Automatic Fault Diagnosis in Embedded Software*

Rui Abreu[†] **Peter Zoeteweij**[†] **Rob Golsteijn**[‡] **Arjan J.C. van Gemund**[†]

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

[‡]NXP
The Netherlands
rob.golsteijn@philips.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. An important aspect of this technique is the similarity coefficient used to rank potential fault locations. We evaluate the effectiveness of spectrum-based fault localization in a set of benchmark programs. In this context, our experiments indicate that a particular coefficient consistently outperforms the coefficients currently used by other tools. Furthermore, we also applied this technique to an industrial TV software product. We discuss why it 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: *Software reliability, automated debugging, software fault diagnosis, fault localization, program spectra.*

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 the bottleneck for improving reliability. Automated debugging techniques can help to reduce this bottleneck. These techniques give a diagnosis for errors that are detected during the execution of a program, which can help programmers to locate their root causes, and thus to reduce the effort spent on manual debugging.

Diagnosis techniques, which include automated debugging, can be classified as white box or black box, depending on the amount of knowledge that is required about the system under study. An example of a white box technique is model-based diagnosis (see, e.g., [7, 8]), where a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. While white box approaches to software diagnosis exist (see, e.g., [14, 17, 19, 20]), software modeling is extremely complex. Hence, most software diagnosis techniques are black box (see, e.g., [4, 6, 12, 13]). The subject of this paper is a specific automated debugging technique, namely software fault localization through the analysis of program spectra. Because this technique requires practically no information about the system being diagnosed, it can be classified as a black box diagnosis technique.

Program spectra can be seen as projections of traces of software activity. Typically, this projection is much more compact than a trace, which makes it an attractive technique in resource-constrained environments, such as embedded systems. The technique studied in this paper is based on analyzing the differences between program spectra obtained for correct behavior of the software, and program spectra obtained for faulty behavior of the software. An essential part of this analysis is the computation of a measure of similarity between different vectors in the program spectra data (each vector represents whether a given part in the program was executed in different transactions) and a vector that contains information about the detected errors. Different application reports of the technique are now emerging in the literature, but all use different similarity measures.

In this paper we first investigate the influence of the similarity measure on the quality of the diagnosis. To this end we apply the fault localization technique on a benchmark set of software faults known as the *Siemens Suite*. We obtain multiple diagnoses for every fault in the suite, each of them for a different similarity measure. These similarity

*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.

measures are taken from existing diagnosis tools in the areas of recovery and automated debugging, and from the molecular biology domain. A diagnosis for a particular measure of similarity consists of a list of possible locations for the fault ranked in order of similarity. Our evaluation is based on the position of the (known) location of the fault in this ranking. In particular, the contributions of this experiment are the following.

- We recognize that several existing tools are implementations of the same technique: fault diagnosis through the comparison of program spectra, which allows us to compare the techniques.

- We identify a measure of similarity between the program spectra that yields an average performance improvement of 5% under the specific conditions of our experiments, in terms of the amount of code that must be inspected. Improvements up to 30% are measured.

Besides the evaluation of the effectiveness of fault localization through program spectra, we also study the applicability of the technique to embedded software, and specifically to embedded software in high-volume consumer electronics products. To support the relevance of the tool in this context, 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 Philips. 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 introduce some basic concepts and terminology, and explain the diagnosis technique in more detail. In Section 3 we evaluate the diagnostic quality of a different number of similarity coefficients in pinpointing the faulty location using program spectra, and in Section 4 we discuss its applicability to embedded software in consumer electronics products. Related work is presented in Section 5. We present our conclusion in Section 6.

2 Preliminaries

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

```
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

- 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 latent: 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. Note that 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 [16] 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. For 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].

To illustrate the concept of a program spectrum, 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 [9] 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 4.2).

2.3 Fault Diagnosis

For fault diagnosis we need program spectra both for runs of the software in which an error has been detected (called *failed* runs), and for runs in which no error has been detected (called *passed* runs). Analyzing these spectra may identify those parts of a program that are active primarily in failed runs. These parts are also likely to contain the faults that cause the detected errors. We will demonstrate this approach using the `RationalSort` example.

Suppose we apply `RationalSort` to the two sequences $S_1 = \langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$ and $S_2 = \langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$. S_1 is already sorted, and leads to a passed run, but for S_2 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 two runs are as follows ('X' denotes a hit).

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

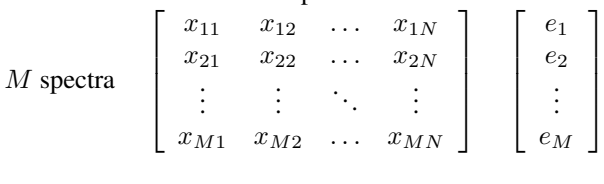

Figure 2. The ingredients of fault diagnosis

	block					
input	1	2	3	4	5	error
S_1	X	X	X		X	
S_2	X	X	X	X	X	X

The difference between the two block hit spectra (correctly) identifies block 4 as the most likely location of the fault: while all other blocks are executed in both runs, block 4 only occurs in the run where the error is detected.

Of course, this example is contrived in many ways: the number of blocks is small, no undetected errors have occurred, no routine in the program has multiple call sites, etc. However, it serves to illustrate the basic principle: the spectra of M runs constitute a binary matrix, whose columns correspond to N different parts of the program (see Figure 2). In some of the runs an error is detected. This information constitutes another column vector, the error vector. This vector can be though of as to represent a hypothetical part of the program that is responsible for all observed errors. Fault diagnosis 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., [11]). As an example, the Jaccard similarity coefficient, which we used in our experiments, 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, divided by this same number plus the number of positions in which the vectors have different entries:

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

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.

3 Benchmark Experiments

In this section we evaluate the effectiveness of the technique previously presented using a benchmark set of programs.

3.1 Benchmark Set

Evaluating different similarity coefficient techniques requires us to thoroughly test them. For this purpose we used a set of test programs known as the *Siemens suite* [10]. The Siemens suite is composed of seven programs. Every single program has a correct version and a set of faulty versions of the same program. Each faulty version contains exactly one fault. However, the fault may span through multiple statements and/or functions. Each program also has a set of inputs. Those inputs were created with the intention to test the full coverage of the programs. Table 1 provides more information about the programs in the package (for more detailed information refer to [10]). Although the Siemens suite was not assembled with the purpose of testing fault diagnosis techniques, it is typically used by the research community as the set of programs to test their techniques.

In our experiments we were not able to use all the programs provided by the Siemens suite. Because we conduct our experiments using block hit spectra, we can not use programs which contain data-dependent faults, i.e., faults that do not influence control flow. Versions 4 and 6 of *print_tokens*, version 38 of *tcas*, and version 10 of *tot_info* contain errors that are considered to be data dependent and were therefore discarded. Version 9 of *schedule2* and version 32 of *replace* were not considered in our experiments because no test case fails and therefore the existence of a fault was never revealed. Furthermore, as we are comparing ranking techniques, we decided to limit our experiment to single site faults. Hence, versions 12, and 21 of *replace*, versions 10, 11, 15, and 40 of *tcas*, version 7 of *schedule*, and version 1 of *print_tokens* were also discarded because the fault is extended to more than one site. In total, we discarded 14 versions out of 132 versions provided by the suite, using 118 versions in our experiments.

3.2 Data Acquisition

Collecting Spectra For obtaining block hit spectra we instrumented the source code of every single program in the Siemens suite. A function call was automatically inserted in the beginning of every block of code to log its execution. To automatically instrument the programs, we use the *Front* parser generator [2] (See [1] for details of the instrumentation process). Moreover, the programs were compiled on a Linux based environment with gcc-3.2.

Error Detection As for each program the Siemens suite includes a correct version, we use the output of the correct version of each program as error detection reference. We characterize a run as 'failed' if its output differs from the corresponding output of the correct version, and as 'passed' otherwise. As we explained in Section 2.1, failure detection is a rudimentary form of error detection where a faulty program may well go undetected. Suppose block j is the faulty block and recall a_{pq} as defined in Section 2.3, we define error detection accuracy for a given faulty block as

$$q_e = \frac{a_{11}(j)}{a_{11}(j) + a_{10}(j)}$$

With our strategy to detect failing and passing runs, we cannot expect good error detection accuracy. On average per program, in the Siemens set the error accuracy ranges from 1.2% (*schedule2*) to 21.1% (*tot_info*). This implies that measured diagnostic quality will be limited due to this low error detection accuracy.

3.3 Evaluation Metric

For the purpose of this discussion, we define quality of the diagnosis as the position that the faulty block has in the ranking. The notion behind this measure is how many blocks we still need to inspect until we identify the faulty block. If there are two blocks that rank with the same coefficient, we use the worst ranking position for both of them.

More precisely, let $d \in \{1, \ldots, N\}$ be the index of the block that we know to contain the fault. For all $j \in \{1, \ldots, N\}$, let s_j denote the similarity coefficient calculated for block j. Then the diagnostic quality is given by

$$q_d = |\{j | s_j \geq s_d\}| \tag{2}$$

3.4 Similarity Coefficients

The benchmark set of programs is a controlled environment where it is possible to test the efficiency of the technique described before. An important parameter of this technique is the similarity coefficient used to diagnose the faulty location and therefore we investigate the influence of the similarity coefficient on the quality of the diagnosis by applying a number of similarity coefficients known from the literature on the same data, and comparing the resulting diagnoses. Apart from the Jaccard coefficient presented in Section 2.3, we also evaluate the following similarity coefficients:

- Tarantula:

$$s_j = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}} \tag{3}$$

  This coefficient is used in the Tarantula system [13, 12].

- AMPLE:

$$s_j = | \frac{a_{11}(j)}{a_{01}(j) + a_{11}(j)} - \frac{a_{10}(j)}{a_{00}(j) + a_{10}(j)} | \tag{4}$$

Program	Faulty Versions	Blocks	Test Cases	Description
print_tokens	7	110	4056	lexical analyzer
print_tokens2	10	105	4071	lexical analyzer
replace	32	124	5542	pattern recognition
schedule	9	53	2650	priority scheduler
schedule2	10	60	2680	priority scheduler
tcas	41	20	1578	altitude separation
tot_info	23	44	1054	information measure

Table 1. Description of the Siemens Suite

This coefficient is used by the AMPLE tool. In [6] it is assumed that there is exactly one failing run, in which case the denominator $a_{01}(j) + a_{11}(j)$ equals 1.

- Ochiai:

$$s_j = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}} \quad (5)$$

This coefficient is used in [5] for computing genetic similarity in molecular biology.

In addition to the coefficient of Eq. (3), the Tarantula system uses a second coefficient, which amounts to the maximum of the two fractions in the denominator of Eq. (3). This second coefficient is interpreted as a *brightness* value by its visualization system, but the experiments in [12] indicate that the above coefficient can be studied in isolation. For this reason, we have not taken the brightness coefficient into account.

The AMPLE coefficient is used here outside its context. It amounts to the relative *difference* between the number of occurrences of a block in passed and failed runs, and hence also takes absence of a block in failing runs into account. This probably has little use without accumulating the calculated values to a coarser-grained unit of diagnosis (cf., accumulating call sequence weights to class weights, see Section 5), and therefore one should not project our results for this coefficient to the AMPLE tool.

Several other similarity coefficients are used in data clustering (see [5] for a study in the context of molecular biology). Although all the coefficients presented in [5] were used in our experiments, in this paper we have only included the Ochiai coefficient, which gave the best results.

3.5 Experimental Results

This section presents results obtained by applying the coefficients of similarity described in Section 3.4 to the data collected by the execution of the benchmark programs. The intention of this experiment is to find which of the coefficients of similarity leads to the better diagnosis according to the metric (2).

In Table 2 we show the average ranking position of the actual fault for the different similarity functions introduced above, per program. For each program we run all the versions (faults) on all test cases, and rank the blocks according to Equations (1), (3), and (5). Per equation we determine the position q_d of the faulty block in the ranking, and average this number over all versions of the program. We do not average over the seven programs because they have largely varying numbers of blocks.

	Jaccard	Tarantula	AMPLE	Ochiai
print_tokens	7.3	10.5	2.0	1.0
print_tokens2	17.5	20.0	16.4	13.9
replace	11.6	12.2	12.7	7.6
schedule	2.9	3.0	11.3	1.6
schedule2	31.1	31.3	34.7	25.1
tcas	8.8	8.8	9.8	7.9
tot_info	9.6	11.0	13.2	7.1

Table 2. Average fault ranking position

One significant observation from this table is that the Ochiai coefficient (Eq. (5)) consistently outperforms the other techniques in terms of diagnostic quality. The Jaccard coefficient and the coefficient used by Tarantula sometimes happen to have the same quality of diagnosis. However, in all except one of the situations where they differ we observe that Jaccard is better than Tarantula's coefficient. AMPLE's coefficient produces the worst quality of diagnosis except for two programs where the quality was better than Jaccard and the coefficient used by Tarantula. In one of these programs, the AMPLE coefficient yields similar quality compared to the Ochiai coefficient. As noted in Section 5, we are using the AMPLE coefficient outside its original context, and hence, this does not imply that the AMPLE system performs worse than the Tarantula system.

Figure 3 displays the same information as Table 2, expressed as the *percentage* of blocks a programmer has to inspect until he/she finds the bug, assuming that the programmer would inspect the code according to the ranking created by the diagnosis technique. This percentage is defined by the average rank of the fault divided by the total number of blocks. From this figure it is immediately clear that, under the specific conditions of our experiments, the Ochiai coefficient is superior. Using this coefficient of similarity, in the worst case of this experiment, the software de-

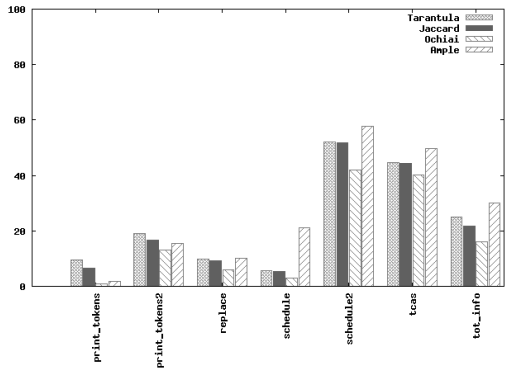

Figure 3. Percentage of blocks to be inspected

veloper is still 'obliged' to inspect 40% of the code to find the fault. In the best case, only 1% needs to be inspected. The Ochiai coefficient presents improvements ranging from 2.6% to 10% on average per program over the Jaccard coefficient (second-best technique). Per faulty version, improvements up to 30% were measured. Overall, this coefficient decreases the percentage of blocks of code to be inspected by 5%.

Figure 3 also shows that all of the coefficients work poorly for some programs, for instance *schedule2* and *tcas*. The possible causes for poor rankings are, mainly, when the faulty block is always exercised in passed and failed runs (for instance, the main function), deleted code, and dependent blocks (i.e., blocks that are always executed when the faulty block is also executed).

4 Applicability to Embedded Software

In this section we evaluate the feasibility of spectrum-based fault diagnosis to embedded (TV) software.

4.1 Relevance to Embedded Software

The effectiveness of the diagnosis technique previously introduced in Section 2 has been demonstrated in Section 3 and also in several articles (see, e.g., [4], [13]). In this section 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 condition 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 4.6). 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.

4.2 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 teletextfunctionality 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 [18].

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 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.

4.3 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 4, 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.

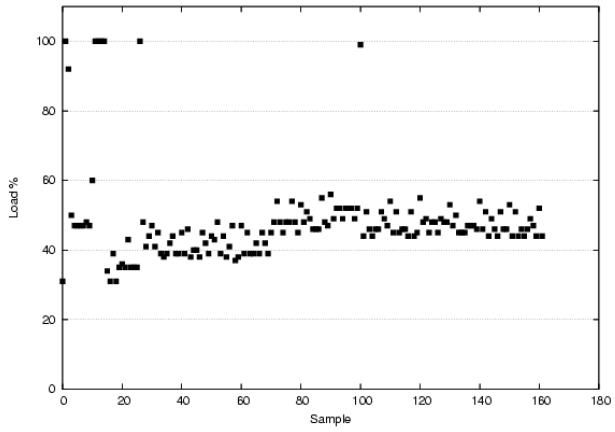

Figure 4. CPU load measured per second

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

4.4 Technical Details

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 4.2, 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 as described in Section 3.2.

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.

4.5 Diagnosis

For the load problem we used the scenario of Figure 4. 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.

4.6 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 several issues with our current 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 needed, 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. Because 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.

5 Related Work

The diagnosis approach described in Sections 2.2 and 2.3 has appeared in various guises in literature. Three systems are of particular interest, because the similarity coefficient that is used in the diagnosis is clearly described. They are Pinpoint, Tarantula, and AMPLE.

Pinpoint [4] is a framework for root cause analysis on the J2EE platform. It is developed in the context of the Recovery Oriented Computing project [15], and is targeted at large, dynamic Internet services, such as web-mail services and search engines. It combines the technique of the previous section with a specific form of error detection, based on information coming from the J2EE framework, such as caught exceptions, and errors visible to users, such as HTTP errors. This makes the approach self-contained in the sense that no external characterization of traces is needed.

The Tarantula system [12, 13] has been developed for

the C language, and applies the technique of the previous section to statement hit spectra. Compared to block hit spectra, the higher resolution of statement hit spectra may give a more detailed diagnosis in presence of statements that alter the flow of control inside a block, namely `break`, `continue`, `return`, and `goto`. Tarantula comes with a graphical user interface, that interprets the calculated value for the similarity coefficient as a color index, used to visualize the suspiciousness of program statements. Tarantula relies on external error detection for the classification of runs as passed or failed: whereas Pinpoint uses information from the J2EE framework for this classification, this information is input data for Tarantula. In other words, Tarantula implements only the diagnosis, and has to be complemented by adding a method of error detection.

AMPLE (Analyzing Method Patterns to Locate Errors) [6] is a system for identifying faulty classes in object-oriented software. It collects hit spectra of *method call sequences*, which are subsequences of a given length that occur in a full trace of incoming or outgoing method calls, received or issued by individual objects of a class. Each call sequence is assigned a weight, which captures the extent to which its occurrence or absence correlates with the detection of an error, i.e., it is a combined measure of similarity and inverted similarity. These weights are averaged over all call sequences of a class, leading to a class weight. Classes with a high weight are most likely to contain the fault that causes the detected error. Although the calculation of the sequence weights in AMPLE can be explained as an application of the technique of Section 2.3, the diagnosis is at class level, and the calculated coefficients are used only to collect evidence about classes, not to identify suspicious method call sequences.

Concluding, we can observe that three existing tools for diagnosis and automated debugging rely on an analysis of program spectra. Program spectra themselves were introduced in [16], 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 [9], where several kinds of program spectra are evaluated in the context of regression testing. As we already mentioned in the introduction, in the context of computer programs, fault localization based on the analysis of program spectra is an automated debugging technique. An example of a different (black box) technique in that category is Delta Debugging [21], which compares the program states of a failing and a passing run, and actively searches for failure-inducing circumstances in the differences between these states.

6 Conclusion

From our experiments with fault localization by means of analyzing program spectra we can draw the following conclusions

- Even at a low accuracy of error detection, the technique provides useful information about the location of a fault, and can therefore improve the efficiency of the debugging process.
- The Ochiai coefficient consistently outperforms the other coefficients that we studied.
- The technique seems to lend itself well for application in the resource-constrained environments that are typical for the development of embedded software.

In our future work we plan to investigate the influence of several factors that may affect the quality of the diagnosis, such as the fraction of the runs in which an error occurs, the accuracy with which these errors are detected, and the units of measurement for which the program spectra are obtained. We also plan to look at multiple faults, and at specific kinds of software faults that play an important role in practice, such as memory faults.

While improving the efficiency of the debugging process will improve the reliability of embedded software, we believe that the techniques studied here have a wider applicability, such as to enable the recovery of autonomous systems based on detailed information on what component or thread has become at fault [15]. Especially in a process-based environment, diagnosis may serve as the basis for an automated recovery strategy: it may well be possible to improve the reliability of a system by rebooting those processes whose activities correlate with "suspect," or potentially erroneous transactions. In this case, error detection could be based on generic indications that something is wrong, such as the handling of null pointers, and the violation of timing constraints.

7 Acknowledgments

We gratefully acknowledge the feedback from discussions with our TRADER project partners from Philips, Research, Philips Semiconductors, Philips TASS, Philips Consumer Electronics, Embedded Systems Institute, Design Technology Institute, IMEC, Leiden University, and Twente University.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Program spectra analysis in embedded systems: A case study. In *12th Ann. Conf. of the Advanced School for Computing and Imaging (ASCI'06)*. ASCI, June 2006. On `http://arxiv.org/corr`.

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

[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 the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[5] A. da Silva Meyer, A. A. Franco Farcia, and A. Pereira de Souza. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L). *Genetics and Molecular Biology*, 27(1):83–91, 2004.

[6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In A. P. Black, editor, *ECOOP 2005 : 19th European Conference, Glasgow, UK, July 25–29, 2005. Proceedings*, volume 3568 of *LNCS*, pages 528–550. Springer-Verlag, 2005.

[7] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

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

[9] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '98, Montreal, Canada, June 16, 1998*, pages 83–90, 1998.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200, Sorrento, Italy, 1994. IEEE Computer Society Press.

[11] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, NY, USA, 2005. ACM Press.

[13] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, May 2002*, pages 467–477. ACM Press, 2002.

[14] W. Mayer and M. Stumptner. Approximate modeling for debugging of program loops. In *Proceedings of the Fifteenth International Workshop on Principles of Diagnosis*, Carcassonne, June 2004.

[15] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, U.C. Berkeley, March 2002.

[16] 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 the Sixth European Software Engineering Conference (ESEC/FSE 97)*, volume 1301 of *LNCS*, pages 432–449. Springer–Verlag, 1997.

[17] M. Stumptner. Using design information to identify structural software faults. In *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence*, volume 2256 of *LNCS*, pages 473–486, London, UK, 2001. Springer-Verlag.

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

[19] F. Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002.

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

[21] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE-10), Charleston, South Carolina, November 2002*. ACM Press, 2002.