be pretty clear that checkpoint debugging will not work for all real-time programs. For example, it will be difficult to use checkpoint debugging if the real-time program does not have the ability to record the checkpoint information. However, there are plenty of examples of real-time programs that can be debugged using checkpoint debugging. For example, service nodes on a distributed system are an excellent example of real-time programs where it should be feasible to use checkpoint debugging. Indeed, the examination of distributed systems servers will provide a very useful focus for the exploration of checkpoint debugging.

So far only the advantages of checkpoint debugging have been discussed. It clearly time to discuss some of the disadvantages of checkpoint debugging. One of the disadvantages of

checkpoint debugging is that it requires continual overhead to take the program checkpoints. Most other debuggers do not incur any overhead until either 1) after the bug has been encountered or 2) the user has manually entered debugging mode. In addition, secondary storage is required to store the checkpoint information. One of the major areas that this thesis will have to investigate is the efficiency issues of checkpoint debugging. There are several obvious optimizations that can be applied to reduce checkpoint overhead.

- ~ The rate at which checkpoints occur can be adjusted to reduce the total amount of storage required to store the checkpoints. In order to conserve storage it turns out to be the case that a new checkpoint should be taken whenever the storage required by the incremental part of a checkpoint starts to exceed the fixed part of the checkpoint.

- ~ A previous checkpoint can be deleted once a new checkpoint has been successfully been taken. (This does not mean that the checkpoint *should* be deleted.)

- ~ The immutable information associated with a server does not need to be stored with fixed part of each checkpoint.

There are undoubtedly other optimizations that are applicable. It is expected that this thesis will present and evaluate all applicable optimizations for reducing checkpoint overhead.

Many interesting programs can access secondary storage directly. Conceptually, it is desirable to include the secondary storage as part of the program state for the purposes of checkpointing. Pragmatically, there are two significant reasons why this is not practical. First, it is necessary to dedicate an equivalent amount of secondary storage to contain the checkpoint information. Second, the amount of time required to actually checkpoint the secondary storage is measured in units of minutes. It is quite likely that the program will be unavailable for service for the duration of the secondary storage checkpoint. For many programs, a regular program unavailability that is measured in minutes will be unacceptable. This thesis will examine this problem and propose methods for alleviating this problem.

Many interesting programs essentially require a real-time response. It may be the case that these programs will not even be able to tolerate the delay required to checkpoint the primary memory associated with a program. (Garbage-collectors present a similar problem concerning real-time response.) This thesis will investigate the possibility of developing techniques for checkpointing programs that require real-time response. It may be the case that the only practical method for reducing delay will require an architectural solution.

Many programs are implemented as cooperating processes that communicate via shared memory. For example, an interrupt process will frequently modify some common memory to indicate the occurrence of an event. It should be pretty clear that this will potentially introduce non-determinism into the program. It is pretty clear that many programs will not be implementable without the ability to share memory. Thus, it will be necessary to devise techniques to permit communication via shared memory under checkpoint debugging.

It is claimed that checkpoint debugging converts most non-deterministic failures into deterministic failures. However, it is not claimed that checkpoint debugging converts *all* non-deterministic failures into deterministic failures. Any time there is non-determinism in program it is possible to encounter a non-deterministic failure. This suggests that non-determinism should be minimized if not completely eliminated in order to reduce the probability that non-deterministic failure will occur. If a non-deterministic failure occurs despite these precautions, the user will know that it is the non-deterministic portion of the code that is responsible for the failure. This thesis will examine techniques for removing non-determinism from programs.

The security aspects of checkpoint debugging are interesting. Take a printer server as an example. Suppose a user asks the printer server to print a confidential document and the server fails in the midst of printing the document. This means that the relevant checkpoint will possibly contain portions of this confidential document as part of its state. Thus, the maintainer will have access to this confidential information while debugging the server failure. Since solving the security problems introduced by checkpoint debugging does not appear to be central to the development of this thesis, no extensive effort will be undertaken to solve the security loopholes.

4. Distributed Systems

So far, checkpoint debugging has only been discussed in the context of a single processor environment. Checkpoint debugging is also applicable in the context of a distributed system environment. While additional problems occur in using checkpoint debugging in a distributed environment, these problems do not seem to insurmountable. Before continuing it is worthwhile to briefly discuss what a distributed system is and why it is useful.

Distributed systems are the next evolutionary step in computer systems after time-sharing systems. A time-sharing system presents the user with the illusion of a dedicated computer with

associated primary memory, secondary storage, and peripherals. Processor time-slicing is the mechanism by which this illusion is implemented. A distributed system actually provides the user with a dedicated personal computer. This personal computer contains a processor, memory, display, keyboard, and a local network interface. This personal computer may optionally provide a modest amount of fixed and/or demountable secondary storage. The personal computer does not contain the more expensive peripherals. High-speed printers, magnetic tape drives, and large disk drives are examples of such expensive peripherals. The local network permits the sharing of the more expensive peripherals. Distributed systems offer several advantages over time-sharing systems.

~ The response time of the personal computer tends not to vary significantly.

~ A faulty piece of hardware/software is less likely to affect all users.

~ A distributed system is potentially more reliable because critical components can be more readily replicated.

There is currently a great deal of interest in the area of distributed systems.

Generally speaking, a *distributed system server* or simply *server* is a program that permits multiple users on a local network to share either an expensive peripheral (i.e. a high-capacity disk drive) or a database (i.e. a phone directory). This particular definition of a server is not particularly exact. There are several attributes of servers that are interesting.

~ Most servers are real-time programs.

~ Most servers are expected to be highly available.

~ Most servers are run without human supervision.

Each of these attributes makes the debugging of servers difficult. As was mentioned earlier, real-time programs are susceptible to non-deterministic failures which are difficult to locate. The high availability requirement suggests that the server should be rebooted as soon as a fatal bug is encountered. The high availability constraint further suggests that the server should be run with as few bugs as possible. It is difficult to remove bugs from a program 1) when nobody is around when the failure occurs and 2) when the program is supposed to running continuously. Of course, it is a premise of this thesis that checkpoint debugging will make it significantly easier to debug servers.

Two additional problems become an issue in using checkpoint debugging in servers. The first problem is the nested server problem. The second problem is the failure detection and propagation problem. Each of these issues will be discussed below.

An interesting situation occurs when servers are nested. Suppose server A uses server B to accomplish part of its service. Further suppose that server B performs the service incorrectly in such a manner as to subsequently cause server A to fail. The most recent checkpoint pertaining to server A will be retained. However, the most recent checkpoint pertaining to server B will not be retained since server B never detected a failure. Thus, when the maintainer of server A determines that server B was the actual cause of the failure of server A there will be no associated checkpoint for server B to determine why server B behaved incorrectly.

A simple solution to this problem is to have server A request that server B save its relevant checkpoint information whenever server A fails. There are several drawbacks to this solution. First, server A might in actuality user servers B_1 through B_n. Thus, it will be necessary to save the checkpoint information for servers B_1 through B_n when server A detects a failure. Also, server B might invoke servers C_1 through C_m. Thus, when server A asks for the server B checkpoint information, it is necessary for server B to also ask for the checkpoint information for severs C_1 through C_m. Hence, the number of checkpoints that need to be retained can potentially fan-out to encompass the entire distributed system. Another problem with this solution is that server B may not be running with checkpoint debugging enabled. Thus, it will not even be possible to get the checkpoint information associated with B. As can be seen, this simple solution to the nested server problem suffers from several drawbacks. The thesis will examine this and solutions to the nested server problem in much greater detail.

The server failure detection problem will be discussed presently. Human supervision is the most common means of providing failure detection for programs. The human notices that the program is "behaving funny" and then terminates the program. As was mentioned earlier, servers tend not to have much in the way of human supervision. Thus, it is necessary to provide some other means of detecting failures. Computer supervision is the obvious next choice if human supervision is unavailable. Thus, it will probably be necessary for the user to provide some code for detecting server failures. This failure detection code is called *watch-dog code.* Hopefully, the watch-dog code will be sufficiently simple that it will not fail more often than the program fails. This thesis will investigate the issues involved in developing and maintaining watch-dog code.

The failure detection problem is compounded by the nested server problem. This is called the failure propagation problem. If server A invokes server B which subsequently fails, it is quite possible that server A will also fail. Alternatively, servers A_1 and A_2 may both invoke server B. Now assume that A_1 sends a message to server B that causes server B to fail. Now server A_2 sends a message to server B that would otherwise not cause server B to fail. However, since server B has just failed, server B is unable to respond properly to the message from server A_2. Thus, server A_2 may fail. This situation can be further complicated by server A_1 timing out and retransmitting its message to server B causing to be down continuously. This thesis will also examine this problem and propose various solutions.

5. Previous Work

There is nothing particularly new about the mechanism of checkpointing. The file systems of many operating systems are checkpointed in order to be able to recover from a damaged file system. Similarly, many database systems also use checkpointing for crash recovery purposes. Indeed, the most common usage for checkpointing is for some sort of crash recovery purpose. However, the usage of checkpointing for debugging purposes has had little previous investigation.

In some sense, checkpoint debugging is already commonly used to debug batch programs. A batch program is clearly deterministic since its output is dependent completely upon static input files. Thus, the user may reproduce a program failure simply by rerunning the program on the same input files. Debugging batch programs that take a long time to complete may be very inconvenient if the failure occurs near the end of program execution. This is because no intermediate snapshots are taken. Thus, while batch program debugging fulfills the definition of checkpoint debugging, it does not fulfill the spirit of checkpoint debugging.

The closest system to checkpoint debugging in the literature is Randell's recovery block scheme [Randell 75]. The primary purpose of recovery blocks is to produce software that can continue to operate in the presence of residual software failures. Whenever a failure is encountered, the program is backed up to the beginning of the most recent recovery block. At this point an alternate method of computing the result is tried. In this case, the checkpointing implicit in backing up is used strictly for error recovery purposes. Little information is saved about the cause for the original failure. Thus, the checkpoint information collected for recovery blocks is only kept for as long as it

takes to successfully reach the end of a recovery block. This is in sharp contrast to checkpoint debugging which keeps the checkpoint information around until the user is satisfied that the cause for a failure has been isolated.

Balzer's EXDAMS (EXtendable Debugging And Monitoring System) [Balzer 69] is notable in that it collects information about a program on a *history tape.* This history tape corresponds roughly to the incremental part of a checkpoint. After EXDAMS has collected the history tape, this history tape is examined by the user to discover what the program did. One major drawback of EXDAMS is that the user is responsible for selecting what information gets sent to the history tape. Thus, if the user does not send the appropriate information to the history tape, the user will not be able to determine the cause of a failure.

The AIDS system [Grishnman 70] is a somewhat typical debugging system with the exception that AIDS contains a RETREAT command. The RETREAT command permits the user to backup a program by roughly a thousand machine instructions. Of course, this is a limited version of checkpointing. The RETREAT command will not be very useful if the cause for the failure occurred more than a thousand instructions before it was detected. In addition, the RETREAT command is accomplished at the expense of having to simulate the entire program on an instruction-by-instruction basis. Each time an instruction stores to a memory location, the previous contents of the memory location is stored into a circular buffer. Continuous instruction simulation is clearly unacceptable for production grade software.

In the past decade, there have been a small number of theses on the general topic of debugging. Supposedly one of the most comprehensive discussions of debugging technology is to be found in Gaines' thesis [Gaines 69]. Since this thesis is not readily available, no further comments can be made at this time. Satterhwaite's thesis [Satterhwaite 75] is concerned with providing language level debugging facilities. In particular, Satterhwaite's system provided very powerful tracing facilities for the Algol W programming language. Model's thesis [Model 79] is concerned with monitoring complex programs. This thesis shows that good program monitoring tools are essential for locating program bugs. Myers' thesis [Myers 80] is an spin-off of Model's thesis and provides tools for graphically displaying program data structures. Schiffenbauer's thesis [Schiffenbauer 81] provides a tool for debugging application programs that are distributed across a network. These theses are mentioned because they deal with some aspect of debugging and not because they are particularly pertinent to checkpoint debugging.

There are currently a number of graduate students across the ARPAnet who are working on theses in the general area of debugging. The most relevant work to checkpoint debugging is that of Richard Pattis at Stanford who is working on a system that is vaguely similar to EXDAMS. In particular, Pattis is collecting detailed program history information for subsequent analysis by a debugger. The list of other people known to the author to be working on debugging related issues is Marzullo (Stanford), Kenniston (Stanford), Subramanian (CMU), Chiu (MIT), and Bauman (MIT).

6. Outline and Schedule

The current rough outline for this thesis contains four substantial chapters excluding the introductory chapter (chapter one) and the summary chapter (chapter six). Currently, chapter two will investigate any theoretical models that seem appropriate to the thesis. The model of how programs are debugged developed earlier in this proposal will clearly be one section of this chapter. Chapter three will concern itself with the implementation and performance issues of checkpoint debugging in a single processor environment. Chapter four will concern itself with the issues of using checkpoint debugging in a distributed system environment. The failure detection and propagation problem is currently considered to be sufficiently severe as to warrant an entire chapter dedicated to the problem. As the thesis develops, chapters may be added or removed as deemed appropriate.

The areas of checkpointing distributed systems and failure detection need additional research and insight before the corresponding thesis chapters can even be started. Also, a more extensive literature search into debugging practices will be necessary. While all programmers seem to go through the experience of debugging programs, very few programmers publish anything about debugging experiences. Debugging practices for actual real-time systems, such as airline reservation systems, bank teller systems, database management systems, phone exchanges, and network packet switches will be particularly relevant to this thesis. A simple incomplete experimental checkpoint debugging system has been implemented. A real-time program running under this simple system ran three times slower. However, the reasons for this factor three slow down turned out to be artifacts of the implementation. In order to make any definitive statements about performance it would be necessary to do a full scale implementation of an operating system that supports checkpoint debugging. However, time will not permit the expenditure of effort

required to actually implement such an operating system. Thus, it is unlikely that the performance predictions for checkpoint debugging made by this thesis will be backed up by experimental results. However, other areas of the thesis should be amenable to experimental verification without requiring vast expenditures of effort. As these areas are identified, the experiments will be formulated and performed. The schedule for the thesis is that a rough draft of the major chapters by the end of spring term 1983 and the final thesis should be handed in by the end of summer term 1983. These dates are optimistic and realistically some slippage is expected.

References

[Balzer 69]
Balzer, R. M.
EXDAMS -- EXtendable Debugging and Monitoring System.
In *Proceedings AFIPS Spring Joint Computer Conference*, pages 567-580. RAND Corporation, AFIPS, 1969.

[Gaines 69]
Gaines, R. Stockton.
The Debugging of Computer Programs.
PhD thesis, Princton University, Augest, 1969.

[Grishnman 70]
Grishman, Ralph.
The Debugging System AIDS.
In *Proceedings AFIPS Spring Joint Computer Conference*, pages 57-75. New York University, AFIPS, 1970.

[Model 79]
Mitchel L. Model.
Monitoring System Behavior In a Complex Computational Environment.
PhD thesis, Stanford, Augest, 1979.

[Myers 80]
Brad A. Myers.
Displaying Data Structures for Interactive Debugging.
Master's thesis, M.I.T., June, 1980.

[Randell 75]
Brian Randell.
System Structure for Software Fault Tolerance.
IEEE Transactions on Software Engineering SE-1(2):220-232, June, 1975.

[Satterhwaite 75]
Satterhwaite, Edwin H., Jr.
Souce Language Debugging Tools.
PhD thesis, Stanford University, May, 1975.

[Schiffenbauer 81]
Schiffenbauer, Robert David.
Interactive Debugging in a Distributed Computational Environment.
Master's thesis, M.I.T., September, 1981.