Diagnosis of Embedded Software using Program Spectra*

Peter Zoeteweij[1] Rui Abreu[1] Rob Golsteijn[2] Arjan J.C. van Gemund[1]

[1] Embedded Software Lab
Delft University of Technology
The Netherlands
{p.zoeteweij,r.f.abreu,a.j.c.vangemund}@tudelft.nl

[2] Innovation Center Eindhoven
NXP Semiconductors
The Netherlands
rob.golsteijn@nxp.com

Abstract

Automated diagnosis of errors detected during software testing can improve the efficiency of the debugging process, and can thus help to make software more reliable. In this paper we discuss the application of a specific automated debugging technique, namely software fault localization through the analysis of program spectra, in the area of embedded software in high-volume consumer electronics products. We discuss why the technique is particularly well suited for this application domain, and through experiments on an industrial test case we demonstrate that it can lead to highly accurate diagnoses of realistic errors.

Keywords: diagnosis, program spectra, automated debugging, embedded systems, consumer electronics.

1 Introduction

Software reliability can generally be improved through extensive testing and debugging, but this is often in conflict with market conditions: software cannot be tested exhaustively, and of the bugs that are found, only those with the highest impact on the user-perceived reliability can be solved before the release. In this typical scenario, testing reveals more bugs than can be solved, and debugging is a bottleneck for improving reliability. Automated debugging techniques can help to reduce this bottleneck.

The subject of this paper is a particular automated debugging technique, namely software fault localization through the analysis of *program spectra* [11]. These can be seen as projections of execution traces that indicate which parts of a program were active during various runs of that program. The diagnosis consist in analyzing the extent to which the activity of specific parts correlates with errors detected in the different runs.

Locating a fault is an important step in actually solving it, and program spectra have successfully been applied for this purpose in several tools focusing on various application domains, such as Pinpoint [4], which focuses on large, dynamic on-line transaction processing systems, AMPLE [5], which focuses on object-oriented software, and Tarantula [9], which focuses on C programs.

In this paper, we discuss the applicability of the technique to embedded software, and specifically to embedded software in high-volume consumer electronics products. Software has become an important factor in the development, marketing, and user-perception of these products, and the typical combination of limited computing resources, complex systems, and tight development deadlines make the technique a particularly attractive means for improving product reliability.

To support our argument, we report the outcome of two experiments, where we diagnosed two different errors occurring in the control software of a particular product line of television sets from a well-known international consumer electronics manufacturer. In both experiments, the technique is able to locate the (known) faults that cause these errors quite well, and in one case, this implies an accuracy of a single statement in approximately 450K lines of code.

The remainder of this paper is organized as follows. In Section 2 we explain the diagnosis technique in more detail, and in Section 3 we discuss its applicability to embedded software in consumer electronics products. In Section 4 we describe our experiments, and in Section 5 we discuss how our current implementation can be improved. In Section 6 we discuss related work. We conclude in Section 7.

2 Preliminaries

In this section we introduce program spectra, and describe how they are used for diagnosing software faults.

*This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

```
void RationalSort(int n, int *num, int *den)
{  /* block 1 */
   int i,j,temp;

   for ( i=n-1; i>=0; i-- ) {
      /* block 2 */
      for ( j=0; j<i; j++ ) {
         /* block 3 */
         if (RationalGT(num[j], den[j],
                        num[j+1], den[j+1])) {
            /* block 4 */
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp; } } }
}
```

Figure 1. A faulty C function for sorting rational numbers

First we introduce the necessary terminology.

2.1 Failures, Errors, and Faults

As defined in [3], we use the following terminology.

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is the part of the total state of the system that may cause a failure.
- A *fault* is the cause of an error in the system.

To illustrate these concepts, consider the C function in Figure 1. It is meant to sort, using the bubble sort algorithm, a sequence of `n` rational numbers whose numerators and denominators are passed via parameters `num` and `den`, respectively. There is a fault (bug) in the swapping code of block 4: only the numerators of the rational numbers are swapped. The denominators are left in their original order.

A failure occurs when applying `RationalSort` yields anything other than a sorted version of its input. An error occurs after the code inside the conditional statement is executed, while `den[j]` $\neq$ `den[j+1]`. Such errors can be temporary: if we apply `RationalSort` to the sequence $\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$, an error occurs after the first two numerators are swapped. However, this error is "canceled" by later swapping actions, and the sequence ends up being sorted correctly. Faults do not automatically lead to errors either: no error will occur if the input is already sorted, or if all denominators are equal.

The purpose of *diagnosis* is to locate the faults that are the root cause of detected errors. As such, error detection is a prerequisite for diagnosis. As a rudimentary form of error detection, failure detection can be used, but in software more powerful mechanisms are available, such as pointer checking, array bounds checking, deadlock detection, etc.

In a software context, faults are often called *bugs*, and diagnosis is part of *debugging*. Computer-aided techniques as the one we consider here are known as *automated debugging*.

2.2 Program Spectra

A program spectrum [11] is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consist of a number of counters or flags for the different parts of a program. As such, recording a program spectrum is a lightweight analysis compared to other run-time methods, such as, e.g., dynamic slicing [10].

As an example, a *block count spectrum* tells how often each block of code is executed during a run of a program. In this paper, a block of code is a C language statement, where we do not distinguish between the individual statements of a compound statement, but where we do distinguish between the cases of a switch statement[1]. Suppose that the function `RationalSort` of Figure 1 is used to sort the sequence $\langle \frac{2}{1}, \frac{3}{1}, \frac{4}{1}, \frac{1}{1} \rangle$, which it happens to do correctly. This would result in the following block count spectrum, where block 5 refers to the body of the `RationalGT` function, which has not been shown in Figure 1.

block	1	2	3	4	5
count	1	4	6	3	6

Block 1, the body of the function `RationalSort`, is executed once. Blocks 2 and 3, the bodies of the two loops, are executed four and six times, respectively. To sort our example array, three exchanges must be made, and block 4, the body of the conditional statement, is executed three times. Block 5, the `RationalGT` function body, is executed six times: once for every iteration of the inner loop.

If we are only interested in whether a block is executed or not, we can use binary flags instead of counters. In this case, the block count spectra revert to block *hit* spectra. Beside block count/hit spectra, many other forms of program spectra exist. See [7] for an overview. In this paper we will work with block hit spectra, and hit spectra for logical threads used in the software of our test case (see Section 4.1).

2.3 Fault Diagnosis

The hit spectra of M runs constitute a binary matrix, whose columns correspond to N different parts of the program (see Figure 2). In our case, these parts are blocks of

[1] This is a slightly different notion than a *basic block*, which is a block of code that has no branch.

$$
\begin{array}{c} \\ M \text{ spectra} \\ \\ \\ \end{array}
\begin{array}{c} \overset{N \text{ parts}}{\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix}} \\ \begin{array}{cccc} s_1 & s_2 & \dots & s_N \end{array} \end{array}
\quad
\overset{\text{errors}}{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}}
$$

Figure 2. The ingredients of fault diagnosis

C code. In some of the runs an error is detected. This information constitutes another column vector, the error vector. This vector corresponds to a hypothetical part of the program that is responsible for all observed errors. Fault localization essentially consists in identifying the part whose column vector resembles the error vector most.

In the field of data clustering, resemblances between vectors of binary, nominally scaled data, such as the columns in our matrix of program spectra, are quantified by means of *similarity coefficients* (see, e.g., [8]). As an example, the Jaccard similarity coefficient (see also [8]) expresses the similarity s_j of column j and the error vector as the number of positions in which these vectors share an entry 1 (i.e., block was exercised and the run has failed), divided by this same number plus the number of positions in which the vectors have different entries:

$$s_j = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)} \qquad (1)$$

where $a_{pq}(j) = |\{i \mid x_{ij} = p \wedge e_i = q\}|$, and $p, q \in \{0, 1\}$.

Under the assumption that a high similarity to the error vector indicates a high probability that the corresponding parts of the software cause the detected errors, the calculated similarity coefficients rank the parts of the program with respect to their likelihood of containing the faults.

To illustrate the approach, suppose that we apply the `RationalSort` function to the input sequences $I_1 = \langle\ \rangle$, $I_2 = \langle \frac{1}{4} \rangle$, $I_3 = \langle \frac{2}{1}, \frac{1}{1} \rangle$ and $I_4 = \langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$, $I_5 = \langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$, and $I_6 = \langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$.

I_1, I_2, and I_6 are already sorted, and lead to passed runs. I_3 is not sorted, but the denominators in this sequence happen to be equal, in which case no error occurs. I_4 is the example from Section 2.1: it is not sorted, and an error occurs during its execution, but this error goes undetected. Only for I_5 the program fails. The calculated result is $\langle \frac{1}{1}, \frac{2}{2}, \frac{4}{3}, \frac{3}{4} \rangle$ instead of $\langle \frac{1}{4}, \frac{2}{2}, \frac{4}{3}, \frac{3}{1} \rangle$, which is a clear indication that an error has occurred.

The block hit spectra for these runs are as follows ('1' denotes a hit), where block 5 corresponds to the body of the `RationalGT` function, which has not been shown in Figure 1.

	block					
input	1	2	3	4	5	error
I_1	1	0	0	0	0	0
I_2	1	1	0	0	0	0
I_3	1	1	1	1	1	0
I_4	1	1	1	1	1	0
I_5	1	1	1	1	1	1
I_6	1	1	1	0	1	0

For this data, the calculated Jaccard coefficients are $s_1 = \frac{1}{6}$, $s_2 = \frac{1}{5}$, $s_3 = \frac{1}{4}$, $s_4 = \frac{1}{3}$, $s_5 = \frac{1}{4}$, which (correctly) identifies block 4 as the most likely location of the fault.

3 Relevance to Embedded Software

The effectiveness of the diagnosis technique described in the previous section has already been demonstrated in several articles (see, e.g., [1], [4], [9]). In this paper we present the benefits and discuss the issues specifically related to debugging embedded software in consumer electronics products. Especially because of constraints imposed by the market, the conditions under which this software is developed are somewhat different from those for other software products:

- To reduce unit costs, and often to ensure portability of the devices, the software runs on non-commodity hardware, and computing resources are limited.

- As a consequence, many facilities that developers of non-embedded software have come to rely on are absent, or are available only in rudimentary forms. Examples are profiling tools that give insight in the dynamic behavior of systems.

- At the same time, the systems are highly concurrent, and operate at a low level of abstraction from the hardware. Therefore, their design and implementation are complicated by factors that can largely be abstracted away from in other software systems, such as deadlock prevention, and timing constraints involved in, e.g., writing to the graphics display only in those fractions of a second that the screen is not being refreshed.

- On top of challenges that the entire software industry has to deal with, such as geographically distributed development organizations, the strong competition between manufacturers of consumer electronics makes it absolutely vital that release deadlines are met.

- Although important safety mechanisms, such as short-circuit detection, are sometimes implemented in software, for a large part of the functionality there are no personal risks involved in transient failures.

Consequently, it is not uncommon that consumer electronics products are shipped with several known software faults outstanding. To a certain extent, this also holds for other software products, but the combination of the complexity of the systems, the tight constraints imposed by the market, and the relatively low impact of the majority of possible system failures creates a unique situation. Instead of aiming for correctness, the goal is to create a product that is of value to customers, despite its imperfections, and to bring the reliability to a commercially acceptable level (also compared to the competition) before a product must be released.

The technique of Section 2 can help to reach this goal faster, and may thus reduce the time-to-market, and lead to more reliable products. Specific benefits are the following.

- As a black-box diagnosis technique, it can be applied without any additional modeling effort. This effort would be hard to justify under the market conditions described above. Moreover, concurrent systems are difficult to model.
- The technique improves insight in the run-time behavior. For embedded software in consumer electronics, this is often lacking, because of the concurrency, but also because of the decentralized development.
- We expect that the technique can easily be integrated with existing testing procedures, such as overnight playback of recorded usage scenarios. In addition to the information that errors have occurred in some scenarios, this gives a first indication of the parts of the software that are likely to be involved in these errors. In the large, geographically distributed development organizations that we are dealing with, it may also help to identify which teams of developers to contact.
- Last but not least, the technique is light-weight, which is relevant because of the non-commodity hardware and limited computing resources. All that is needed is some memory for storing program spectra, or for calculating the similarity coefficients on the fly (which reduces the space complexity from $O(M \times N)$ to $O(N)$, see Section 5). Profiling tools such as `gcov` are convenient for obtaining program spectra, but they are typically not available in a development environment for embedded software. However, the same data can be obtained through source code instrumentation.

While none of these benefits are unique, their combination makes program spectrum analysis an attractive technique for diagnosing embedded software in consumer electronics.

4 Experiments

In this section we describe our experience with applying the techniques of Section 2 to an industrial test case.

4.1 Platform

The subject of our experiments is the control software in a particular product line of analog television sets. All audio and video processing is implemented in hardware, but the software is responsible for tasks such as decoding remote control input, displaying the on-screen menu, and coordinating the hardware (e.g., optimizing parameters for audio and video processing based on an analysis of the signals). Most teletext[2] functionality is also implemented in software.

The software itself consists of approximately 450K lines of C code, which is configured from a much larger (several MLOC) code base of Koala software components [12].

The control processor is a MIPS running a small multitasking operating system. Essentially, the run-time environment consists of several threads with increasing priorities, and for synchronization purposes, the work on these threads is organized in 315 logical threads inside the various components. Threads are preempted when work arrives for a higher-priority thread.

The total available RAM memory in consumer sets is two megabyte, but in the special developer version that we used for our experiments, another two megabyte was available. In addition, the developer sets have a serial connection, and a debugger interface for manual debugging on a PC.

4.2 Faults

We diagnosed two faults, one existing, and one that was seeded to reproduce an error from a different product line.

Load Problem. A known problem with the specific version of the control software that we had access to, is that after teletext viewing, the CPU load when watching television (TV mode) is approximately 10% higher than before teletext viewing. This is illustrated in Figure 3, which shows the CPU load for the following scenario: one minute TV mode, 30 s teletext viewing, and one minute of TV mode. The CPU load clearly increases around the 60th sample, when the teletext viewing starts, but never returns to its initial level after sample 90, when we switch back to TV mode.

Teletext Lock-up Problem. Another product line of television sets provides a function for searching in teletext pages. An existing fault in this functionality entails that searching in a page without visible content locks up the teletext system. A likely cause for the lock-up is an inconsistency in the values of two state variables in different components,

[2] A standard for broadcasting information (e.g., news, weather, TV guide) in text pages, very popular in Europe.

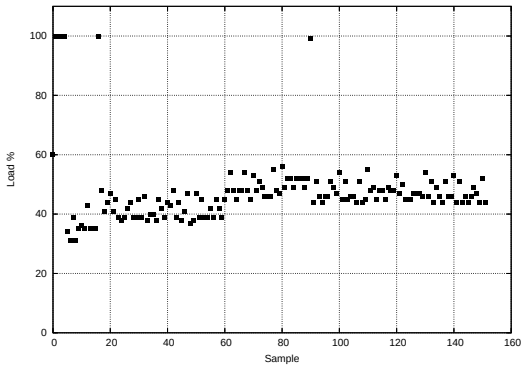

Figure 3. CPU load measured per second

for which only specific combinations are allowed. We hard-coded a remote control key-sequence that injects this error on our test platform.

4.3 Implementation

We wrote a small Koala component for recording and storing program spectra, and for transmitting them off the television set via the serial connection. The transmission is done on a low-priority thread while the CPU is otherwise idle, in order to minimize the impact on the timing behavior. Pending their transmission via the serial connection, our component caches program spectra in the extra memory available in our developer version of the hardware.

For diagnosing the load problem we obtained hit spectra for the logical threads mentioned in Section 4.1, resulting in spectra of 315 binary flags. We approached the lock-up problem at a much finer granularity, and obtained block hit spectra for practically all blocks of code in the control software, resulting in spectra of over 60,000 flags.

The hit spectra for the logical threads are obtained by manually instrumenting a centralized scheduling mechanism. For the block hit spectra we automatically instrumented the entire source code using the Front [2] parser generator.

In Section 2.3 we use program spectra for different runs of the software, but for embedded software in consumer electronics, and indeed for most interactive systems, the concept of a run is not very useful. Therefore we record the spectra per *transaction*, instead of per run, and we use two different notions of a transaction for the two different faults that we diagnosed:

- for the load problem, we use a periodic notion of a transaction, and record the spectra per second.
- for the lock-up problem, we define a transaction as the computation in between two key-presses on the remote control.

4.4 Diagnosis

For the load problem we used the scenario of Figure 3. We marked the last 60 spectra, for the second period of TV mode as 'failed,' and those of earlier transactions as 'passed.' In the ranking that follows from the analysis of Section 2.3, the logical thread that had been identified by the developers as the actual cause of the load problem was in the second position out of 315. In the first position was a logical thread related to teletext, whose activation is part of the problem, so in this case we can conclude that although the diagnosis is not perfect, the implied suggestion for investigating the problem is quite useful.

For the lock-up problem, we used a proper error detection mechanism. On each key-press, when caching the current spectrum, a separate routine verifies the values of the two state variables, and marks the current spectrum as failed if they assume an invalid combination. Although this is a special-purpose mechanism, including and regularly checking high-level assert-like statements about correct behavior is a valid means to increase the error-awareness of systems.

Using a very simple scenario of 23 key-presses that essentially (1) verifies that the TV and teletext subsystems function correctly, (2) triggers the error injection, and (3) checks that the teletext subsystem is no longer responding, we immediately got a good diagnosis of the detected error: the first two positions in the total ranking of over 60,000 blocks pointed directly to our error injection code. Adding another three key-presses to exonerate an uncovered branch in this code made the diagnosis perfect: the exact statement that introduced the state inconsistency was located out of approximately 450K lines of source code.

5 Discussion

Especially the results for the lock-up problem have convinced us that program spectra, and their application to fault diagnosis are a viable technique and useful tool in the area of embedded software in consumer electronics. However, there are a number of issues with our implementation.

First, we cannot claim that we have not altered the timing behavior of the system. Because of its rigorous design, the TV is still functioning properly, but everything runs much slower with the block-level instrumentation (e.g., changing channels now takes seconds). One reason is that currently, we collect block *count* spectra at byte resolution, and convert to block *hit* spectra off-line. Updating the counters in a multi-threaded environment requires a critical section for every executed block, which is hugely expensive. Fortunately, this information is not used, and we believe we can implement a binary flag update without a critical section.

Second, we cache the spectra of passed transactions, and transmit them off the system during CPU idle time. Be-

cause of the low throughput of the serial connection, this may become a bottleneck for large spectra and larger scenarios. In our case we could store 25 spectra of 65,536 counters, which was already slowing down the scenarios with more than that number of transactions, but even with a more memory-efficient implementation, this inevitably becomes a problem with, for example, overnight testing.

For many purposes, however, we will not have to store the actual spectra. In particular for fault diagnosis, ultimately we are only interested in the calculated similarity coefficients, and all similarity coefficients that we are aware of are expressed in terms of the four counters a_{00}, a_{01}, a_{10}, and a_{11} introduced in Section 2.3. If an error detection mechanism is available, like in our experiments with the lock-up problem, then these four counters can be calculated on the fly, and the memory requirements become linear in the number columns in the matrix of Figure 2.

6 Related Work

Program spectra were introduced in [11], where hit spectra of intra-procedural paths are analyzed to diagnose year 2000 problems. The distinction between count spectra and hit spectra is introduced in [7], where several kinds of program spectra are evaluated in the context of regression testing. In the introduction we already mentioned three practical diagnosis/debugging tools [4, 5, 9] that are essentially based on the same diagnosis method as ours. A recent study, reported in [1], indicates that the choice of the similarity coefficient, as introduced in Section 2.3 can be of significant influence on the quality of the diagnosis. In the experiments reported in the present paper we used both the Jaccard coefficient of Eq. (1), and the best coefficient identified in [1], but the results were essentially the same.

As we mentioned in Section 3, black box techniques like spectrum-based diagnosis can be applied without additional knowledge about a system. An example of a white box technique is model-based diagnosis (see, e.g., [6]), where a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. White box approaches to software diagnosis exist (see, e.g., [13]), but software modeling is extremely complex, so most software diagnosis techniques are black box.

7 Conclusion

In this paper we have demonstrated software fault diagnosis through the analysis of program spectra, on a large-scale industrial test case in the area of embedded software in consumer electronics devices. In addition to confirming established effectiveness results, our experiments indicate that the technique lends itself well for application in the resource-constrained environments that are typical for the development of embedded software.

While our current experiments focus on development-time debugging, they open corridors to further applications, such as run-time recovery by rebooting only those parts of a system whose activities correlate with detected errors.

8 Acknowledgments

We would like to thank Pierre van de Laar for valuable comments on an earlier version of this paper.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of PRDC'06*, pages 39–46. IEEE Computer Society, 2006.

[2] L. Augusteijn. Front: a front-end generator for Lex, Yacc and C, release 1.0, 2002. See `http://front.sourceforge.net/`.

[3] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of DSN 2002*, pages 595–604. IEEE Computer Society, 2002.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In A. P. Black, editor, *Proceedings of ECOOP 2005*, volume 3586 of *LNCS*, pages 528–550. Springer-Verlag, 2005.

[6] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[7] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. *ACM SIGPLAN Notices*, 33(7):83–90, 1998.

[8] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of ICSE 2002*, pages 467–477. ACM Press, 2002.

[10] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[11] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of ESEC/FSE 97*, volume 1301 of *LNCS*, pages 432–449. Springer–Verlag, 1997.

[12] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, March 2000.

[13] F. Wotawa, M. Strumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In T. Hendtlass and M. Ali, editors, *Proceedings of IAE/AIE 2002*, volume 2358 of *LNCS*, pages 746–757. Springer-Verlag, 2002.