Chapter 4
Automated Reproduction and Analysis of Bugs in Embedded Software

Hanno Eichelberger, Thomas Kropf, Jürgen Ruf and Wolfgang Rosenstiel

4.1 Introduction

The importance of embedded software increases every year. For example, modern cars currently contain about 100 million lines of source code in embedded software [8]. The highest safety level of ISO26262 standards demands 10^9 h of operation without failure. However, projects in history show that, even with comprehensive testing, bugs remain undetected for years. The control computer of the space shuttle with just 500000 lines of source code was tested overall 8 years with an effort of $1000 per source code line, i.e., with a total effort of $500 million [18]. However, it was expected that one bug per 2000 lines of code remained in the last release in 1990. Such bugs may occur in very rare cases and may be only detected while testing the embedded system in a real operation environment.

Static analysis is effectively used during early testing, e.g., unit testing [4]. However, static analysis of bugs is getting close to its limits for complex software. The state space or control flow of big software is difficult to explore completely. Thus, the analysis has drawbacks in performance or in precision. Furthermore, semantic bugs are very application-specific and a wrong behavior cannot be detected even with optimized static analysis tools. Things are getting more severe if a complete and correct specification is not available as golden reference.

System testing describes the process of deploying and testing the software on the target platform. Compared to unit or component testing, which only tests single modules, the software is executed and tested with all integrated software and hardware modules. About 60% of the bugs are not detected before system testing [30].

H. Eichelberger (✉) · T. Kropf · J. Ruf · W. Rosenstiel
University of Tübingen, Tübingen, Germany
e-mail: hanno.eichelberger@uni-tuebingen.de

T. Kropf
e-mail: thomas.kropf@uni-tuebingen.de

J. Ruf
e-mail: juergen.ruf@uni-tuebingen.de

W. Rosenstiel
e-mail: wolfgang.rosenstiel@uni-tuebingen.de

D. Lettnin and M. Winterholer (eds.), *Embedded Software Verification and Debugging*, Embedded Systems, DOI 10.1007/978-1-4614-2266-2_4

However, depending on the software development process model, system testing may be applied only very late in the development process. Real sensors and devices are connected to the embedded system to achieve realistic inputs during system tests. This way, incompatibilities between sensor hardware and the software under test may be detected.

Studies show that the later a bug is detected, the more effort is required to fix it. Some studies present an exponential growth of bug fixing costs during development time [30]. Thus, much more effort is required when bugs are detected during system testing compared to unit testing, caused by close hardware interaction and certification requirements. However, a small percentage of bugs (20%) requires about 60–80% of the fixing effort [11]. When a big portion of complex bugs is detected very late in the development process, project schedules and project deadlines are negatively affected. Thus, the product can often not be placed on the market in time.

One cause for the high effort for bug fixing is the difficulty to reproduce bugs. Users in field or developers during operation tests are often not able to provide enough information to reproduce the bug in the laboratory. Different nondeterministic aspects (e.g., thread scheduling) may make it difficult to reproduce the same execution in the laboratory as the execution observed during operation. About 17% of the bugs of open source desktop and server applications [26] are even not reproducible based on bug reports of the community. The amount may be higher for embedded software with sensor-driven input.

When the bug can be reproduced with a test case, additional effort is required for analyzing the bug. About 8 h are required for running test cases and repairing a bug during system tests [24], if the test case is available. Open source projects like Mozilla receive 300 bug reports a day [5]. With such high bug fixing efforts, it is difficult to handle such high bug rates. Dynamic verification can support the developers in fixing bugs. However, most dynamic analysis tools require the monitoring of the software during runtime. Such monitoring tools are often only applicable on specific platforms.

Bugs are categorized into memory bugs, concurrency bugs, and semantic bugs. Empirical studies show that semantic bugs are the dominant root-cause [37]. The most common semantic bugs are implementations which do not meet the design requirements or which do not behave as expected. Tools to automatically locate root-causes of semantic bugs are required. Memory bugs are not challenging, because many memory profiling tools are available. Concurrency bugs are more problematic, especially their reproduction. More than one out of ten concurrency bugs cannot be reproduced [37].

Our own portable debugger-based approach for bug reproduction and dynamic verification achieves the following contributions to the current state of the art in the area of embedded software development:

- It avoids effortful manual bug reconstruction on any embedded platform by automatically recording and reproducing bugs using debugger tools.
- It improves multi-threaded bug detection by forcing randomized thread switching.

- It decreases the manual debugging effort by applying automated root-cause analysis on reproduced bugs.
- It avoids expensive monitoring hardware by implementing performance optimizations with hierarchical analysis using low-cost debugging tools.
- It reduces porting costs of dynamic verification tools by implementing them in extendable and easily adaptable modules.

Our approach supports developers to detect and reconstruct (mainly semantic and concurrency) bugs faster. It further helps them to fix bugs faster by implementing automated root-cause analyses. Our tool saves debugging costs and hardware investment costs. It is supported by most embedded platforms. It assists developer teams to place their software products earlier on the market.

Section 4.2 gives a brief overview of the normal manual debugging process. Section 4.3 presents methods for automated reproduction of bugs, followed by our own debugger-based approach. Section 4.4 shows how assertion-based verification can be implemented using debugger tools. Section 4.5 presents concepts for analyzing the root-causes of bugs without assertions and concepts for the acceleration of monitoring implementations with cheap debugger interfaces.

4.2 Overview

The workflow in Fig. 4.1 presents the process of manually locating and fixing a bug. It starts by testing the embedded system and the included software during operation in a system testing environment. For example, a navigation software can be executed in a test drive connected to real sensors. During the execution, inputs are being traced and logged into a bug report (A.). This bug report is submitted to a bug report repository. In the laboratory, the developers try to manually reproduce the bug based on the bug report (B.). If the bug can be reproduced, the developers have to manually locate the root-cause of the failure in the source code (C.). After the bug is fixed, the software can be executed with a regression test suite (D.). This way, it is possible to ensure that no new bugs are added during the fixing process.

As presented in Sect. 4.1, all steps of the workflow are usually time-consuming. Our approach presented in the following sections shows that most steps can be supported by automated tools. Automated tools to support bug reproduction (B.) and

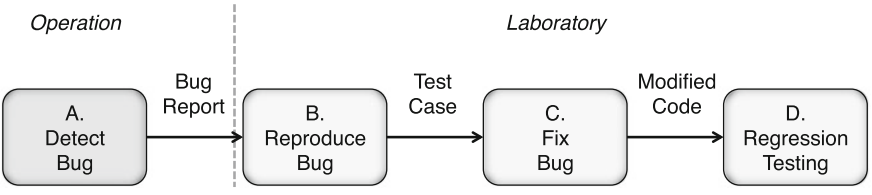

Fig. 4.1 Overview of manual debugging

to generate regression tests (D.) are presented in Sect. 4.3. The bug detection of multi-threaded bugs (A.) is supported by tools presented in Sect. 4.3 as well. Automated bug analyses to support manual debugging (C.) are presented in Sects. 4.4 and 4.5.

4.3 Debugger-Based Bug Reproduction

This section presents approaches of tools for the automated support of bug reproduction. For reproducing bugs, the sensor inputs have to be captured during operation. During replay, the same sensor inputs (e.g., from a GPS or touch screen) have to be triggered or injected to achieve the same execution or instruction sequence. The normal execution of software is deterministic. The same instructions are executed when a software runs with the same inputs. However, these inputs are nondeterministic and are not exactly the same when executing a software twice. For example, it is difficult to achieve the same movements on a touch screen between two test executions. Minimal differences between executions can be determinant whether a failure is triggered. Figure 4.2 shows the different inputs of the software under test (SUT). Zeller [41] lists the nondeterministic inputs of software which are described below. Sensors are the most common input source for embedded software (e.g., from a GPS sensor). User interactions are triggered with human interface devices (e.g., a touch screen). Static data may be stored and read from a disk or flash memory (e.g., an XML configuration file). Access to time or randomness functions in the hardware can change the execution and thus make reproduction difficult. Network interactions may be used for communicating with other devices in the embedded system (e.g., on a CAN bus). The operating environment may be represented by an operating system which controls schedules and memory management. Physical effects may cause hardware changes and they are the most difficult divergence to handle. In the perspective of the SUT, the sources of nondeterminism are categorized into [34]: OS SDK accesses (like system calls), signals or interrupts, specific processor functions, schedules or shared-memory access orderings as well as memory initializations and memory allocations (presented in Fig. 4.2 in the box around the SUT box).

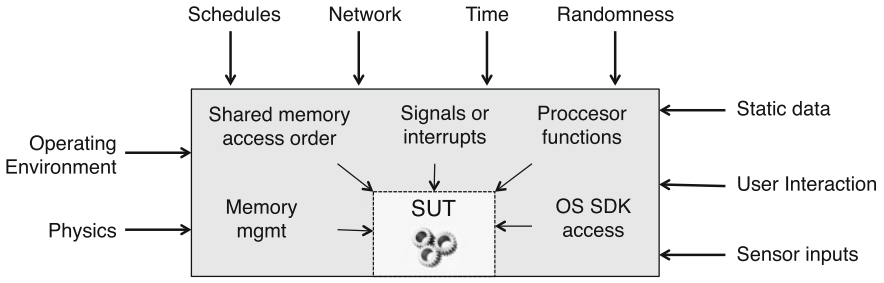

Fig. 4.2 Types of inputs to embedded software

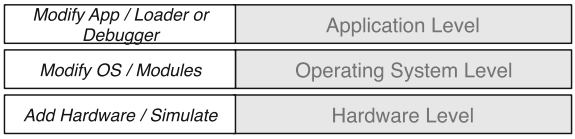

Fig. 4.3 Different levels of replay modules

4.3.1 State of the Art

This section presents current state-of-the-art approaches for the automated reproduction of bugs. The tracing and replaying of the different events caused by the presented sources of nondeterminism can be implemented on different levels (see Fig. 4.3).

It is possible to record/replay the execution of the software on hardware level (Sect. 4.3.1.1). Therefore, additional special hardware is added to support tracing and replay. Another option is to simulate the hardware. In the simulation platform, capturing modules can be integrated. The operating system has control between software and hardware and can record/replay events as well (Sect. 4.3.1.2). It is possible to modify the operating system and to install it on the target platform. Some operating systems provide support to integrate new modules. Furthermore, the software can be modified to record/replay events on application level (Sect. 4.3.1.3). Therefore, the application is modified on source code or binary code level. Debugger tools provide a layer between software and OS and can therefore record/replay events, like presented in Sects. 4.3.2 and 4.3.3. The different approaches are presented in the following sections according to every level.

4.3.1.1 Hardware Level Replay

We examine three types of approaches in the area of hardware level replay: hardware-supported replay, full circuit replay, and virtualization-based replay. Most **hardware-supported replay** approaches consider multiprocessor platforms [21]. To achieve similar executions, the access to shared memory between different cores is recorded and brought into the same sequence during replay. The tracing of shared-memory accesses can be implemented with additional hardware. The advantage of this method compared to software-based approaches is the low overhead achieved by hardware-based functions. **Full circuit replay** approaches add debug instrumentations into the circuits for the FPGA synthesis. This way, it is possible to record and replay the data flow of FPGA circuits in real time [17]. However, only a portion of the program execution can be captured in real time. Other approaches implement **virtualization-based replay** where the hardware is simulated. Moreover, there are approaches for the virtual prototype platform Quick Emulator (QEMU) [10]. However, virtual prototype platforms have a large base overhead which can disturb the interaction with the connected sensor hardware.

4.3.1.2 Operating System Level Replay

We examine three different approaches for operating system level replay: event sequence replay, synchronized event replay, and cycle-accurate event replay. For replaying the same **sequence of events**, some approaches use OS SDK specific commands to trace low-level events and to trigger the same sequence during replay. RERAN [19] uses the *getevent* and an own *sendevent* function of the Android SDK for tracing and replaying on Android mobile platforms. RERAN can record and replay the top 100 Android apps without modifications. Complex gesture inputs on the touch screen can be recorded and replayed with low overhead (about 1%). However, there must be functions available in the OS which support event tracing and sending. Different scheduling or parallel executions on several cores can cause another behavior. Parallel executions require the **synchronization of events** to achieve the same access order to shared resources. SCRIBE [27] is implemented as a Linux kernel module for tracing and injecting. It supports multiprocessor execution and achieves the synchronization of parallel accesses using synchronization points. Using such a synchronization, system calls can be brought into the same order during recording and replaying even when running on multiple cores. However, real-time systems have strict timing requirements and require an **instruction-accurate reproduction of events**, e.g., interrupts. RT-Replayer [33] instruments a real-time operating system kernel to trace interrupts. For replaying, the instructions at memory addresses where interrupts occurred are instrumented with trap instructions (similar to breakpoints of a debugger). Thus, the software stops at traps during replay and the same interrupt functions are triggered.

4.3.1.3 Application Level Replay

For tracing and injecting events on application level, we examine three different approaches for application level replay: source code instrumented replay, binary instrumented replay, and checkpoint-based replay. Using **source code instrumentation**, the source code is modified to trace the control flow and the assignments of variables. Jalangi [35] presents an approach to instrument every assignment of variables and to write the assigned values into a trace file during runtime. During replay, the traced variable values are injected. The approach was presented for JavaScript, but is portable to any other programming language. Drawbacks of Jalangi are: high overhead, big trace files as well as possible side effects of the instrumentation. **Dynamic binary code instrumentation** modifies the source code during execution, e.g., the recording. It dynamically instruments the loaded binary code during runtime, and injects additional tracing code. The instrumented code can be highly optimized and only a low overhead is required for the execution of the code. PinPlay [34] uses dynamic binary code instrumentation with the Pin instrumentation framework. It considers several sources of nondeterminism, as presented in the introduction of this section. However, the Pin framework is only available for specific instruction sets. **Checkpointing approaches** capture the current process state in high frequency

(e.g., [40]). Starting at a checkpoint, all nondeterministic events (like system calls) are captured. Checkpoints commonly base on platform-dependent OS SDK operations.

4.3.2 Theory and Algorithms

The previous section presented the way state-of-the-art approaches record and replay bugs on different system levels. This section describes our own approach for tracing and replaying bugs for debugger tools. It considers two sources of nondeterministic inputs: sensor inputs and thread schedules. We do not consider memory violation, because this kind of bug can easily be detected with many available memory profiling tools. We start by presenting the record/replay of sensor accesses followed by the record/replay of thread schedules.

Embedded software often accesses the connected devices triggered by a timer or an interrupt. The device state is requested in a specific frequency. A navigation software may access the GPS sensor with 10 Hz for example. Figure 4.4 shows how a sensor is accessed by a timer function.

The tracing can be implemented by pausing the execution of the software at the location where the access to the device is finished (defined by us as *Receive*). Here, the read data is written to a log file. Figure 4.5 shows this concept.

During the replay the execution is being paused at a location, where the interrupt starts the access to the device (defined by us as *Request*). The access to the sensor is skipped and a jump to the receive location (defined by us as *Receive*) is triggered. At this point, the data is read from the log file. These data are injected into the execution. This way, the sensor data from the recording run is replayed. This concept can be implemented with a debugger tool, as presented in Sect. 4.3.3. The debugger-based replay is illustrated in the sequence diagram in Fig. 4.6.

Other sources of nondeterminism might be thread schedules. The record/replay of thread schedules bases on the reconstruction of sequences of thread events and IO events [28]. This way, the active threads at thread actions, like *sem_wait* and *sem_post*, are monitored. During replay, the same sequences of the invocations of these events are triggered. Listings 1 and 2 show examples of two threads of a

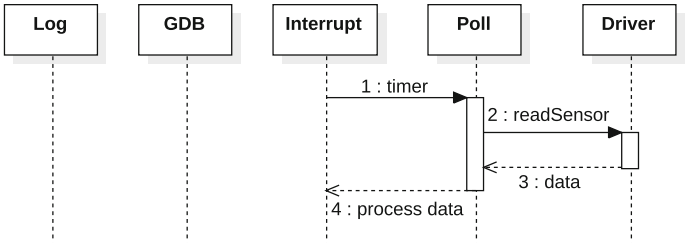

Fig. 4.4 Sequence of sensor accessing

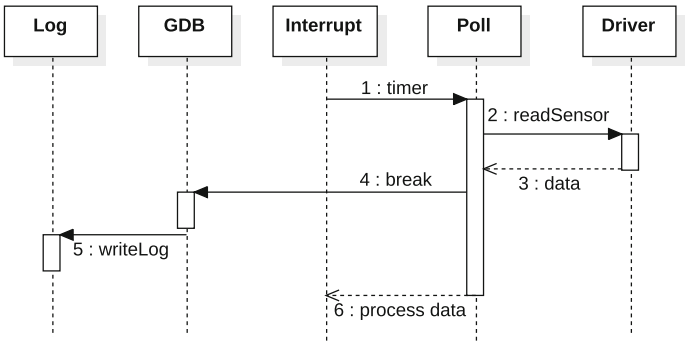

Fig. 4.5 Sequence of recording

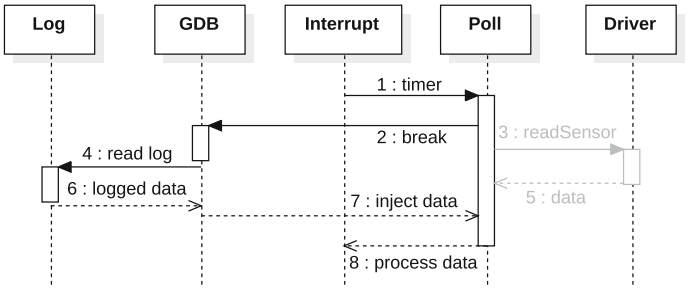

Fig. 4.6 Sequence of replaying

software component for a pedestrian recognition: the thread *Proc* for the recognition of pedestrians in the pictures and the thread *GUI* for drawing rectangles in the pictures where pedestrians were detected. In case the two threads are alternately executed, no failure occurs. However, when thread *Proc* is executed twice, one picture is not drawn. Additionally, when *Proc* is executed twice, the semaphore holds the value two and the *GUI* thread can be executed twice as well. Thus, the current image is released in the thread *GUI* in line 4 and the next call of *drawRec* is triggered with a released image. This case occurs very rarely in normal execution, because, after a long pedestrian recognition in the image in the thread *Proc* (line 2), a thread switch is usually triggered to *GUI*.

Listing 1 Proc

```
1 img=loadImg();
2 recogPed(img);
3 sem_post(sem);
```

Listing 2 GUI

```
1 sem_wait(sem);
2 drawRec(img);
3 showImg(img);
4 releaseImg(img);
```

Our approach implements the serializing of active threads and the randomized switching to threads at thread actions. Therefore, the normal thread scheduler is being locked and only one thread is active at any time. This way, the active thread cannot

Table 4.1 List of tool triggered thread switch actions

Breakpoint	Script actions
sem_init	Set initialization value to the semaphore
sem_wait	If `sem==0` register thread and switch thread, else continue `sem--`
sem_post	Finish post `sem++` and switch to random thread
thread_start	Add available thread into list
thread_end	Delete thread from list
thread_join	Switch to other threads, until joined ones are finished

be preempted by other threads. In our concept, the thread switches are triggered by our tool at thread events (e.g., *sem_wait*, *sem_post*). The scripts monitor the thread events by pausing on the corresponding functions and triggering the thread switches. At each thread event, our tool triggers thread switch actions (see Table 4.1).

Our tool holds a counter for every semaphore *sem*. At every occurrence of *sem_wait*, the algorithm checks whether the semaphore is higher than zero and hence may be passed and the semaphore counter *sem* is decreased. If the semaphore counter is zero, the thread is registered as waiting thread and a switch to another (nonwaiting) thread is triggered. At the occurrence of *sem_post*, the post is finished and a switch to a random thread is triggered. During replay, the same random thread switches are invoked by setting the same seed to the random function like during the tracing. Using this approach, thread switches are always triggered by the scripts. Therefore, the scripts have complete control over thread scheduling.

4.3.3 Implementation

Debugger tools are used to control the execution of the SUT. A popular debugging tool is the GNU Debugger [15], which is available on different embedded platforms. Currently[1], the GDB homepage lists 80 host platforms which are supported by the GDB. Additionally, it is adapted to other platforms by different suppliers. During normal use, the GDB is manually controlled by a developer who types commands into a console terminal. Table 4.2 lists the most frequently used GDB commands.

The GDB is delivered with an external API. Using this API, it is possible to take control over the debugger executions and commands using the Python programming language. Listing 3 shows a simple Python script that can be loaded with the GDB. It starts loading the program to test (line 2) and sets a breakpoint on the method *foo*

[1] Status July 2016

Table 4.2 Thread event actions

Command	Action
file	Loads the program to debug
break	Sets a breakpoint at specific method/line, where execution pauses
run	Starts the execution of the program to debug
continue	Runs the program until next breakpoint
jump	Skips the next lines and jump to a specific location in the code
set	Injects or modifies some variable values
where	Prints current halt point

(line 3). It starts by running the program (line 4). The execution pauses at occurrences of calls of *foo*. The script counts the calls to *foo* (lines 6–8).

Listing 3 Counter GDB Python Script

```
1  import gdb
2  gdb.execute("file␣a.out")
3  gdb.execute("break␣foo")
4  gdb.execute("run")
5  counter=0
6  while True:
7     counter=counter+1
8     gdb.execute("continue")
```

This way, the debugger is controlled with scripted logic. Other debuggers provide different APIs to control the execution of the software under test. Even when the debugger only provides a terminal command interface, these terminal commands can be simulated by scripts and the terminal output can be evaluated as well. Our debugger-based approach records and replays the events using this debugger tool API. Listing 4 shows the way we implemented the replay of GPS sensor data for the Navit navigation software [1].

Listing 4 Record/replay of sensor inputs

```
1  gdb.execute("break␣gpsrequest")
2  gdb.execute("break␣gpsreceive")
3  gdb.execute("run")
4  while running:
5      where=gdb.execute("where",to_string=True)
6      if record and "gpsreceive" in where:
7        lat=gdb.execute("print␣lat",to_string=True)
8        ... # Access lng and angle
9        trace.write(lat,lng,angle)
10     elif not record and "gpsrequest" in where:
11       data=trace.read()
12       gdb.execute("jump␣gpsreceive")
13       gdb.execute("set␣%s"%(data.lat))
14       ...# Set lng and angle
15     gdb.execute("cont",to_string=True)
```

A breakpoint is set in the source code lines where the request to a device is started and where the access to the device is finished (lines 1–2). For the recording, the execution pauses at the location where the GPS data was received. In this case, current GPS values are printed (lines 7–8) and written to a log file (line 9). For the replaying, the execution pauses at the location where the access to the GPS is started. The access is skipped with the *jump* command (line 12) and the data from the log is injected (lines 13–14). If the debugger-based recording is too slow using breakpoints, it can be implemented with *printf* statements or a trace buffer implementation.

The following paragraph presents how the program under test can be controlled to achieve deterministic thread schedules. The GDB provides the commands listed in Table 4.3 for debugging multi-threaded programs. It includes the three commands we used for our implementation. At every breakpoint pause, the developer can manually examine the current thread or can switch to other threads that are contained in the thread list.

Listing 5 presents an implementation for the replay of the pedestrian recognition (and is similar for the game case study presented in Sect. 4.3.4).

Table 4.3 List of GDB commands for handling multiple threads

Command	Action
info threads	Shows current loaded threads
thread	Switch to another thread
set scheduler-locking on	This commands disables the normal thread scheduler

Listing 5 Replay for two-threaded pedestrian recognition

```
 1  gdb.execute("break␣main")
 2  gdb.execute("run")
 3  gdb.execute("set␣scheduler-locking␣on")
 4  gdb.execute("break␣sem_wait")
 5  gdb.execute("break␣sem_post")
 6  random.seed(trace.readSeed())
 7  while running:
 8      where=gdb.execute("where",to_string=True)
 9      if "sem_wait" in where:
10              if sem==0:
11                      gdb.execute("thread␣%s"%Proc)
12              else:
13                      sem=sem-1
14       elif "sem_post" in where:
15              sem=sem+1
16              t=random(Proc,Gui)
17              gdb.execute("thread␣%s"%t)
18       # Sensor data replay code
19      ...
```

The command `"set scheduler-locking on"` disables the current thread scheduler (line 5). When this option is activated, only one thread can be executed at any time. Similar effects can be achieved with portable non-preemptive thread libraries [16]. This way, the parallel execution of threads is serialized. For implementing the required thread switches, our tool sets breakpoints at the used thread actions (see lines 4–5). At every passing of a thread action, our tool triggers the thread switches (Listing 5, line 14–17) with the *thread* command of the GDB based on actions presented in Table 4.1. The control of thread schedules is integrated with the sensor replay (Listing 5, after line 18).

Using randomized switching at thread actions achieves a better thread interleaving coverage than the normal thread scheduler. This way, concurrency bugs can be manifested faster.

4.3.4 Experiments

Figure 4.7 shows our measurements [14] for tracing or recording the sensor input data of the single-threaded software Navit executed on Ubuntu Linux on an X86 Intel platform. The measurements consider a route with 1200 GPS coordinates. These coordinates are read from a file by a Mockup GPS server. The GPS frequency was tested at 50, 33, 20, and 10 Hz, and in parallel, the user cursor inputs (e.g., from touch screen) was captured at 100 Hz. The overhead between the normal execution (labeled as *Normal*) and the recorded execution (labeled as *Rec*) remains nearly

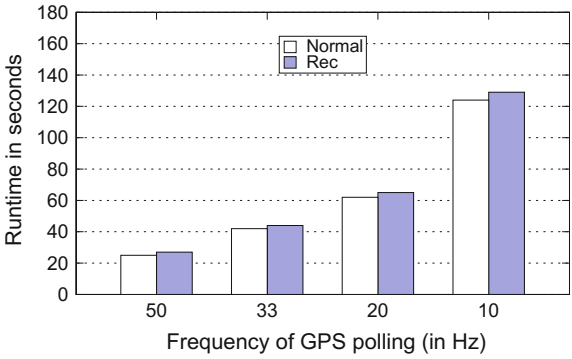

Fig. 4.7 Performance measurements for sensor input recording for single-threaded Navit [14]

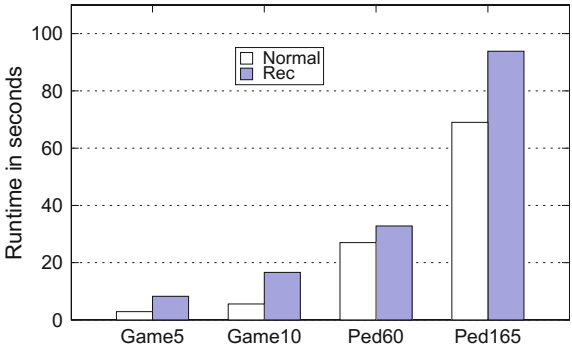

Fig. 4.8 Performance measurements for deterministic scheduling and recording

constant, since the time used to pause the execution at breakpoints is caught up by the invocations of the polling timers. This way, our approach for timer-based software achieves minimal overhead. Tracing optimizations are required for other types of software, e.g., by grouping several inputs and only tracing a set of inputs at once [13].

We tested our approach for deterministic scheduling and recording with two embedded software examples implemented with POSIX threads. The performance measurements for these examples are presented in Fig. 4.8. The first is an ASCII-based fly and shoot game. The game uses two threads, one for drawing the scene and one for reading from the keyboard. The second example is a pedestrian recognition [38] in video data of a vehicle camera (see Sect. 4.3.2). For every example, we used two scenarios for each measurement, a short and a long one. We measured the game until 5 or 10 lives were lost without interaction of the user. We measured the pedestrian recognition with a set of 60 or 165 pictures as inputs. Our experiments were executed five times on an NVIDIA Tegra K1 with ARM CPU and Linux OS.

During recording the scenario of the game, in average 377 thread switches are scheduled for the short scenario and 642 thread switches are scheduled for the longer scenario. For the pedestrian recognition example, in average 186 (short) and 475 (long) thread switches are scheduled. The recording of the pedestrian software requires, in average, 1.22X and 1.36X overhead. The overhead for recording the

game is higher with, in average, 2.86X and 2.98X, because more thread switches have to be triggered in a shorter time.

We observed that the overhead keeps similar for short and long scenarios in both case studies. The measurements show that performant recording can be implemented with minimal effort. Every recording and replay script contains less that 50 lines of source code. Additionally, the pedestrian recognition example shows that concurrency bugs can be detected faster when forcing randomized switches with our tool. Hence in our tests, the example bug is detected in a few seconds with our approach, but it is not triggered during 10×165 picture inputs with the normal thread scheduler. For the performance measurements, we triggered the alternate invocation of the two threads for not triggering the bug.

4.4 Dynamic Verification During Replay

Even if a bug can be reproduced, it is difficult to locate the root-cause of the bug during the replay. Manual debugging of a replay (e.g., with GDB) is effortful. The source code is often implemented by other developers. Thus, it is difficult to comprehend which sequence of actions (e.g., method calls) leads to the failure. Additionally, it is difficult to understand how the faulty sequence of actions is caused. The approaches in the area of runtime verification provide concepts for automatically analyzing executions during runtime by comparing them against a formal specification. Runtime verification tests whether a set of specific properties are held during the execution. The components which observe the execution are called monitors.

4.4.1 State of the Art

The *online monitors* approach runs a monitor in parallel to the execution during operation. Online monitors have to be very efficient, because the normal execution should not be disturbed. However, online monitors may react to observed anomalies during the execution [29]. At the occurrence of anomalies, fail-safe or recovery modes may be activated during operation. *Log monitors* examine log files, captured during a long-term execution of a software. The trace files are efficiently generated during operation. The tracing should be lean or implemented with fast additional hardware for not disturbing the normal execution. Offline, the trace file is analyzed in detail [6]. Wrong event sequences in the trace can be detected, pointing to the failure or even to the root-cause of the failure. Some approaches combine record/replay and dynamic analysis of software [32, 40, 41]. However, they do not use a framework for the implementation of complex assertions and were not tested on embedded platforms.

Table 4.4 presents the advantages and disadvantages of each mode. The two approaches do not provide support to check whether the failure still occurs or not (*Control Test*). Moreover, they cannot be applied fine-grained (*Fine-grained*),

Table 4.4 Comparison of different characteristics for the monitoring types

Type	Online	Log	Replay
Control test	✗	✗	✓
Fine-grained	✗	✗	✓
Recovery	✓	✗	✗
Long term	✓	✓	✗

because they would disturb the normal interaction with the user or with other systems. Our replay approach fulfills the first two categories (*Control Test* and *Fine-grained*). However, the activation of *recoveries* is only possible with the online mode as summarized in Table 4.4. *Long-term* tracing and monitoring is, in our opinion, best applicable with tracing logs.

4.4.2 Theory and Workflow

The concept of applying dynamic verification during replay [12] is based on the concept of tracing only the relevant inputs to the software and replaying them offline. During replay, fine-granular tracing can be executed. Monitors or analysis can check these traces for anomalies. During replay, the requirements for efficient tracing and monitoring are less compared to normal operation. Additionally, the generated replay can be used later as a control replay, after the bug has been fixed. The replay concept is, in our opinion, the best option for system testing, because control replays which can be used as regression test cases later are generated. Figure 4.9 shows how the different manual steps are supported or replaced by automated tools. The detection of multi-threaded bugs (A.) is optimized by the randomized scheduling concepts presented in Sect. 4.3.3. The replay of bugs (B.) is automized (see Sect. 4.3) and can be used as regression test (D.). Automated analyses during replay (B.) support the manual bug fixing (C.). These analysis tools are presented in the following sections.

A. Systematic Bug Detection: The software is tested in real-world operation. The incoming events to the software are captured during these tests. Recording mecha-

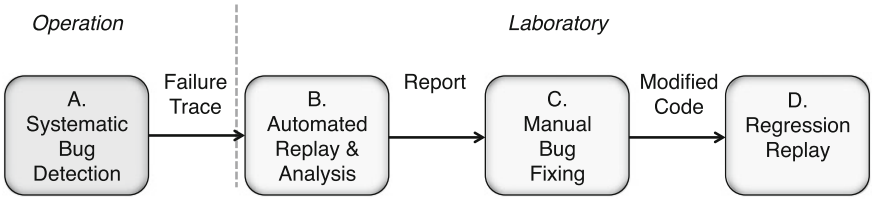

Fig. 4.9 Workflow for debugger-based dynamic verification during replay

nisms are implemented with a symbolic debugger in order to avoid instrumentation and to achieve platform compatibility. Therefore, the debugger is controlled by a script. The developer decides which events are relevant and have to be captured. Thus, the recording can be kept lean. To efficiently detect multi-threaded bugs, the thread scheduler is controlled by our scripts triggering randomized switches at any thread event.

B. Automated Replay and Analysis: The failure sequence can be loaded and deterministically replayed in the laboratory. The replay mechanisms are implemented with portable debugger tools. The software is executed with the debugging interface on the same hardware as during operation for arranging the same system behavior as during the original run. During replay, the failure occurs again based on the deterministic replay. Manually debugging the complete execution sequence or even several processing paths of events is very time-consuming. Therefore, we apply dynamic verification during replay to automatically detect potential anomalies. These information can give a hint to the fault. Analyses performed online during operation disturb the normal execution, but do not cause drawbacks during replay, because no interactions with the user or with external components are required for the execution of the replay.

C. Manual Bug Fixing: Based on the report of the dynamic verification, the developer can manually fix the fault. The step results in a patched program.

D. Regression Replay: The modified program can be tested with a control replay. It is executed with the recorded sequence of failure inputs to observe whether the faulty behavior occurs again. Finally, the bug replay can be archived as a test case for a regression test suite.

4.4.3 Implementation of Assertions During Replay

In our workflow presented in the previous section, dynamic verification is applied during replay for detecting the cause of the bug. This section shows how this cause of a bug can be detected using assertions. Such assertions can be easily implemented with a debugger. Therefore, during debugger-based replay, the execution can be monitored with the debugger as well. The following paragraph considers the multi-threaded replay of a pedestrian recognition software (as presented in Sect. 4.3). During the replay of the software, the event sequence can be analyzed using assertion-based verification. The sequence of method calls is monitored by setting breakpoints on the corresponding methods. In the pedestrian recognition example, three events are relevant: **recogPed()**, **drawRec()**, and **showImg()**. Temporal conditions can be checked during replay, implemented with method breakpoints or watchpoints. The automaton in Fig. 4.10 checks whether the correct sequence for loading and processing the images is called. If another transition occurs, a specification violation is detected.

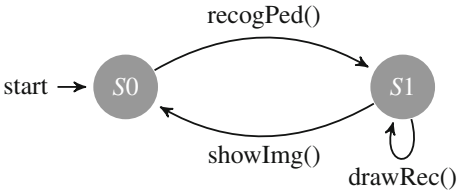

Fig. 4.10 Automaton for the action sequence of the pedestrian recognition

This kind of monitor can easily be implemented in a GDB Python script (see Listing 6). Lines 1–3 set the breakpoints and watchpoints on the presented conditions. Lines 6–14 check the reached point and the current state.

Listing 6 Runtime Verification GDB Python script

```
 1  gdb.execute("break␣recogPed")
 2  gdb.execute("break␣drawRec")
 3  gdb.execute("break␣showImg")
 4  state=0
 5  while running:
 6              where=gdb.execute("where",to_string=True)
 7              if "drawRec" in where and state==1:
 8                      state=1
 9              elif "recogPed" in where and state==0:
10                      state=1
11              elif "showImg" in where and state==1:
12                      state=0
13              else:
14                      print "Spec␣violation!"
...             # Sensor and Thread Replay Code
```

4.4.4 Experiments

Figure 4.11 shows our performance measurements for different monitoring scenarios compared to the multi-threaded recording overhead.

We measured the runtime when 2, 5, and 10% of all methods of the corresponding software were monitored using the GDB (this represents the *online monitor* or *log monitor* mode). The monitored methods can be compared to parallel running state machines as presented in the previous section. We randomly selected a specific percentage (2, 5, or 10%) of methods and set breakpoints on them. We measured the runtime for executing our multi-threaded recording approach (labeled as *Record*) as well. We used the same four scenarios as in Sect. 4.3: the fly and shoot game with 5 or 10 lives and the pedestrian recognition with 60 or 165 video pictures. In all scenarios, the monitoring of 2% of the methods is faster than the recording.

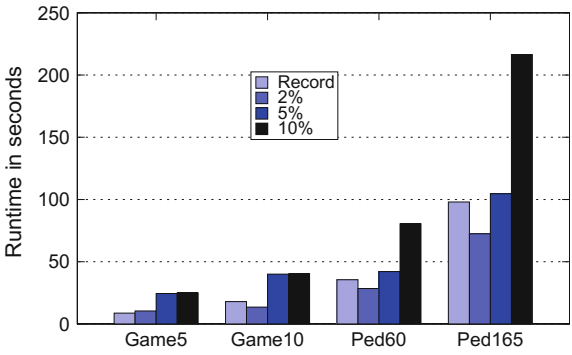

Fig. 4.11 Comparing the overhead for our recording with methods monitoring overhead

Table 4.5 Number of GDB actions for recording and methods monitoring

Scenario	Record	2%	5%	10%
Game5	383	7069	13050	13333
Game10	810	7069	18252	18840
Ped60	181	910	7099	17143
Ped165	496	2117	17546	47593

The monitoring of 5% of the methods is slower than the recording, especially when considering the game scenarios. The recording of the execution of the scenarios is in average 2.45 times faster than the runtime for monitoring of 10% methods. However, the performance of method monitoring mainly depends on how often every method occurs in the execution. Table 4.5 shows the amount of GDB pauses for the recording scenarios and every method monitoring scenario (2, 5 and 10%).

The higher the number of pauses, the higher the recording or monitoring time. However, the recording had a smaller number of pauses compared to the monitoring scenarios, the proportional runtime overhead was higher. The reason for this finding is that more GDB commands are required at every breakpoint pause for implementing the recording. Additionally, we tested the monitoring of 15% of methods of the pedestrian recognition, but the monitoring of the first picture was not finished after 10 min.

4.5 Root-Cause Analyses

In the previous section, we showed how wrong action sequences can be detected by comparing the executed actions with a specification property. The bug was triggered by the wrong sequence causing the program to crash. However, in many cases, a correct or complete specification of actions is not available. Usually, this is the case, as a specification property often already points to a potential failure, which can be fixed manually. In this section, we consider semantic bugs (in software without a

specification), i.e., the root-cause of the bug is found in the value processing or program logic. We present concepts to detect wrong or missing method calls in processing and to identify the root-cause of the corresponding error logic.

4.5.1 State of the Art

This section presents the state of the art in the area of fault localization of noncrashing bugs and embedded software monitoring.

4.5.1.1 Delta Debugging

The book 'Why Programs Fail' [41] presents different concepts for fault localization in software. It describes several concepts for dynamic analyses, including **delta debugging** and **anomaly detection**. Some of these concepts are similarly considered in our work, e.g., for our delta computation approach. Later work of Burger and Zeller [7] developed **dynamic slicing** for the localization of noncrashing bugs. They apply several steps to isolate the failure location by following back the bug in the execution. However, delta debugging bases on experiments with the program to automatically generate passing runs and failing runs, which is difficult and runtime-consuming in embedded contexts (like mentioned by [3]).

4.5.1.2 Dynamic Verification for Noncrashing Bugs

Zhang et al. [42] implement an approach to detect noncrashing bugs caused by wrong configurations. It profiles the execution of failing and not failing configurations. Many embedded softwares do not even require a configuration and the bugs can be located in the source code. Liu et al. [31] apply **support vector machines** to categorize passing runs and failing runs of noncrashing bugs. Their approach generates behavior graphs on method level to compare different runs. For classification, a lot of input runs are required and only suspect methods can be detected (not the relevant source code lines). Abreu [2] implement fault localization for embedded software using **spectrum-based coverage analyses**. They apply model-based diagnosis to improve the results of the analyses. Similar to Tarantula [25], the coverage of executed statements of each failing run is compared to the passing runs. They assume that a big set of test cases of failing and passing runs are available. However, all approaches require a set of failing runs, which can be used for classification. In our use case for system testing, a big set of nonfailure runs and failure runs is not available for classification.

4.5.1.3 Monitoring of Embedded Software

For the implementation of fault localization, the software has to be monitored. Amiar et al. [3] use special **tracing hardware** to monitor embedded software. They apply spectrum-based coverage analyses on a single trace. When detecting a failing cycle, it is compared to previous similar ones to detect spectrum-based coverage deltas. However, they assume a tracing hardware for the specific embedded platform is available. Such hardware is usually expensive. Several runtime verification approaches [20, 36] use **cheap debugger interfaces** with the GNU Debugger (GDB) [15] to achieve platform compatibility, but they do not present a concept to detect bugs without a specification. FLOMA [23] observes the software fine-grained using **probabilistic sampling**, but it does not monitor every source code line. It randomly decides if a specific execution step is monitored. However, probabilistic sampling can miss important steps and FLOMA requires the instrumentation of the source code. Zuo et al. [43] present a **hierarchical instrumentation** approach to accelerate monitoring. Their approach instruments the software to monitor and analyze the method call sequences. Afterwards, only the suspect parts are monitored on source code line level. This way, the monitoring can be accelerated. Our approach extends this approach and applies it to debugger tools.

4.5.2 Theory and Concepts

We apply root-cause analysis on a failure replay to automatically detect suspect source code lines which are potential root-causes of the failure (based on [3, 13, 14]). This analysis results in a report, which can give the developer a hint where the bug is located in the source code. In the following, we present a workflow which bases on a failure replay and a nonfailure replay. We split the execution of the software into parts (see Sect. 4.5.2.1). Several executions of a partition have overlaps and can be compared. Every execution of one partition is called a run. Afterwards, this run in the replay which executes the failure has to be detected (see Sect. 4.5.2.2). The failure run in the replay is compared to several runs in the replay which are similar and which are not categorized as failing runs (see Sect. 4.5.2.3). For the comparison of the failure run to the similar runs, we apply fine-granular analyses on source line level, aiming at detecting the buggy source code line. We show metrics for coverage analysis as well as invariant generation analysis (see Sect. 4.5.2.4). However, fine-grained monitoring can be very slow using cheap debugger interfaces. Therefore, we present an acceleration approach in Sect. 4.5.2.5.

We exemplify our own approach with a noncrashing bug in the open source navigation software Navit [1]. If Navit receives GPS sensor data with an angle smaller than -360, the vehicle pointer is not drawn for a short amount of time. This bug might disturb the driver, e.g., when looking for the correct crossing on a busy street. This bug does not throw an exception. It can just briefly be observed in the GUI.

It is caused by the processing of wrong sensor data (the sensor sends angle values <-360). Such a case occurs when the software is not compatible with the sensor hardware outputs. The bug is caused by a wrong calculation in the Navit software (presented in the following).

4.5.2.1 Partition Replay into Runs

The approaches of the state of the art in the area of anomaly detection provide concepts for detecting root-causes by comparing nonfailing with failing execution runs of the software under test [41]. However, the execution of complex embedded software may contain parts which execute different functionalites. The comparison of different functionalities can cause many false positives, especially when only a small set of reference runs is available. In our approach, we split the execution of the embedded software in several comparable parts (executing similar functionalities). Embedded software usually processes sensor data to update the program states. This processing is usually very similar every time it is executed. Figure 4.12 exemplarily presents the concept for the replay of Navit. A replay of Navit contains different types of processing, e.g., GPS processing, touch screen input processing or traffic data processing.

The processing of sensor data starts after the data has been read from the sensor hardware (in Sect. 4.3 defined as the point *receive*). A processing is represented by the following tuple:

$$Processing = (Start, Run, End) \tag{4.1}$$

Start and *End* represent source code lines passed in the execution. *Start* is the position in the execution where the system starts to process this sensor data. *End* is the position in the execution where the processing of the sensor data is finished. The processing *Run* includes a list of all operations *Ops* and method calls *M* in the processing of the sensor data:

$$Run = (M, Ops) \tag{4.2}$$

Fig. 4.12 Partitioning of a replay into runs

and

$$M = \{m_1, m_2, \ldots, m_{N1}\} \tag{4.3}$$

$$Ops = \{op_1, op_2, \ldots, op_{N2}\} \tag{4.4}$$

$$op_x = \{v_1, v_2, \ldots, v_{N3}\} \tag{4.5}$$

In our approach, the execution breaks at *Start*. Beginning at this breakpoint, the processing is observed either on method or on source code line level. On method level, every method $m_i \in M$ is monitored (4.3). On source code line level, every operation $op_i \in Op$ is monitored (4.4). The monitoring stops with the execution of *End*. We consider one statement or source code line as operation op_i. Each operation op_x holds several global, local, and argument variables $v_1, v_2, \ldots, v_{N3}$ (4.5). The runs of the same type can be compared by detecting differences considering the executed methods, the executed operations or the observed variable values. In our current approach, *Start* and *End* have to be specified with debugger scripts. Other approaches implement automated cycle detection [3], which can be similarly applied to our approach.

4.5.2.2 Detect the Failure Run

Every execution of the partition of the software is considered a run of a replay. When running the failure replay, one or more runs of a specific sensor processing cause the observed failure. We present a lightweight concept for the detection of the failure run in the replay. It classifies those runs as failing which are most different to the runs in a nonfailure replay (as described in the following). Every run of a replay can be compared to other runs, because similar operations and methods are executed. Differences in the run may point to the failure. To detect the failure run, our approach expects two replays. One replay which causes the failure and a second replay which does not cause the failure. The runs of a replay with a failure can be compared to the runs of a replay not causing an observed failure. Two runs can be compared by checking which source code lines or methods are covered by a run. As being presented in Sect. 4.5.2.5, it is more efficient to consider the coverage of methods in this stage.

Table 4.6 shows an example matrix (representing the Navit bug), which contains the coverage of methods of every run of a replay (like presented by [3] for traces).

The callable methods are represented in the rows. The runs are represented in the columns. The value 1 in a cell means that the method in this row is executed by the run in the corresponding column. The difference of two runs can be compared using the hamming distance, i.e., by counting the differences in rows between the two columns of runs. In our example: $distance(Run1, Run7) = 2$, $distance(Run2, Run7) = 3$ and $distance(Run3, Run7) = 3$. For a straightforward presentation, we consider some pseudo-methods $m1 - m4$ and the *draw* method.

Table 4.6 Coverage matrix for monitored runs of a replay

Methd/run	Run_1	Run_2	Run_3	...	Run_7
m_1	1	1	1	…	1
m_2	1	0	1	…	1
m_3	1	1	0	…	1
m_4	1	1	1	…	0
draw	1	1	1	…	0
…	…	…	…	…	…

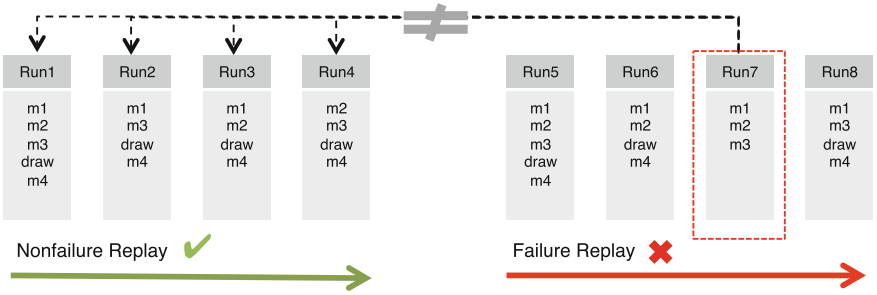

Fig. 4.13 Detect the failure run

The failure is detected by comparing every run in the failure replay with every run in the nonfailure replay using the matrix presented above and the hamming distance. The run in the failure replay which is not similar (or most different) to all runs in nonfailure replay is considered a failure run. Figure 4.13 shows how *Run*7 in the failure replay is compared to every run in the nonfailure replay. *Run*7 is most different to the nonfailure runs, because the call of the *draw* method of the vehicle pointer is missing. The number of occurrences of every method can be combined with the hamming distance as well. Differences between two cells higher than a threshold *threshold* can be counted as 1 and else as 0. This way, one method of two runs can be categorized as different if one run executes a method many more times than the other run. It is possible to consider the call sequence as well. However, the method coverage can be monitored faster than source code line coverage (see Sect. 4.5.4). Additionally, it is possible to categorize several runs as failing runs. For example, it is possible to categorize those ones as failure runs which show 20% differences when compared to the runs of a nonfailure replay. However, the following explanations base on one failure run. Note: If several runs are ranked with same distance, the latest is chosen (because the bug is expected to be near the end of the replay).

Here, *Run*7 in the failure replay is most different to all other runs in the nonfailure replay. That means: *Run*5, *Run*6, and *Run*8 have a corresponding run in the nonfailure replay with a hamming distance, which is smaller than the hamming distances of *Run*7 to each run of the nonfailure replay.

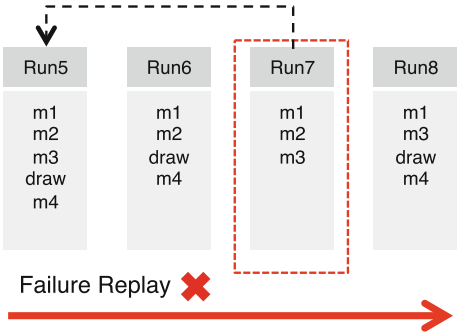

Fig. 4.14 Detect the similar runs

4.5.2.3 Detect Similar Runs

In the previous section, we described the concept for detecting the failure run. Anomalies can be detected by comparing this failure run to runs with no failure. Therefore, our approach detects several runs in the failure replay which are similar to the failure run, but which are not categorized as failing runs, i.e., it detects those runs which have similar method coverage to the failing run using the hamming distance (bases on [3]). Figure 4.14 shows the failure replay. In this example, the most similar run to the failure run *Run*7 is *Run*5. This way, the failure run is compared in detail to the runs which are most similar to it. This concept bases on the nearest neighbor model, which was similarly applied to log files of a tracing hardware [3]. Our approach detects similar runs in the same replay where the failure run occurs at, these similar runs being executed under the same context as the failure run (e.g., considering the configuration context).

In our tests, we detected three runs which are similar to the failure run. These runs and the failing run are compared in detail using delta analysis.

4.5.2.4 Delta Computation

The failing run and the similar runs are compared in detail to detect deltas which can point to the failure root-cause. When comparing the similar runs to the failing run in the failure replay, different metrics can be applied to identify suspect source code lines. We present the concepts and metrics for the delta analyses based on the Navit bug example.

The Navit bug is based on a wrong calculation in the GPS processing (pseudocode presented in Listing 7). If the angle is smaller than 0, the value 360 is added to the angle in line 2. However, in the case the angle is smaller than -360, the angle keeps a negative value after line 2 and lines $5 + 6$ are skipped and the vehicle pointer is not drawn. In a correct implementation, a mathematical modulo operation should be applied to the angle computation to generate a positive value. Line 5 and the parameter variable *lazy* are explained in the following section.

In our example, the following GPS input sequence to the *vehicle_update* method triggers the bug (…{42, 9, 40, 0}, {42, 9, 55, 0}, {42, 9, −370, 0}, {42, 9, 70, 0}). Here, the third input sends wrong angle data, e.g., from a sensor device.

Listing 7 Navit bug example

```
0 void vehicle_update(latitude,longitude,angle,lazy)
1       if (angle<0)
2               angle+=360;
3       ...
4       if (angle in range){
5               setDrawingMode(lazy);
6               draw();
7       }
8       return
9 }
```

We apply fault localization metrics to detect the root-cause of such bugs in the failure replay (in the example, the wrong calculation in line 2). We define these metrics with three coefficients, which are generated for every executable source code line. These coefficients count the number of runs which fulfill a specific characteristic for a specific source code line *op*. These characteristics define the amount of runs causing a failure or not failure. And they define whether they cover the specific source code line *op* or not.

- e_p: #runs with **nonfailure**, which *execute the line*.
- e_f: #runs with **failure**, which *execute the line*.
- n_p: #runs with **nonfailure**, which do *not execute the line*.
- n_f: #runs with **failure**, which do *not execute the line*.

Popular metrics for fault localization were given by Tarantula, Jaccard, and Occhiai. The metrics are defined as follows [39]:

$$Tarantula : d_T = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f} + \frac{e_p}{e_p+n_p}} \tag{4.6}$$

$$Jaccard : d_J = \frac{e_f}{e_f + e_p + n_f} \tag{4.7}$$

$$Occhiai : d_O = \frac{e_f}{\sqrt{(e_f + e_p)(e_f + n_f)}} \tag{4.8}$$

Tarantula measures which lines are primarily executed by failing runs. These lines are considered as more likely to be the root-cause of the failure. The suspicious ranking is decreased if many nonfailing runs execute the line as well. The *Jaccard* coefficient is based on e_f as well, but results in a less suspicious ranking when

many nonfailing runs execute the line or many failing runs do not execute the line. *Occhiai* additionally weights the difference between e_p and n_f. Thus, the ranking is lower, when many nonfailing runs execute the line and many failing runs do not execute the line at the same time. The previously presented metrics mainly consider, which source code lines are often represented in the failing runs. However, in case of noncrashing bugs of embedded software, the error of the bug may be the missing call to an OS SDK library. Additionally, missed source code lines in the failing run may point to the wrong evaluation of conditional logic. A comparison of missing code in the failing run compared to the nonfailing run can point to the failure. Thus, in our opinion, it is required to rank missing features in the failing runs as well. The AMPLE metrics (4.9) consider missing features in the failing run as well [9].

$$AMPLE : d_A = \left\| \frac{e_f}{e_f + n_f} - \frac{e_p}{e_p + n_p} \right\| \tag{4.9}$$

Table 4.7 shows the different results of the metrics for the presented Navit bug example (see Listing 7) for three similar runs (Run_{Sim}) and one failing run (Run_{Fail}). The higher the resulting factor, the higher the suspiciousness of the corresponding source code line. We observed that every metric ranks the operation with index 2 as suspect. However, operations 5 and 6, which additionally point to the wrong conditional case, are not ranked as suspect, despite by AMPLE.

In most fault localization approaches, the metrics result in a list of source code lines with their according suspicious rankings. However, it is difficult to evaluate the source code manually based on this list. In our use case, our tool sets breakpoints on the most suspect source code lines during replay. The developer can step through the suspect lines during replay and check which lines are executed in the failing and the nonfailing runs. Our tool only sets breakpoints on source code lines with AMPLE suspicious ranking 1. Additionally, it notes at every suspect source code

Table 4.7 Calculation of metrics by example

Methd/op	op_1	op_2	op_3	op_4	op_5	op_6
Run_{Sim}	1	0	1	1	1	1
Run_{Sim}	1	0	1	1	1	1
Run_{Sim}	1	0	1	1	1	1
Run_{Fail}	1	1	1	1	0	0
e_p	3	0	3	3	3	3
e_f	1	1	1	1	0	0
n_p	0	3	0	0	0	0
n_f	0	0	0	0	1	1
Tarantula	0.5	1	0.5	0.5	0	0
Jaccard	0.25	1	0.25	0.25	0	0
Occhiai	0.5	1	0.5	0.5	0	0
Ample	0	1	0	0	1	1

line, whether it is executed in the failure run (but not in any similar run) or whether it is executed in every similar run (but not in the failure run). Other approaches propose the combination of suspicious metrics [39], e.g., combining AMPLE and Occhiai. This way, machine learning algorithms generate weighted combinations of different metrics. Studies show that better metric results are achieved with combined metrics. However, a learning phase which is not possible in our use case is required.

Previously, we showed how delta computation is able to show coverage deltas between the failure run in the failure replay and some similar runs in the replay. However, root-causes which start to propagate at a wrong variable assignment can often only be detected by monitoring all variable values in every source code line.

In our example, another sequence of GPS inputs may trigger the bug: ({42, 9, −340, 0}, {42, 9, −355, 0}, {42, 9, $\boxed{-370}$, 0}, {42, 9, −355, 0}). This sequence can happen, when the sensor sends data smaller than 0 and incrementally switches to an angle smaller than −360. Applying coverage-based analyses during the replay of this sequence, the wrong source code line (line 2 in Listing 7) cannot be detected, because line 2 is executed in every run. However, the root-cause can be detected when monitoring all variable values in every source code line.

Anomalies in variable values can be detected using invariants. Invariants are characteristics of variable/value pairs which are stored for each run. The invariants being held by the nonfailing run can be compared with invariants stored from the failing run. Therefore, we automatically generate invariants for the nonfailing and failing run. Range invariants check, which ranges of variable values are observed during a run. In our Navit example, the invariant in Eq. (4.10) is stored by every nonfailure run. This invariant can be checked in the failure run, where it is violated. In our implementation, we generate invariants for every single passed source code line.

$$inv_{line}(angle) = -360 \leq angle \leq 360 \tag{4.10}$$

However, with few reference runs, it is difficult to build range invariants. Relation invariants between variable/value pairs check the relation between two numeric variables. Variable relations are often checked in conditional branches. This relation may be that a variable *angle* is always bigger than a variable *lazy* in a specific source line (4.11).

$$inv_{line}(angle) = lazy < angle \tag{4.11}$$

Our analysis compares each variable value to all other variable values to detect the relations between the variables. In our Navit example, the method *vehicle_update* is always called with the fourth variable *lazy* which defines the drawing mode and is, for most cases, 0 or 1. Comparing the variable *angle* against the variable *lazy*, shows that, before operation 2, $angle < lazy$. After operation 2, $lazy < angle$ is fulfilled for every nonfailure run. However, for the failing run, $angle < (lazy == 0)$ still holds. Figure 4.15 shows the sequence of variable relations between the variables *angle* and *lazy*.

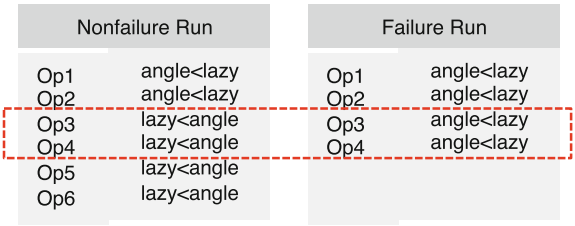

Fig. 4.15 Detecting invariant deltas between two Navit GPS processing

In Fig. 4.15 the wrong relations in lines 3 and 4 in the failure run point exactly to the root-cause of the failure. The resulting analysis report includes anomalies detected by comparing the coverage of the failing run to the similar runs. Additionally, the report includes the generated relation invariants which differ between the failing run and the similar runs.

4.5.2.5 Accelerated Monitoring

This section presents how to accelerate software monitoring based on [13, 43]. Many approaches use special hardware to monitor the embedded software under test. However, this hardware can be expensive or is even not available for new platforms. Therefore, we present an approach on how to optimize the monitoring to achieve fast dynamic verification results. Most developers already use an incremental approach to manually debug software. They first set breakpoints on methods starting to detect anomalies in the method call sequence. Afterwards, they examine suspect methods and they stepwise refine their examination. Our tool achieves accelerated monitoring for bug root-cause analyses, by applying this concept in an automated way. First, we define the basic concept for single-level (SL) monitoring.

Single-Level Monitoring—SL: Monitoring single steps through every executed source code line during a (processing) run. It monitors the current variable values for every monitored source code line.

The analyses presented in the previous section can be applied on the traces generated by *SL* monitoring. However, running single-level monitoring can be slow. The methods of the software are usually executed much less frequently than the source code lines. Therefore, it is usually faster to monitor the methods instead of monitoring every source code line.

MultiLevel Monitoring—ML: In the first replay, the method calls are monitored. The following activities can be applied on this method calls trace: *Detect Failure Run* and *Detect Similar Runs*. In a second replay, the failure run as well as the similar runs are monitored in detail with single-step monitoring. The *Delta Computation* can be applied on this generated trace. This way, all runs despite the failure and the similar runs do not have to be monitored in detail.

The concept is illustrated in Fig. 4.16. It shows two replays: one for method level monitoring and one for monitoring the relevant runs in detail with single-stepping. On method level, first, the failure run is detected (A.) by comparing it to a nonfailure replay, as presented in Sect. 4.5.2.2. The failure run in Fig. 4.16 is *Run*7. Afterwards, the similar runs to the failing run are identified on method level (B.), as presented in Sect. 4.5.2.3. Here, the similar run is *Run*5. However, our approach can detect and handle several similar runs. The difference between the failing run and the similar runs (*Run*5 and *Run*7) is implemented by single-step monitoring (same as *SL*) and analyzing every source code line as presented in Sect. 4.5.2.4. Thus, in the example, *Run*6 and *Run*8 are not monitored in detail.

However, pausing at every passing of method causes a high monitoring time as well (like presented in Sect. 4.5.4), especially when short methods are called in high frequency. Thus, in the following, we propose a concept for debugger-based efficient monitoring of the method coverage of a run.

ML Method Monitoring—MLMethd: Method coverage monitoring traces every executed method in a run only once.

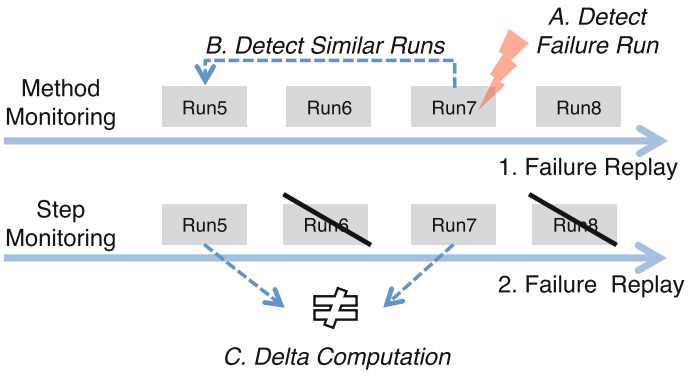

Fig. 4.16 Multi-level (ML) monitoring

Algorithm 1 shows the concept for *MLMethd* in pseudocode. This way, the monitoring of method coverage is efficient, because the monitoring tool only has to trace every method once.

Algorithm 1 Efficient method coverage monitoring for a run

Require: *allmethods* = List of all methods to monitor
Require: *context* = Monitoring context
Ensure: *methodcov* = Set of covered methods in the run
$\forall_{m \in allmethods} context.setBreakpoint(m)$
while $context.nextBreak()$ **do**
$where = context.where()$
$methodcov = methodcov \cup where$
$context.removeBreakpoint(where)$
end while

However, after the failure run is detected and the similar runs are computed, the delta computation step requires a fine-grained trace. The monitoring of this trace can be very slow. A stepwise refinement can accelerate the monitoring. *MLMethd* results in a list of suspect methods (*relmethds*) which are either executed in the failing run or in the similar runs.

ML Backtrace Monitoring—MLBack: The backtraces of the methods in *relmethds* are identified. Every method which occurs in those backtraces is first monitored without stepping into the called methods (side steps). Methods which are executed either in the similar runs or the failing run are not monitored (no comparison is possible). *MLBack* includes *MLMethd*.

For our Navit example, we consider five different methods:

- *update*: Updates the current vehicle state based on GPS input data.
- *route*: Represents the routing calculation.
- *vehi.*: Represents the *vehicle_draw* method.
- *set*: Changes the drawing mode.
- *draw*: Invokes the draw of the vehicle pointer.

Figure 4.17 presents the concept of *MLBack* for the Navit bug. Every black line (or solid line) represents a monitored method. Every red line (or dotted line) represents a not monitored method.

First, a replay monitors the method coverage and detects that the *draw* method misses in the failing run. Several similar runs to the failing run are detected based on method monitoring. During *MLBack*, the methods which occurred in the backtrace of the invocation of *draw* in the nonfailure runs are collected in *relmethds*. During a second replay, the methods in *relmethds* are monitored in the similar runs and the failing run without stepping into the called (or side step) methods. During stepping

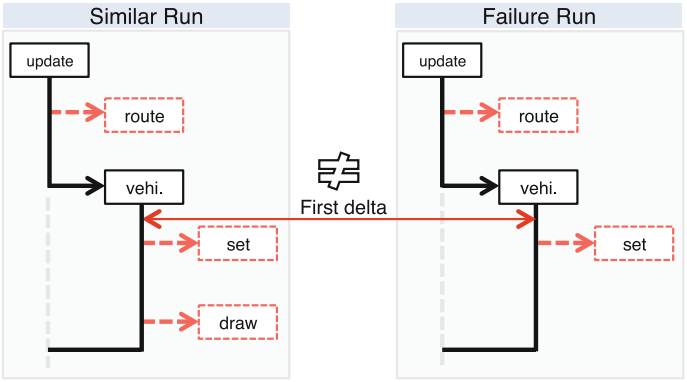

Fig. 4.17 Multi-level backtrace (MLBack) monitoring

through these methods, the values of local variables and method parameter variables are monitored. *MLBack* results in a list of methods which contain anomalies *suspectmethd*. The methods *update* and *vehi.* are monitored without stepping into side steps (here, *route* and *set*). This way, the routing calculation which is not relevant for our considered bug, but would require a lot of operations, is not monitored. Additionally, *draw* is not monitored, because it does only occur in the similar runs, but does not occur in the failure run (and cannot be compared). A first anomaly in the variable values can be detected after the calculation $angle+ = 360$ in *vehi.*

Definition ML Step Monitoring—MLStep: This monitoring concept has a list of suspect methods *suspectmethd* as input. These methods are monitored in an additional replay with the called side steps. Methods which are executed either in the similar runs or the failing run are still not monitored.

In the Navit example, a third replay is executed to additionally monitor the side steps in suspect methods (see Fig. 4.18).

Here, the method *vehi.* is additionally monitored with side steps. This includes the stepping into the method *set* (and some other methods not presented in Fig. 4.18). In an alternative implementation of Navit, the method *set* (or another method *even* which is called by *vehi.*) may include the wrong source code line with $angle+ = 360$, after which wrong relations between *angle* and *lazy* are detected.

4.5.3 Implementation

This section presents how a set of runs in a replay is monitored by single-stepping over every source code line (Single-Stepping Monitoring). On source code level,

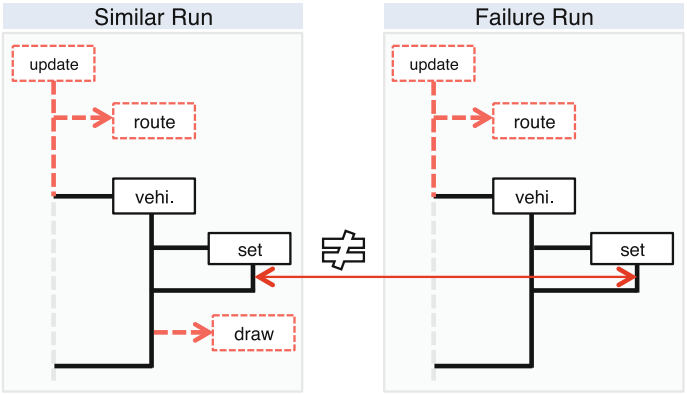

Fig. 4.18 Multi-level step (MLStep) monitoring

the variable values are monitored for every executed source code line in a run. This implementation is required for *SL* as well as for *ML* in the second replay. We show how method level monitoring (Method Coverage Monitoring) is implemented. On method level, the coverage of executed methods in a run is monitored. Additionally, we show how *MLBack* and *MLStep* can be implemented using GDB Python scripts.

4.5.3.1 Single-Stepping Monitoring

Listing 8 shows the implementation for single-stepping monitoring for the Navit example (considering the GPS processing). Lines 1–2 set breakpoints at the locations in the source code where the GPS processing starts and ends. In case the location where the processing starts is reached (line 7), the monitoring is activated (line 8). If the monitoring is activated, every step in the processing is monitored by printing local and argument variables (lines 13–14). The next source code line in the software under test is reached by executing the GDB *step* command (line 16).

4.5.3.2 Method Coverage Monitoring

Listing 9 shows the implementation for method coverage monitoring for the Navit example (considering the GPS processing). Lines 1–2 set breakpoints at the locations in the source code where the GPS processing starts and ends. Additionally, breakpoints are set on every method of the Navit software (lines 5–6); they are disabled at the beginning of the execution (line 7). If the location where the processing starts is reached (line 10), the monitoring is activated (line 11) and, additionally, the breakpoints for every method in the Navit software are enabled (line 12). If only the coverage of executed methods should be monitored, every breakpoint could be disabled after first passing (lines 18–19).

Listing 8 Implementation of single-stepping monitoring

```
gdb.execute("break␣gpsprocessingstart")
gdb.execute("break␣gpsprocessingend")
monitor=False
while running:
  where=gdb.execute("where",to_string=True)
  if "gpsprocessingstart" in where:
    monitor=True
    gdb.execute("step")
  elif "gpsprocessingend" in where:
    monitor=False
  elif monitor:
    locs=gdb.execute("info␣locals",to_string=True)
    args=gdb.execute("info␣args",to_string=True)
    trace.write(where,locs,args)
    gdb.execute("step")

...   # Sensor and Thread Replay Code
```

Listing 9 Implementation of method coverage monitoring

```
gdb.execute("break␣gpsprocessingstart")
gdb.execute("break␣gpsprocessingend")
monitor=False
for m in navitmethods:
   point=gdb.execute("break␣%s"%m)
   disable(point)
while running:
 where=gdb.execute("where",to_string=True)
 if "gpsprocessingstart" in where:
   monitor=True
   enableAllMethodBreakpoints()
 elif "gpsprocessingend" in where:
   monitor=False
   disableAllMethodBreakpoints()
 elif monitor:
   trace.write(where)
   if coverage:
     disable(where)
   gdb.execute("cont")

...   # Sensor and Thread Replay Code
```

4.5.3.3 Method Backtrace Monitoring

Listing 10 shows the implementation for method backtrace monitoring for the Navit example (considering the GPS processing). Lines 2–3 set breakpoints on all methods in *relmethds* reported from previous method monitoring and analysis. These methods are stepped through with the command *next* of the GDB (line 14), while monitoring variable values and without stepping into called methods (side steps). The implementation of *MLStep* is similar, but monitors the methods reported from the *MLBack* analysis. It uses the command *step* of the GDB instead of the command *next*.

Listing 10 Implementation of MLBack

```
 1  ...
 2  for m in relmethd:
 3     point=gdb.execute("break␣%s"%m)
 4     disable(point)
 5  while running:
 6    where=gdb.execute("where",to_string=True)
 7    if "gpsprocessingstart" in where:
 8      monitor=True
 9      for b in breaks:
10        gdb.execute("enable␣%s"%b)
          ...
11    elif monitor:
12      locs=gdb.execute("info␣locals")
          ...
13      trace.write(where,locs,args)
14      gdb.execute("next")

...   # Sensor and Thread Replay Code
```

4.5.4 Experiments

The previous sections showed how dynamic verification supports the developer to analyze the root-causes of bugs. In Sect. 4.5.2.5, we presented optimization techniques to accelerate the monitoring for those analyses. We tested the multi-level monitoring (ML) concept for the Replace tool of the Siemens Test Suite [22]. The Replace program is delivered in 32 different versions and with several test cases. We tested 19 of the first 20 versions, which all contain one bug (we could not manually detect a bug in version 19). To generate a possible random replay, we implemented a replay generator which randomly selects 99 test cases from the 5542 test cases in the Siemens Test Suite fault matrix which do not cause the failure. In the end of the replay, we added as run 100 a failing run which executes the bug. For the random selection of test cases, we used the Python random function with seed 100 for every generated replay. We generated a second replay with 200 nonfailing runs to simulate

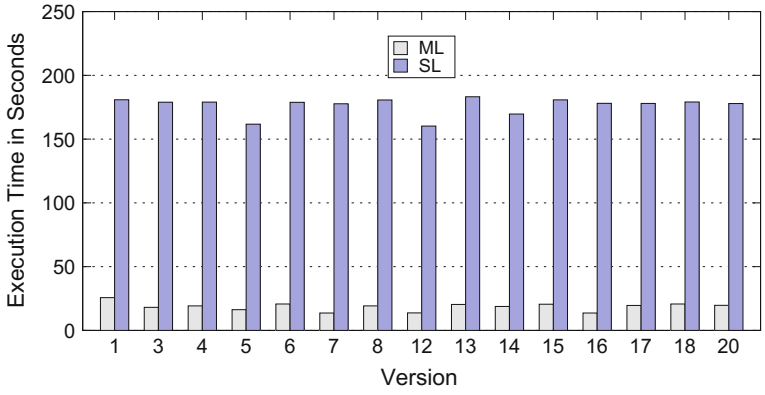

Fig. 4.19 Monitoring runtime for multi-level analyses for replace

noise (using same seed 100). We used this replay as nonfailure replay for the failing run detection presented in Sect. 4.5.2.2. For SL, we monitored the execution of the 100 replay runs collecting source code line coverage as well as numeric local and argument variable values. For ML, the failing run was detected based on method coverage. For this implementation, we additionally used the count of occurrences of a specific method for building hamming words (as presented in Sect. 4.5.2.3 using $threshold = 5$). The three most similar runs to the failure run were determined based on method coverage as well. In a second replay, the failing run and the similar ones were monitored and compared considering line coverage and variable relations. We observed that ML monitoring is much faster than SL monitoring. Figure 4.19 shows the runtime for single-level (SL) monitoring compared to multi-level (ML) monitoring for Replace (for those versions where we could detect the respective bug, using our algorithms).

For a general evaluation, only the monitoring runtimes were included, because the runtimes for analyzing the monitoring result depend on the type of analysis. ML consists of a first replay monitoring on method level and a second replay monitoring on line level. ML and SL monitored all local variables, arguments, and macros for the delta computation for the failing run and the three most similar runs. We found that ML monitoring is much faster than SL monitoring. For Replace, a monitoring acceleration of ∅ 9.5X was achieved. Table 4.8 shows the number of reported suspect lines of SL and ML and the hamming distance from the failing run to the three similar runs for the different Replace versions.

δ-ML/SL presents the count of reported suspect lines. *Fail Detect* shows whether the failing run could be detected on method level. *Dist. Meth.* and *Dist. Line* show the hamming distances to the computed similar runs. *Detect* presents whether the buggy lines are detected with the coverage delta computation or the variable relation delta computation. In six versions (3, 4, 7, 13, 14, 16), the detected similar runs were the same for ML and SL. In these cases, the reported suspect lines were the same. In the 15 experiments, every bug of the version was detected with SL and ML, either pointing to the coverage delta (Cov) or to a delta in variable relations (Rel). Four reports for bugs indirectly pointed to the failure (Version 12—many occurrences of MAXPAT

Table 4.8 Reported delta lines for replace and hamming distances

Example	V1	V3	V4	V5	V6	V7	V8	V12	V13	V14	V15	V16	V17	V18	V20
δ-ML	89	109	117	34	83	71	50	102	119	119	62	71	47	105	47
Fail detect	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓
Dist. Meth.	0,0,1	0,0,0	1,1,1	0,1,1	0,0,0	5,5,5	2,2,2	6,6,6	0,0,0	0,0,0	3,3,3	5,5,5	0,0,0	0,0,0	0,0,0
Dist. line	18,36,36	13,11,11	22,24,24	44,22,11	19,37,11	48,51,51	39,15,45	53,51,66	15,17,17	9,21,17	36,46,43	48,51,51	26,2,9	16,9,21	26,2,9
δ-SL	118	109	117	66	73	71	53	97	119	119	41	71	51	131	51
Dist. line	18,19,23	11,11,13	22,24,24	11,22,22	9,11,11	48,51,51	15,15,15	49,51,53	15,17,17	9,17,21	36,36,38	48,51,51	2,2,5	9,10,12	2,2,5
Detect	Cov	Rel	Rel	Rel	Cov	Rel	Rel	iRel	Cov	iRel	iRel	Rel	Rel	iRel	Rel

in relation deltas, Versions 15 and 18—relation difference of other variables in buggy line, Version 14—difference of relations in enclosed line of if-clause). We tested our tooling with the four other versions of Replace (2, 9, 10, 11), but in these tests, the buggy source code line could not be detected with our delta analysis with ML and SL. Here, the bugs are mainly caused by predicates in if-clauses which access arrays (which are only available in registers). The row *Fail Detect* shows the versions, for which the failing run could be detected on method level. The failing run could not be detected for the versions 6, 14, and 18 with the optimized lightweight classification presented in Sect. 4.5.2.2 based on a randomized selected nonfailure replay. Thus, 63% of the bugs could be automatically and efficiently localized with ML based on one failure replay with 100 runs and one nonfailure replay with 200 runs.

In some cases, delta computation for ML reported less false positives, because the similar runs detected by SL coincidentally caused more variable relation differences. In general, the report quality difference between SL and ML mainly depends on the composition of different runs in the replay. Note: Delta computation (cov+rel) for SL between the failing and every run in the replay would cause higher runtime for parsing and comparison of monitoring results (delta computation runtime without monitoring of three runs was ∅ 13.7 s for Replace).

However, the ML monitoring can still be too slow for monitoring instruction-intensive calculations. For example, the monitoring of a routing calculation of Navit on method level (which calls transformation methods in high frequency), can be very slow when the execution pauses at every method call. We measured the method monitoring for 20 GPS coordinates processing including the routing calculation, which required more than 50 h on an NVIDIA Tegra K1 ARM platform.

Figure 4.20 shows our experiments for the acceleration of the monitoring for root-cause analyses using refinement for the Navit software (*MLBack* and *MLStep*).

We measured a replay with 20 GPS and one with 30 GPS coordinates, both containing the Navit bug from the previous sections. In these experiments, each analysis (*MLBack*, *MLStep* and *SL*) detected the root-cause of the bug. These measurements include the routing calculation of the Navit GPS processing.

MLBack is 1354X faster than *SL* for the 20 GPS scenario. *MLStep* is 334X faster than *SL* for the 20 GPS scenario. For the 30 GPS scenario, the acceleration factors

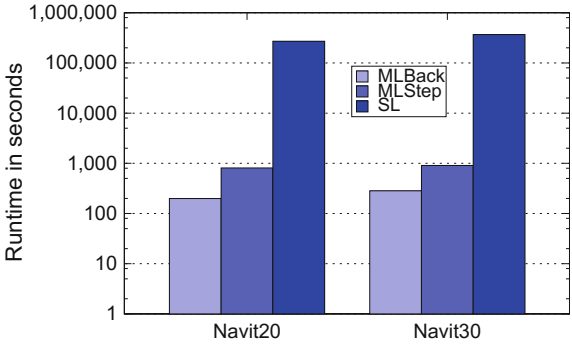

Fig. 4.20 Acceleration of monitoring using refinement for root-cause analyses for the Navit software [13]

are: *MLBack* 1292X and *MLStep* 405X. The acceleration does not increase with the longer sequence, because some GPS processing at the beginning take longer (for recalculating and redrawing of the routing line). The Navit measurements show that analyses with high monitoring overhead can be accelerated with hierarchical refinement, resulting in practicable dynamic verification performance.

4.6 Summary

The approach presented in this chapter showed how the manual debugging process can be supported by automated tools. We presented state-of-the-art approaches for automated bug reproduction. However, these approaches are mainly developed for specific platforms. Therefore, we developed the debugger-based approach which is portable to different embedded platforms. It reproduces sensor inputs and implements randomized thread scheduling for efficient concurrency bug localization. The chapter described how assertions can be implemented with a debugger tool to locate the cause of reproduced concurrency bugs. Afterwards, we showed how the root-cause of bugs can be located without needing specifications or assertions. The root-cause can be tracked down to changes in variable values which cause the bug. Based on a navigation software, we demonstrated how these root-cause localization techniques can be accelerated. This way, the application of slow monitoring tools can be optimized to make them applicable in practice.

The presented approach requires little adaptations for other software. However, the implemented scripts are all very short (every record/replay or monitoring script is shorter than 200 LOC). Thus, the tooling is highly extendable. Additionally, the GDB is supported by most embedded platforms and our script implementations are applicable on most embedded platforms. With few modifications, we could run all our scripts on an ARM Linux the same way as on an X86 Linux.

References

1. Navit-car navigation system. http://www.navit-project.org. Accessed Aug 2016
2. Abreu R (2009) Spectrum-based fault localization in embedded software. PhD thesis, University Delft
3. Amiar A, Delahaye M, Falcone Y, du Bousquet L (2013) Fault localization in embedded software based on a single cyclic trace. In: ISSRE ’13: proceedings of the 24th international automated reproduction and analysis of bugs in embedded software 39 symposium on software reliability engineering. IEEE, pp 148–157
4. Anderson P (2008) The use and limitations of static-analysis tools to improve software quality. CrossTalk J Defence Softw Eng 42(4):18–21
5. Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: ICSE ’06: proceedings of the 28th international conference on software engineering. ACM, pp 361–370
6. Barringer H, Havelund K (2011) Tracecontract: a scala dsl for trace analysis. In: FM ’11: proceedings of the 17th international symposium on formal methods. Springer, pp 57–72
7. Burger M, Zeller A (2011) Minimizing reproduction of software failures. In: Proceedings of 2011 international symposium on software testing and analysis, pp 221–231

8. Charette RN (2009) This car runs on code. IEEE Spectr 21(6)
9. Dallmeier V, Lindig C, Zeller A (2005) Lightweight bug localization with ample. In: AADEBUG '05: proceedings of the sixth international symposium on automated analysis-driven debugging. ACM, pp 99–104
10. Dovgalyuk P (2012) Deterministic replay of system's execution with multi-target qemu simulator for dynamic analysis and reverse debugging. In: CSMR '12: proceedings of the 16th European conference on software maintenance and reengineering. IEEE, pp 553–556
11. Ebert C, Jones C (2009) Embedded software: facts, figures and future. Computer 42(4):42–52
12. Eichelberger H, Kropf T, Greiner T, Rosenstiel W (2013) Runtime verification driven debugging of replayed errors. In: ICTSS '13: proceedings of the PhD workshop of ICTSS'13
13. Eichelberger H, Kropf T, Ruf J, Greiner T, Rosenstiel W (2015) Efficient fault localization during replay of embedded software. In: SEAA '15: proceedings of the 41th euromicro conference series on software engineering and advanced applications. IEEE, pp 43–52
14. Eichelberger H, Ruf J, Kropf T, Greiner T, Rosenstiel W (2014) Debugger-based record replay and dynamic analysis for in-vehicle infotainment. In: ICCSA '14: Proceedings of the 14th international conference on computational science and its applications. Springer, pp 387–401
15. Foundation G (2016) Gdb: the gnu project debugger. http://www.sourceware.org/gdb. Accessed Aug 2016
16. Foundation G (2016) Gnu pth—the gnu portable threads. http://www.gnu.org/software/pth/. Accessed Aug 2016
17. Goeders J, Wilton S (2014) Effective fpga debug for high-level synthesis generated circuits. In: FPL '14: proceedings of the 24th international conference on field programmable logic and applications. IEEE, pp 1–8
18. Goll J (2012) Methoden des software engineering. Springer, Wiesbaden
19. Gomez L, Neamtiu I, Azim T, Millstein T (2013) Reran: timing- and touch-sensitive record and replay for android. In: ICSE '13: proceedings of the 35th international conference on software engineering. ACM, pp 72–81
20. Heckeler P, Eichelberger H, Schlich B, Kropf T, Ruf J, Huster S, Burg S, Rosenstiel W (2013) Accelerated model-based robustness testing of state machine implementations. ACM Appl Comput Rev 13(03):50–67
21. Hower D, Hill M (2008) Rerun: exploiting episodes for lightweight memory race recording. In: ISCA '08: proceedings of the 35th international symposium on computer architecture. ACM/IEEE, pp 265–276
22. Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: ICSE '94: proceedings of the 16th international conference on software engineering. IEEE, pp 191–200
23. Jiang B, Long X, Gao X, Liu Z, Chan W (2011) Floma: statistical fault localization for mobile embedded system. In: ICACC '11: proceedings of the 3rd international conference on advanced computer control. IEEE, pp 396–400
24. Jones C (2012) A short history of the cost per defect metric. http://www.ifpug.org/Documents/Jones-CostPerDefectMetricVersion4.pdf. Accessed Aug 2016
25. Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of 2002 international conference on software engineering, pp 467–477
26. Joorabchi ME, Mirzaaghaei M, Mesbah A (2014) Works for me! characterizing non- reproducible bug reports. In: MSR '14: proceedings of the 11th working conference on mining software repositories. IEEE, pp 62–71
27. Laadan O, Viennot N, Nieh J (2010) Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In: SIGMETRICS '10: proceedings of the 2010 ACM SIGMETRICS international conference on measurement and modeling of computer systems. ACM, pp 155–166
28. Lee YH, Song YW (2010) Replay debugging for multi-threaded embedded software. In: EUC '10: proceedings of the 2010 IEEE international conference on embedded and ubiquitous computing. IEEE, pp 15–22

29. Leucker M, Schallhart C (2009) A brief account of runtime verification. J Logic Algebraic Program 78(5):293–303
30. Liggesmeyer P (2009) Software-qualitaet. Spektrum Akademischer Verlag, Heidelberg
31. Liu C, Yan X, Yu H, Han J, Yu P (2005) Mining behavior graphs for "backtrace" of noncrashing bugs. In: SDM '05: proceedings of the 2005 SIAM international conference on data mining
32. Liu X, Lin W, Pan A, Zhang Z (2007) Wids checker. In: Proceedings of 4th USENIX conference on networked systems design and implementation, pp 257–270
33. Maeng J, Kwon JI, Sin MK, Ryu M (2009) Rt-replayer: a record-replay architecture for embedded real-time software debugging. In: SAC '09: proceedings of the 2009 ACM symposium on applied computing. ACM, pp 1670–1675
34. Patil H, Pereira C, Stallcup M, Lueck G, Cownie J (2010) Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In: CGO '10: proceedings of the 8th international symposium on code generation and optimization. IEEE/ACM, pp 2–11
35. Sen K, Kalasapur S, Brutch T, Gibbs S (2013) Jalangi: a selective record-replay and dynamic analysis framework for javascript. In: ESEC/FSE '13: proceedings of the 9th joint meeting on foundations of software engineering. ACM, pp 488–498
36. Shin H, Endoh Y, Kataoka Y (2007) Arve: aspect-oriented runtime verification environment. In: Proceedings of 2007 runtime verification, pp 87–96
37. Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Imperical Softw Eng 19(6):1665–1705
38. Wu J, Geyer C, Rehg JM (2011) Real-time human detection using contour cues. In: ICRA '11: proceedings of the 2011 international conference on robotics and automation. IEEE, pp 860–867
39. Xuan J, Monperrus M (2014) Learning to combine multiple ranking metrics for fault localization. In: ICSME '14: proceedings of the 30th international conference on software maintenance and evolution. IEEE, pp 191–200
40. Yasushi S (2005) Jockey: a user-space library for record-replay debugging. In: AADEBUG '05: proceedings of the sixth international symposium on automated analysis-driven debugging. ACM, pp 69–76
41. Zeller A (2009) Why programs fail: a guide to systematic debugging, 2nd edn. Morgan Kaufmann Publishers
42. Zhang S, Ernst MD (2013) Automated diagnosis of software configuration errors. In: Proceedings of 2013 international conference on software engineering, pp 312–321
43. Zuo Z, Khoo SC, Sun C (2014) Efficient predicated bug signature mining via hierarchical instrumentation. In: ISSTA '14: proceedings of the 2014 international symposium on software testing and analysis. ACM, pp 215–224