technische universität
dortmund

Masterthesis

Optimizing Software-based Soft-Error Detector Configurations

Robin Thunig
11th March 2022

Supervisors:
Prof. Dr.-Ing. Horst Schirmeier
Prof. Dr.-Ing. Peter Ulbrich

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
`https://ess.cs.tu-dortmund.de`

Abstract

Software developers often include additional code - so-called assertions - in their software, which checks application-specific invariants at runtime and helps to find programming errors. However, assertions can also detect errors that originate from transient hardware faults in memory. Although such an assertion can reduce the occurrence of silent data corruptions (SDCs) in the checked program state, it also increases the runtime of the program and thus the attack surface of the remaining program state. This tradeoff cannot be optimized for a nontrivial number of assertions by enumerating all possible assertion subsets and running a fault injection (FI) campaign for each configuration. Therefore, the goal of this master thesis is the development of a resource-saving method for optimizing assertion configuration that exploits compositionality properties of susceptibility to faults of program parts and requires only a single FI campaign. Based on the FI results of the configuration with all assertions, unknown configurations are to be computed first. Based on this - for a larger number of assertions N, for which even the mere computation of all 2^N configurations becomes impossible - it shall be investigated how the interdependencies of the assertions can be utilized, which allows the optimization of the configuration with integer linear optimization (ILP). The method will be evaluated using the fault injection tool FAIL* and more complex system software, such as the real-time operating system FreeRTOS and eCos. It is shown that with this method for the evaluated programs an average improvement of the fault tolerance of approximately 14% could be achieved in comparison to the original program, in which all assertions are contained.

Contents

1. Introduction

1.1. Motivation and Background

Computers are usually assumed to always work correctly and predictably, as defined in the program code. While this is generally correct, there may be internal and external influences that effect this assumption. Internal influences can be undetected production errors, which become more likely as the structure size of the chips becomes increasingly smaller. While external influences can be radiation particles, which pass through the chips and lead for example to ionization in the transistors. This could eventually cause the falsely activation of the transistors. Especially if these errors occur only very rarely and the system architecture is based on the assumption that exactly what is defined in the code will happen, severe consequences can be the result. This can particularly be the case if it is a Silent Data Corruption that is not detected by software or hardware. Consequently, if data is silently changed, work continues with corrupted data, which can lead to completely unforeseen results that can have catastrophic consequences.

These faults can be detected by hardware or software, if measures are implemented. In hardware redundancy can be used, i.e. multiple processors, sensors, etc. Such redundancy is applied, for instance, in airplanes, since the radiation exposure at cruising altitude is significantly increased and the system is very safety-critical. Another option is software-based hardware fault tolerance (SIHFT), in which faults are detected with the help of software. This variant can also be integrated after the hardware is developed and produced and is therefore also applicable for commercial off-the-shelf hardware.

Since redundant hardware is expensive to develop, SIHFT may be suitable for cubesats that need to be lightweight and inexpensive or for low cost systems in critical areas, such as a voting machine. This thesis will develop methods based on SIHFT.

It will be considered to what extent fault tolerance can be improved by the use of fault detectors. In this thesis the use of already existing assertions will be examined in this regard. The basic assumption will be that assertions make the code more fault tolerant, since injected faults in memory could be detected by checking an assertion.

1.2. Thesis Structure

At first the following chapter 2 explains the foundations necessary for the thesis. This will involve explaining the fault model, fault metric, and fault injection tool used. Then assertions in general will be discussed and it will be shown why assertions might be suitable to improve fault tolerance.

In chapter 3 the use of assertions in the context of fault tolerance will be examined. In particular, the interdependence of assertions will be considered. Requirements for the solution methods will be worked out.

Subsequently, in chapter 4 possible solution methods are presented, with which the fault tolerance is to be maximized with the use of assertions.

In chapter 5 essential aspects of the implementation are discussed. These serve among other things also for evaluation of the presented solution methods.

Subsequently, the presented solutions are evaluated on the basis of different programs in chapter 6. Thereby also more extensive programs are considered, which contain a variety of assertions.

Finally, in chapter 7 a summary of the thesis is given and an outlook on possible future work is presented.

2. Foundations

In the following, some basics are explained that are necessary for the understanding of the thesis. Chapter 2.1 deals with the fault model used in the thesis. Subsequently, chapter 2.2 explains possible metrics that could be used for the evaluation of fault vulnerability. Chapter 2.3 focuses on the fault injection tool FAIL*, which is used for evaluation in this thesis. Finally, the functionality of assertions used for error detection is discussed.

2.1. Fault Model

Due to ever smaller structures on computer chips, the susceptibility to electromagnetic radiation is also increasing. The thesis deals with this type of external influence, which will therefore be described and defined in more detail.

The effect of electromagnetic radiation in the form of for example cosmic rays results in the corruption of data or signals in a processor, memory or other components of a computer. In figure 2.1 an alpha particle or neutron strike is visualized. In this case, the particles lead to the ionization of the bulk substrate of a MOSFET. The released electrons are then collected by the drain and activate the transistor [8]. Such a distortion is also called a soft error or fault. It is characterized by a change that is not permanent, but can be corrected [17]. However, it does not occur systematically in the context of external radiation, but can hit any point on computer components and generate an fault at that point. In addition, it is assumed that such an event occurs so rarely that only one fault appears at a time. This is also referred to as a single-event upset.

In this thesis the fault injection into memory is investigated. The fault model is a single bit flip. This means that a single-event upset occurs in the memory, i.e. the flip of a memory bit by an ionized radiation particle.

Such an event can lead to various outcomes for the program affected by it [17]. Some important ones are explained below:

- **No Effect (OK)**: The program continues to run normally despite the fault and the output of the program is not changed because, for example, the fault occurs in a memory bit that is no longer read or that basically has no influence on the result.

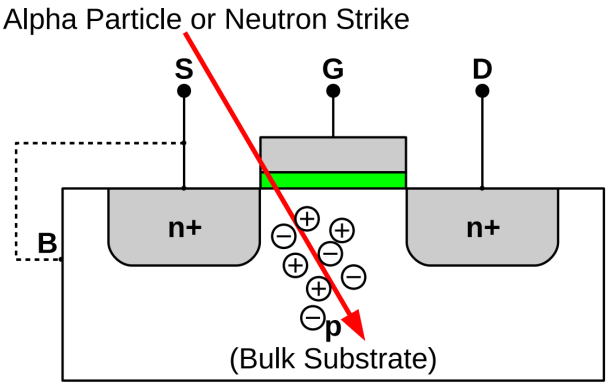

Figure 2.1.: Alpha particle or neutrone creates a trail of ionized bulk substrate atoms in a MOSFET. The electrons will be collected by the drain and lead to an activtion of the transistor [17].

- **Program Crash (TRAP)**: The program terminates prematurely with an exception due to a trap triggered by a fault.
- **Timeout (TIMEOUT)**: The program does not respond any more and must be forced to terminate.
- **Silent Data Corruption (SDC)**: The program does not recognize the fault and continues to run without crashing or a timeout. However, the result is corrupted.
- **Fault is Detected (DETECTED)**: The injected fault is detected by the program or a fault detection mechanism.

Program crash, timeout, or silent data corruption are called a failure and can be catastrophic, depending on the application of the software. For example a SDC was manually detected 2003 in a voting machine in Belgium after the flip of a single bit resulted in 4 096 more votes for a single candidate. The fault was only detected because the candidate got more votes than possible [18].

A fault can occur in any bit of memory at any point in time. For simplicity, assume that instead of occurring at any time, a fault can occur after every CPU cycle. This fault space of a program is visualized in Figure 2.2. It is spanned by the used memory bits on the y-axis and the executed CPU cycles on the x-axis. In Figure 2.3 the faultspace is shown after a fault was injected into each memory bit after each CPU cycle. In the boxes, the result of an injection is visualized.

The faultspace contains the whole attack surface of a program. If it becomes larger by a longer runtime or by more used memory, then the attack surface and

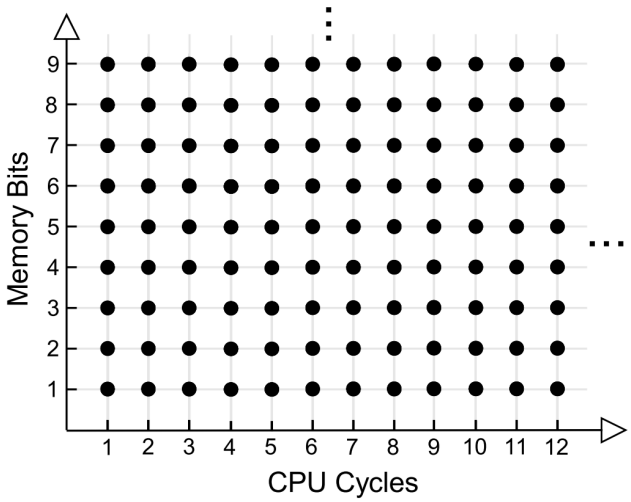

Figure 2.2.: Representation of a fault space for program memory. On the y-axis the memory bits are plotted and on the x-axis the CPU cycles executed until then. Each point visualizes a possible injected fault [17].

thus potentially the fault vulnerability increases. In the thesis, we consider how this fact can affect the evaluation of fault tolerance methods.

2.2. Fault Metrics

In this section, the metrics used in the thesis are explained with which the developed methods will be evaluated. First, in section 2.2.1 a metric commonly used in the literature [13] [14] [16] is explained and the problems arising from the use of this metric are highlighted. Subsequently, in section 2.2.2 an alternative metric is presented that avoids these problems.

2.2.1. Fault Coverage

The fault coverage c describes the probability that a fault injection does not lead to a failure. [15] This is expressed by the following formula:

$$c = P(\text{No Effect}|1\text{ Fault}) = 1 - P(\text{Failure}|1\text{ Fault}) \tag{2.1}$$

The probability can be calculated from the proportion of the number of "No Effect" results to the number of all fault injections. The number of "No Effect" results can be determined from the difference of the number of failures F to the number of all fault injections N, which are applied to a program. This calculation is shown

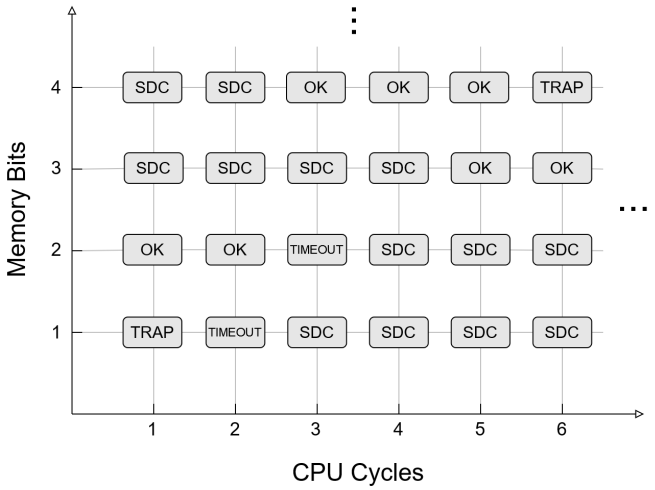

Figure 2.3.: Representation of an injected fault space for program memory. On the y-axis the memory bits are plotted and on the x-axis the CPU cycles executed until then. Each box visualizes a possible result after an injection at this point.

in the following formula:

$$c = 1 - P(\text{Failure}|1\ \text{Fault}) = 1 - \frac{F}{N} \tag{2.2}$$

The problem with this metric is that it ignores the program runtime and the amount of memory used by a program. This leads to the observation that programs evaluated with this metric may appear to improve as the runtime or used program memory becomes larger. This increases the number of possible injections N, but not necessarily the number of failures F. It becomes more clear when a program is assumed to which only NOP instructions are added. However, this increase of N has no effect on the failure behavior of the program and therefore F does not increase. According to the fault coverage this leads to a better fault tolerance even if the program has practically not changed.

2.2.2. Failure Probability

The failure probability [17] describes the probability $P(\text{Failure})$ that a failure occurs in a program. That means this metric considers for a program how high the probability is to produce a failure, if it is assumed that the program is exposed to a continuous and constant particle stream over an arbitrarily large area, which injects faults. The run time and size of the used program memory is considered in this approach, since the program with larger runtime is exposed longer to the

particle stream and with larger program memory a larger surface can be hit. The following approximation can be derived while assuming a constant and low fault rate and rather small programs, given that the goal is to compare the same program [17]:

$$P(\text{Failure}) \propto F \tag{2.3}$$

Accordingly, in order to compare a program with its hardened version, it is necessary to compare the absolute number of failures. In particular, this thesis examines the reduction of silent data corruptions.

This metric is closer to the result of a real radiation experiment than the fault coverage metric and solves its fundamental problem of ignoring runtime and used memory size.

2.3. FAIL*

To test systems for their susceptibility to faults, they can be exposed to real radiation sources and the effects can be observed. However, this has the disadvantage that it is not clear where the radiation particles strike in the system. Furthermore, such an approach is generally not very practical. Therefore, it is more convenient if the radiation effects or, more generally, the fault injections can be simulated precisely. For this purpose the tool FAIL* (Fault Injection Leveraged) [17] is used in this thesis. This tool is able to inject faults into single bits of registers and memory. For further understanding, the process of a fault injection campaign will be briefly explained:

1. **Golden Run**: The program is executed once and each access to memory is traced. In addition, the output of the program is stored as ground truth, which is later compared with the output of the injected program.

2. **Fault space Pruning**: It is utilized that it does not matter if an injection takes place long before a memory area is read or shortly before. It is not necessary to inject into this memory area before every CPU cycle if it is known that the point in time until the next read of this area does not matter. The fault space pruning takes this knowledge into account and reduces the fault space for the injection to the necessary size.

3. **Fault injection**: In this step the fault injections are performed. Faults are injected into each bit one after the other before every CPU cycle. After each injection into exactly one bit at a time, the program is continued to be executed and the effect is recorded.

   To ensure the highest possible parallelism, a server distributes the fault injection experiments to clients, which perform the experiments and send the results back to the server.

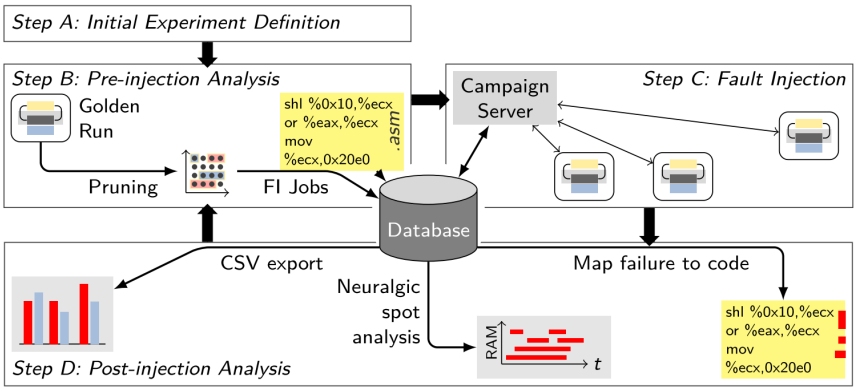

Figure 2.4.: Visualization of Fail*'s assessment-cycle.

4. **Analysis**: After all fault injections have been performed, the results of the injections are filled into a database. On the basis of this database, the information can be further processed. For example, the aggregated number of SDCs can be read out or a fault space plot can be generated.

In figure 2.4 the just described procedure is visualized graphically.

2.4. Assertions

Assertions [2] can be used to ensure that certain conditions are guaranteed during the program runtime. If the condition is no longer respected, the program is terminated with an exception. The condition itself is usually defined by a programmer who wants to ensure that certain error states never occur. In addition, assertions usually have the characteristic to be able to be switched on and off as needed in whole, in order that no overhead results, if they should not be needed for example in the productive application any more. In the following a short program fragment with an assertion is shown:

```
int a = 3;
assert(a < 5);
```

In this section it is checked by an assertion whether the variable a, which should have the value 3, is smaller than 5. This condition is to be fulfilled with each program execution, otherwise the program is terminated with an exception. It should be noted that an assertion usually triggers, i.e. terminates the program with an exception message, if the condition to be checked is false.

This property of the check could be utilized for the detection of faults. If in the program section described above a bit flip occurs in variable a and it changes

to 7, for example, then the assertion would be able to detect this injected fault, since $a < 5$ must hold.

The objective of this thesis is to investigate the use of assertions for fault detection. One advantage of this idea is that existing assertions that programmers have built into their code can be used. Such assertions can be found in most larger software projects, for example in operating systems like the Linux kernel or FreeRTOS.

A possible disadvantage could be the execution time required for the assertion and therefore the larger attack surface. Furthermore, faults may only be detected in certain ranges of values. For example, in the example above, an error is not detected if a changes to the value 1. In this thesis, these advantages and disadvantages will be investigated and discussed.

2.5. Conclusion

In this chapter, the necessary basics for the thesis are explained. First the fault model is specified, which is used in the context of this thesis. In addition, it is described how the fault space is composed and how a fault injection into memory can affect the program. Subsequently, the failure metrics used in the thesis are discussed, which, depending on the choice, can fundamentally change the evaluation of fault tolerance methods. Consequently, the failure probability is used in this thesis, which does not assume the proportion of the occurring failures to all fault injections, but includes both the vulnerability of memory over its size and the program execution time. The probability of a fault is then proportional to the number of SDCs. Afterwards, the functionality of FAIL* is explained, which serves as a fault injection simulation tool for this thesis. Finally, it is described how assertions can be used to detect faults. In particular, this thesis will investigate how many SDCs are detected by assertions during fault injection.

3. Problem Description and Analysis

This chapter examines how assertions can be used in the field of fault tolerance. First, in section 3.1 it is considered how frequently assertions appear in certain operating systems. Afterwards in section 3.2 it is examined how programs consisting of different assertions can behave with respect to fault tolerance. After that, section 3.3 explains the central prior work on which this thesis builds on, as well as other papers that also deal with the topic of assertions in the area of fault tolerance. To further analyze the effect of multiple assertions with respect to fault tolerance, section 3.4 looks at the relations between assertions.

3.1. Existence of Assertions in Operating Systems

This section shows how many assertions are already present in the source code of selected operating systems. For the Linux kernel, its main subsystems (Process Scheduler, Memory Manager, Virtual File System, Network Interface and Inter-Process Communication) were scanned for known assertions.

For the search, an additional bracket is appended to the respective assertion names and there may only be spaces, tabs or an open bracket in front of the name. This ensures that the search using a regular expression only finds occurrences of assertions that actually belong to the code and are not, for example a substring from other assertions or are mentioned in a comment. The search command for bash is then as follows:

```
$ egrep -r "(^|[[:space:]]|\t|\()+<assert-name>\("
```

The occurrence of a selection of frequent assertions in the Linux kernel is shown in table 3.1. Using the same approach, table 3.2 shows the occurrences of the one essential assertion of the FreeRTOS kernel and table 3.3 shows the occurrences of all found assertions of the eCos kernel with more than ten occurrences. It can be seen that a large number of assertions can be found within modern operating systems as well. This fact is crucial for the improvement of fault tolerance of these operating systems by the use of assertions, since in this thesis only already existing assertions are to be considered.

Assertion Name	Occurrences
`assert`	116
`BUG_ON`	3031
`BUILD_BUG_ON`	704
`WARN_ON`	2968
`assert_spin_locked`	158
`lockdep_assert_held`	582
`static_assert`	93
`ubifs_assert`	465
`Total`	8117

Table 3.1.: Table of the occurrences of frequent assertions in the Linux Kernel.

Assertion Name	Occurrences
`assert`	224

Table 3.2.: Table of the occurrences of the central assertion in the FreeRTOS Kernel.

3.2. Effect of Assertions on Fault Tolerance

In section 2.4 it is explained that assertions are able to detect faults. When increasing fault tolerance through assertions, it can also be utilized that a programmer has already implemented them and that they can be found in many software projects, such as FreeRTOS or the Linux kernel. Since assertions can detect faults, it is reasonable to assume that more assertions in a program always make this program more fault tolerant. To disprove this assumption, a simple example consisting of a quicksort algorithm with five assertions can be provided. The algorithm is shown in listing 1 in C code. This quicksort algorithm is adapted from the implementation of the qsort benchmark from the MiBench benchmark suit [9]. Five assertions (`assert1`, `assert2`, `assert3`, `assert4`, `assert5`) are used in this algorithm. To decide whether an assertion has a positive effect on the program's fault tolerance, the program is tried one by one with each possible combination of assertions included in the program. An assertion can either be active, i.e. it can occur and be executed in the program, or it can be inactive, i.e. it can be removed from the program. A combination of active and inactive assertions is called configuration in this thesis. A configuration is defined in the following:

Assertion Name	Occurrences
CYG_ASSERT	1393
CYG_ASSERTC	279
CYG_CHECK_DATA_PTR	241
CYG_CHECK_FUNC_PTR	43
CYG_CHECK_DATA_PTRC	158
CYG_CHECK_FUNC_PTRC	43
CYG_ASSERTCLASS	156
CYG_ASSERTCLASS	156
CYG_ASSERTCLASSO	12
CYG_ASSERT_CLASSC	140
CYG_ASSERT_THISC	12
CYG_PRECONDITION	85
CYG_PRECONDITIONC	231
CYG_POSTCONDITION	24
CYG_POSTCONDITIONC	12
CYG_LOOP_INVARIANT	25
CYG_LOOP_INVARIANTC	26
CYG_PRECONDITION_CLASSC	274
CYG_LOOP_INVARIANT_CLASSC	64
CYG_PRECONDITION_ZERO_OR_CLASSC	19
CYG_PRECONDITION_THISC	482
CYG_POSTCONDITION_THISC	62
CYG_INVARIANT	11
CYG_INVARIANTC	11
CYG_INVARIANT_CLASSC	15
CYG_INVARIANT_CLASSOC	15
CYG_INVARIANT_THISC	24
CYG_ASSERT_DOCALL	34
CYG_FAIL	249
Total	4489

Table 3.3.: Table of the occurrences of all found assertions in the eCos Kernel with more than ten occurrences in the source code.

```
1  void quicksort(char data[], int begin, int end) {
2    assert1(data != 0);
3    if (end > begin) {
4      int pivot = begin;
5      int l = begin + 1;
6      int r = end;
7      while(l < r) {
8        if (data[l] <= data[pivot]) {
9          l += 1;
10       } else if (data[r] > data[pivot])  {
11         r -= 1;
12       } else {
13         swap(data+l, data+r);
14         /* check if swapped elemnts have the right order
15            in regard to the pivot element */
16         assert2(data[l] <= data[pivot] && data[pivot] <= data[r]);
17       }
18       assert3(l <= r);
19     }
20     l -= 1;
21     assert4(data[l] <= data[pivot]);
22     swap(data+pivot, data+l);
23     sort(data, begin, l);
24     sort(data, r, end);
25   }
26 }
27
28 void sort(char data[], input_data_length) {
29   quicksort(input_data, 0, input_data_length - 1);
30   // check if complete array is sorted
31   assert5(is_sorted(input_data, input_data_length));
32 }
```

Listing 1: Code of a quicksort algorithm, which is based on the qsort benchmark from the MiBench benchmark suit [9]. There are five assertions in the code that check in different ways whether a fault has been injected into the variables protected by the assertions.

Definition 3.2.1 (Configuration)
A configuration comprises the states $\boldsymbol{s}$ *of all considered n assertions of a program and is of the form* $[s_1, ..., s_n]$*. There are two possible states. Either an assertion is active, thus is evaluated in the program. This state is called* 1*. Or an assertion is inactive, is accordingly taken out of the program and is not executed. This state is denoted by* 0*.*

A configuration for the quicksort algorithm of the type $[1, 0, 0, 0, 0]$ mentioned above would mean that the assertion `assert1` would be executed in the program and any other assertion would be taken out and will not be executed. When the configuration is applied to the program, it is possible to determine, for example, how many SDCs are the result of an fault injection campaign for the program.

In the following we will define what is meant by an optimal configuration in this thesis:

Definition 3.2.2 (Optimal Configuration)
An optimal configuration describes the configuration of a program that minimizes the number of SDCs when performing a fault injection campaign.

Table 3.4 shows how many SDCs are evaluated for each possible configuration of assertions of the Quicksort algorithm. It can be seen that not the configuration with maximal number of assertions leads to the lowest number of SDCs, but the configuration $[0, 0, 0, 0, 1]$ is the optimal configuration in this case. This means that in order to obtain the lowest number of SDCs, the assertion `assert5` must be activ and executed in this quicksort algorithm and all other assertions must be inactiv.

The above example shows that there is no trivial relation between the choice of assertions and the lowest number of SDCs. It is shown experimentally that for two different programs two different optimal configurations can be the result.

In order to determine the reason why the relation between assertions and fault tolerance does not follow a simple rule, assertions will first be considered in isolation from each other.

In principle, it has already been shown in section 2.4 that an assertion is able to detect faults and therefore reduce the number of SDCs. However, it must also be considered that the assertion must be executed and adds instructions and runtime to the program that would otherwise not be part of it. If faults are injected before each of these instructions, additional SDCs can be added to the program at these points. In addition, during the execution of the instructions of this assertion, the entire memory is injected and in particular also the memory, which is not proteced by the assertion. A short code example to illustrate this is shown below:

[assert1, assert2, assert3, assert4, assert5]	SDCs
[0, 0, 0, 0, 1]	667 792
[1, 0, 0, 0, 1]	688 507
[0, 0, 0, 1, 1]	701 957
[0, 1, 0, 0, 1]	697 033
[0, 0, 1, 0, 1]	703 802
[1, 0, 0, 1, 1]	704 466
[1, 1, 0, 0, 1]	707 584
[1, 0, 1, 0, 1]	715 638
[0, 1, 0, 1, 1]	726 452
[0, 0, 1, 1, 1]	720 954
[0, 1, 1, 0, 1]	722 079
[1, 1, 0, 1, 1]	730 459
[1, 0, 1, 1, 1]	735 990
[1, 1, 1, 0, 1]	741 354
[0, 1, 1, 1, 1]	744 953
[1, 1, 1, 1, 1]	755 252
[0, 0, 0, 1, 0]	1 568 798
[0, 0, 0, 0, 0]	1 539 162
[1, 0, 0, 1, 0]	1 592 419
[1, 0, 0, 0, 0]	1 606 692
[0, 1, 0, 1, 0]	1 613 835
[0, 1, 0, 0, 0]	1 606 239
[0, 0, 1, 1, 0]	1 626 834
[1, 1, 0, 1, 0]	1 650 336
[1, 1, 0, 0, 0]	1 639 217
[0, 0, 1, 0, 0]	1 638 825
[1, 0, 1, 1, 0]	1 647 153
[0, 1, 1, 1, 0]	1 670 647
[1, 0, 1, 0, 0]	1 679 553
[0, 1, 1, 0, 0]	1 672 429
[1, 1, 1, 1, 0]	1 702 363
[1, 1, 1, 0, 0]	1 714 920

Table 3.4.: Table with the occurrences of SDCs of a quicksort algorithm with five assertions and correspondingly 32 possible configurations.

```
bool a = true;
bool b = false;
bool c = true;
someWorkload();                    // executes some unrelated workload
assert(a);
```

In this example, the variable a is checked by an assertion after a workload is executed that does not read or write memory as to not interfere with the rest of the program. In addition, there are also variables b and c that are not protected by an assertion. If faults are injected into variable a, these are detected by the assertion and at this point the assertion reduces the number of SDCs. At the same time, no faults are detected that are injected into variable b or c. When the assertion is executed, it then increases the attack surface for b and c during the execution time of the assertion. In addition, SDCs are caused when injecting into the memory area of a when executing instructions of the assertion that are located after a has been checked. In these cases, additional SDCs are added by assertion at this point.

Figure 3.1 provides an additional visualization of this example. Here, the SDCs detected by the assertion are shown in solid green and the additional SDCs that would not have occurred without the assertion are shown in solid red. It can be seen that the benefit of an assertion is also related to the size of the memory used for the injection. The larger the memory, the less the benefit of an assertion can be, since the additional runtime increases the attack surface in the entire memory area used.

When an assertion is considered in isolation, it is now possible to decide whether the assertion increases or decreases the total number of SDCs by comparing the area of detected SDCs with that of the additional SDCs of the assertion. If the number of detected SDCs is larger, the assertion decreases the number of SDCs and should be activated. Accordingly, the assertion should be disabled if the number of detected SDCs is smaller. In the following sections, further reasons will be explained that explain why assertions have different cost-benefit ratios and how they are connected to each other. In particular, it will be discussed whether assertions can be considered in isolation, as assumed in the previous section.

3.3. Prior Work

The problem investigated in section 3.2, that an optimal configuration does not simply contain as many active assertions as possible, is already explained in the paper by Lenz and Schirmeier [10]. It recognizes that while an assertion detects faults in memory areas protected by it, at the time of execution other areas of memory become more vulnerable as the program becomes a larger runtime. In the

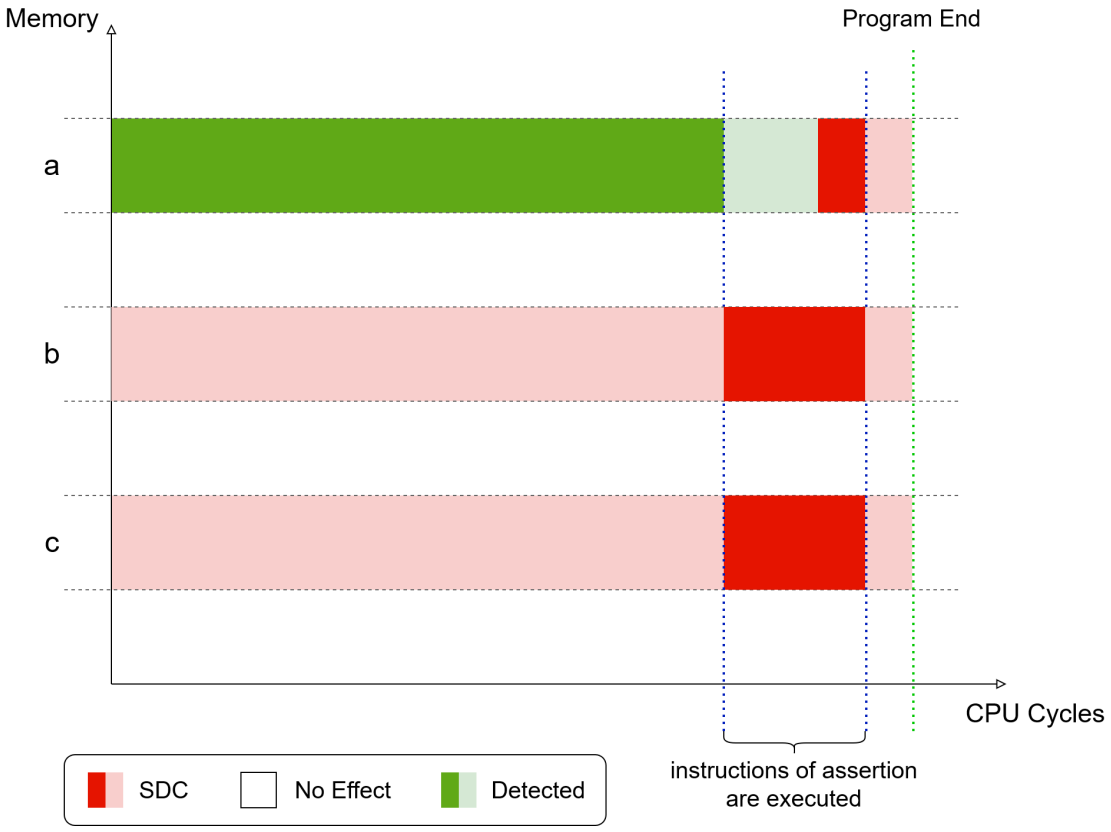

Figure 3.1.: Representation of a fault space for variable a, b and c with one assertion. On the y-axis the memory bits of the variables are plotted and on the x-axis the CPU cycles executed until then. The results for a fault injection are plotted in color for each point. Transparent red represents the occurrence of SDCs in a program that would appear independent of the assertion and solid red the occurrence of additional SDCs in a program if the assertion is included. In solid green the SDCs are marked, which are detected by the assertion and reduce the total number of SDCs. Detected SDCs are marked in transparent green, which are also detected by the assertion, but would not have occurred without the assertion and therefore do not change the total number of SDCs.

paper this is called attack-surface tradeoff. It also states that for a program with N assertions, there are 2^N possible configurations. This quickly leads to the problem that not all configurations can be executed individually, since for example for 100 assertions $2^{100} \approx 10^{30}$ configurations would have to be tried in order to find the optimal configuration. In the conclusion of the paper it is therefore proposed to use optimization methods that are able to find the optimal configuration with the least possible effort. Genetic algorithms or integer-linear programming are suggested as possibilities. This thesis will continue the work of Lenz and Schirmeier and investigate the optimization methods proposed in the paper. It will also be analyzed to what extent the complexity of the optimization issue, prior to the application of an optimization procedure can be reduced. In a preliminary work of Lenz and Schirmeier it was found that even with an optimization method like integer-linear programming, finding an optimal configuration for many assertions can take a lot of time. In order to develop a method that reduces the complexity in advance, the following sections analyze the dependencies between assertions with respect to fault tolerance in more detail. The goal is to be able to include the knowledge about these dependencies already in the optimization procedure and thus to reduce the complexity of the optimization problem.

3.4. Relation between Assertions

In section 3.2, it is experimentally demonstrated that more assertions do not automatically lead to fewer SDCs in a fault injection campaign. Any configuration of assertions could potentially be the optimum when the programs are considered without prior knowledge about them. It is also shown that one of the reasons for this may be the additional instructions required for the assertion, which may result in additional SDCs that would not have appeared without the assertion. In this section, the dependencies between assertions with respect to the avoided and additional SDCs will also be addressed.

3.4.1. Dependency through Redundancy

Firstly, we will discuss the dependency of assertions that occurs when two assertions protect the same memory area. This dependency is called dependency through redundancy in this thesis. The following code can be given as an example:

```
bool a = true;
bool b = true;
someWorkload();                 // executes some unrelated workload
assert1(a && b);
someWorkload();                 // executes some unrelated workload
assert2(a);
```

In this code, variable a and b exist, where injected faults in variable a and b are detected by `assert1`. Additionally injected faults in variable a are detected by `assert2`. To create execution time before assertions, a workload is again created that does not read or write memory. Figure 3.2 visualizes this example in a faultspace. It can be seen that there is an overlap for the SDCs detected by the assertions. In this case, `assert2` detects SDCs injected into variable a that are also detected by `assert1`. Thus, there exists some redundancy in the detection of assertions. If it is assumed that `assert1` would only protect the memory area of variable a, this assertion would clearly be redundant, since all SDCs in a are also detected by `assert2`. `assert1` would in this case only add additional SDCs by its own execution and would have to be disabled in an optimal configuration. `assert1`, however, also protects the memory of variable b in this example. This leads to the consequence that this assertion can potentially reduce the number of SDCs. It can be concluded that mutually redundant assertions must be considered together, since they partially detect the same SDCs and therefore it is no longer correct to consider the cost-benefit ratio individually. In addition, a solution how to choose the optimal configuration for partially redundant assertions is not directly obvious, as shown in the example. The benefit of an assertion outside the redundantly monitored SDCs can justify the activation in an optimal configuration.

3.4.2. Dependency through Time Overlap

After showing in the previous section that assertions can be in interdependency if they protect the same or partly the same memory area and detect the same SDCs in it, in this section it will be considered to what extent assertions can be dependent on each other even if they do not overlap in the protected memory area. The dependency occurs through temporal overlap of the executed instructions of an assertion and the increased attack surface in memory areas that are protected by other assertions. This dependency is called dependency through time overlap in this thesis. The following code example can be given as an example of such a dependency:

```
bool a = true;
bool b = true;
someWorkload();                 // executes some unrelated workload
assert1(a);
someWorkload();                 // executes some unrelated workload
assert2(b);
```

Variable a is protected by assertion `assert1` and variable b is protected by assertion `assert2`. Since no redundancy is included here, i.e. the assertions do not protect overlapping memory areas, it could be assumed that these assertions could

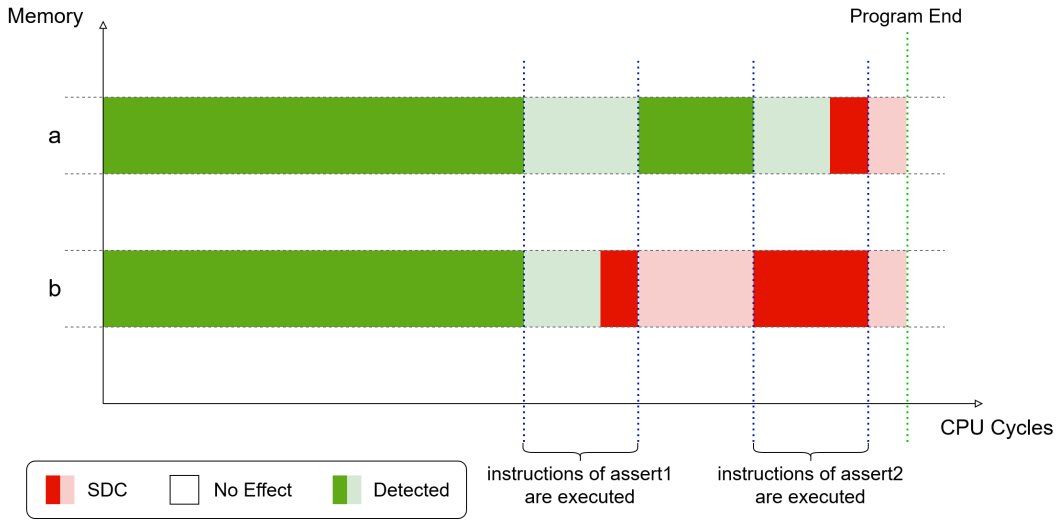

Figure 3.2.: Representation of a fault space for variable a and b with assertions `assert1` and `assert2`. Again the y-axis the memory bits of the variables are plotted and on the x-axis the CPU cycles executed until then. The results for a fault injection are plotted in color for each point. Transparent red represents the occurrence of SDCs in a program that would appear independent of the assertion and solid red the occurrence of additional SDCs in a program if the assertion is included. In solid green the SDCs are marked, which are detected by the assertion and reduce the total number of SDCs. Detected SDCs are marked in transparent green, which are also detected by the assertion, but would not have occurred without the assertion and therefore do not change the total number of SDCs.

be considered independently. Figure 3.3 visualizes the example in faultspace. It can be seen that executing `assert1` would add additional SDCs in the memory space protected by `assert2`. This results in the problem that these additional SDCs are detected if `assert2` is activated and accordingly they are not considered negatively in the cost-benefit analysis of `assert1`. However, if `assert2` is disabled, then these additional SDCs must be considered. So whether `assert1` is an assertion that must be active in the optimal configuration depends on whether `assert2` is active or not. At the same time, it is not necessarily possible to make a general decision for `assert2` whether it should be enabled, since it may be related to other assertions or may also increase the attack surface in the memory space of `assert1` with its own instructions. Accordingly, assertions must also be considered in this temporal context and must be considered together for finding the optimal configuration.

3.5. Conclusion

This chapter experimentaly shows, using a quicksort algorithm, that for an optimal configuration, there are not necessarily as many active assertions as possible. Subsequently, the causes are discussed. Central to this is that an active assertion adds instructions to the program, which increases the attack surface of the program over the entire considered memory space. This could cause an assertion to lead to more additional SDCs than it can detect.

Furthermore, assertions also depend on each other when it comes to finding an optimal configuration. The dependency through redundancy is of importance where assertions partially protect the same memory area and can therefore have a lower benefit in conjunction with the other assertion. In addition, dependency through time overlap must also be considered. Here, the instructions of one assertion increase the attack surface in a protected memory area of another assertion. It depends whether this other assertion is active and therefore the increased attack surface does not play a role, since SDCs are detected in this area or whether it is inactive and the additional SDCs from this memory area must be included accordingly.

Assertions must be considered in context of fault tolerance under the aspects just mentioned. This makes the search for the optimal configuration more complex, since it is not possible to decide for each assertion in isolation whether it should be part of an optimal configuration as an active assertion. In the following chapter it is to be examined, how this can be considered and still an optimal configuration can be found efficiently.

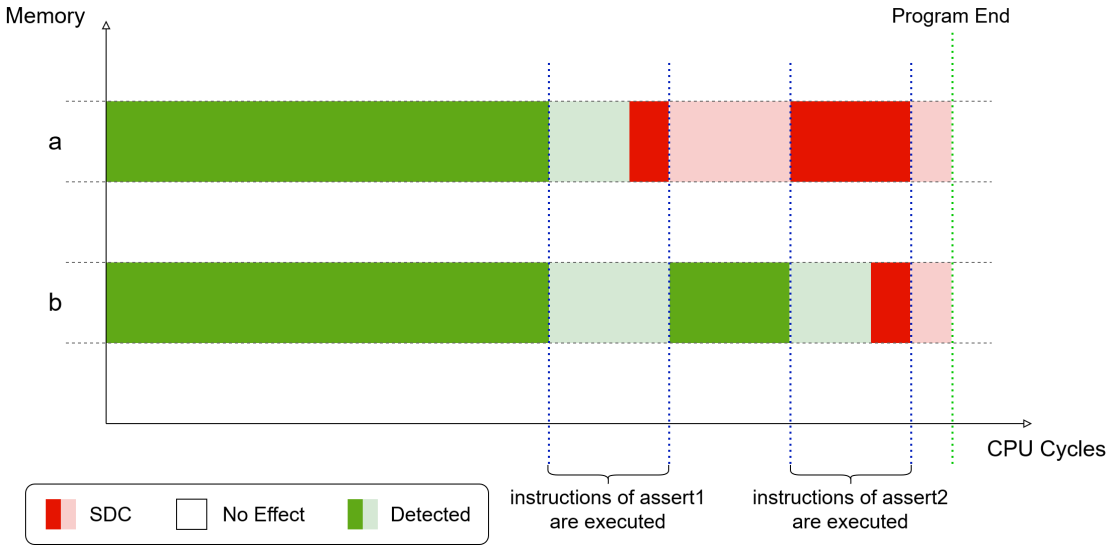

Figure 3.3.: Representation of a fault space for variable a and b with assertions `assert1` and `assert2`. Again the y-axis the memory bits of the variables are plotted and on the x-axis the CPU cycles executed until then. The results for a fault injection are plotted in color for each point. Transparent red represents the occurrence of SDCs in a program that would appear indipendant of the assertion and solid red the occurrence of additional SDCs in a program if the assertion is included. In solid green the SDCs are marked, which are detected by the assertion and reduce the total number of SDCs. Detected SDCs are marked in transparent green, which are also detected by the assertion, but would not have occurred without the assertion and therefore do not change the total number of SDCs.

4. Proposed Solutions

This chapter proposes possible solutions that could determine the most optimal configuration of assertions. Section 4.1 first explains how, with respect to the number of SDCs, further configurations can be calculated from a known configuration of assertion. Then in section 4.2 possible solutions for finding the optimal configuration of assertions are discussed. Finally, in section 4.3, the reduction of the complexity of the optimization problem is investigated.

4.1. Calculating Configurations

In chapter 3 it is shown that a configuration of assertions that minimizes the number of SDCs does not only consist of as many active assertions as possible. Rather, it is an optimization problem to find such an optimal configuration, since the assertions are interdependent.

One way to find this optimal configuration is to execute and record all possible configurations. However, the execution of a configuration can take several hours, which for a program with 20 assertions, i.e. $2^{20} \approx 1,000,000$ configurations, would already take several decades. Therefore, it is desirable that the configurations do not actually require to be executed.

The possibility to calculate the number of SDCs that occur for a program for a given configuration of assertions is provided by the work of Lenz and Schirmeier [10]. In this paper the tool DETOx is presented. It is able to compute the number of SDCs for an arbitrary configuration from a recording of a fault injection campaign in which all assertions are active. Thereby, as described in section 2.3, a fault injection campaign consists of the uncorrupted execution of the program with the tracing of all memory accesses and the ground truth of the output (golden run), then the reduction of the fault space that is injected (faultspace pruning) and finally the injection into every single bit of the fault space before every CPU cycle (fault injection). In DETOx the last step fault injection is extended, so that after each injection it is also recorded which assertions in the subsequent program flow were triggered by the injection, while the program is not terminated by the assertion. In addition, the result type (OK, SDC, TIMEOUT, TRAP) is recorded. In figure 4.1 this recording is graphically illustrated. Here, for each bit before each CPU cycle, it is recorded which assertions are triggered by an injection at that point. For example, an injection into bit_1 before CPU cycle 3 will trigger assertion

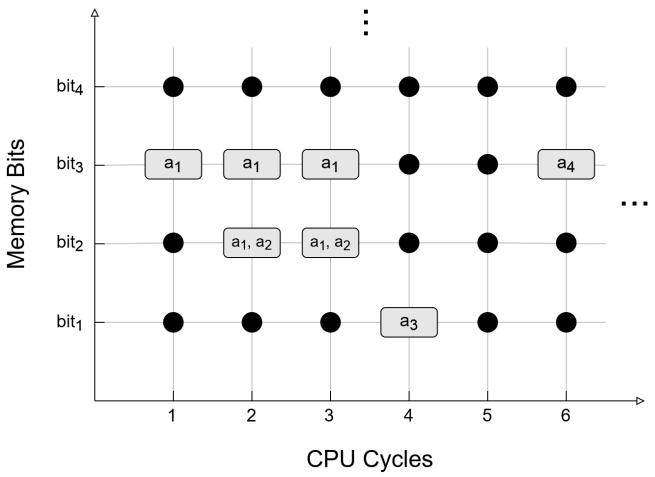

Figure 4.1.: Representation of an fault space for an example program with assertions triggered by an injection. The memory bits are plotted on the y-axis and the CPU cycles executed up to that point are plotted on the x-axis. Each point represents a possible injected fault. Each box represents a possible injected fault with the assertions triggered by that point.

a_1 and a_2 during further program execution.

According to these records, it can be determined which memory areas are covered by an assertion at which CPU cycles. Using this information, any configuration of assertions can be calculated for the program. For these configurations it can be determined, how large the number of the respective result types (OK, SDC, TIMEOUT, TRAP) would be, if these were applied to the corresponding program and if it would be executed. In this thesis, such calculated configurations are called synthetic configurations. In the following these are defined:

Definition 4.1.1 (Synthetic Configuration)
A synthetic configuration describes a configuration that is not actually executed for the corresponding program, but for which the expected result types (OK, SDC, TIMEOUT, TRAP) can be calculated.

This calculation will be explained using figure 3.1 from section 3.2. DETOx is able to determine how many SDCs are detected by an assertion, which is illustrated by the area in solid green in the figure. This information comes from recording which assertions are triggered by which injection. In addition, DETOx can detect the additional SDCs added by the assertion execution, illustrated in solid red in the figure. For this, the dynamic instructions of an assertion are determined and all injections that lead to an SDC in this interval over the whole used memory

are aggregated. For the calculation of the number of SDCs of a configuration, it is also necessary to know the original total number of SDCs, which is determined during the run with all assertions in the active state. When an assertion a is then supposed to be inactive from the original recorded configuration where all assertions are active, all occurrences of a in the record has to be removed. That means for every single injection that includes a, it has to be deleted for this injection. Therefore it appears that a never existed in the record. After that the number of SDCs $\mathrm{SDC}_{\mathrm{without}_a}$ has to be recalculated from the record. Additionally the dynamic instructions d_a of the assertion itself has to be removed. This is done by the following formula:

$$\mathrm{SDC}_{\mathrm{new}} = \mathrm{SDC}_{\mathrm{without}_a} - d_a \tag{4.1}$$

Now exactly assertion a is inactive in the synthetic configuration and would be producing $\mathrm{SDC}_{\mathrm{new}}$ SDCs. If another assertion should be inactive, the same procedure can be used as often as desired for every assertion in a series.

4.2. Optimizing Configurations

In this section, possible optimization methods are presented. For this we first consider the possibility of randomized selection of assertions in section 4.2.1. Subsequently, in section 4.2.2 and 4.2.3 possible optimization methods are examined, which could be suitable for finding an optimal configuration of assertions.

4.2.1. Randomization

In the previous section it is shown that with higher number of assertions not all configurations can be executed individually. This problem can be mitigated by calculating configurations. However it would not be possible to calculate all configurations for example over 2^{100} Assertions, as available in FreeRTOS. therefore a solution must be found, where not each configuration has to be considered individually.

A variant is the omitting of computations. Only randomly selected configurations in a manageable number are computed. However, with this method the search space is probably insufficiently searched, which does not lead to the desired optimum. This can be illustrated with the consideration that 2^{16} computations can still be feasible given that they take about eight days on modern hardware [1]. However, since the total search space can be, for example, 2^{100} configurations in size, only $5.26 \cdot 10^{-28}\%$ would be covered. Even with significant improvements in

[1] From experiments it is known that the time to calculate one configuration for example with around 100 assertions needs around 10 seconds. Therefore the time needed to calculate 2^{16} configurations is $2^{16} \cdot 10s = 655\,360s$ which is about eight days.

the efficiency of the calculation, the coverage would remain extremely low. The probability that the optimal configuration, or a close one, is among those randomly selected is low, unless special unjustifiable assumptions can be made about the search space.

Therefore, although randomization is used in the evaluation in chapter 6.2, it is generally of limited use to reliably find a configuration close to the optimum or even an optimal configuration.

4.2.2. Evolutionary Algorithms

One possibility to find the optimal configuration of assertions could be the use of an evolutionary algorithm [4]. The aim is to imitate the functioning of natural evolution. Such an algorithm is in principle able to find local optima in an arbitrary search space, which would accordingly also be applicable to the present problem of finding an optimal configuration of assertions.

The basic procedure for finding an optimal configuration of assertions could be to first generate a population of random configurations. This population represents the first generation of the algorithm. Then, each configuration of the population would be evaluated by calculating the corresponding number of SDCs. The number of SDCs is the fitness value of a configuration. Based on this fitness value, the best configurations of a population are selected and are recombined in the subsequent step, for example, by randomly exchanging the individual states of the assertions between the configurations. Subsequently, the configurations resulting from the recombination are mutated. In this process, individual states of the assertions in a configuration are switched randomly to active or inactive. The resulting new population is then the second generation. Now the configurations are evaluated again and the process starts from the beginning. The process ends when a termination criterion is reached. This can be, for example, the passing of a certain improvement of the fitness value between two successive generations. This can be a sign that the algorithm is approaching a local optimum, i.e. that it has found a configuration that returns a number of SDCs close to a local optimum.

With an evolutionary algorithm, it is possible to find configurations that are close to local minima with respect to number of SDCs, or if the parameters of the algorithm are chosen correctly, even the global minimum. However, the determination of these parameters is laborious and would require an experimental search of them. In addition, the parameters would then only be safely suitable for the program for which they were optimized. In addition, convergence to a local minimum is typically slow and further complicates the determination of the parameters.

Accordingly, it would be desirable if already known properties could be exploited by determining the number of SDCs for a configuration. Since an evolutionary algorithm does not know the characteristics of the search space, it is not well suited

if a solution should be found in a short time. In the following sections it will be shown how the calculation process of the number of SDCs for a configuration can be used to determine an optimal configuration.

4.2.3. Integer Linear Programming

Since the solution idea presented so far in the previous section cannot quickly and reliably lead to the optimal configuration, this section investigates finding this optimal configuration using Integer Linear Programming (ILP). It will be briefly explained how this method works. The following is based on the book "Linear and nonlinear programming" from Luenberger et al. [12] and the preliminary work of Lenz and Schirmeier.

The goal of Integer Linear Programming is to minimize a cost function that is linear in its unknowns under linear equality and inequality constraints. In addition, all variables must be integers. Formally, a standard form of an Integer Linear Program can be given:

$$\begin{aligned} \text{minimize } & \sum_{i=1}^{n} c_i x_i \\ \text{subject to } & \sum_{i=1}^{n} a_{1i} x_i = b_1 \\ & \vdots \\ & \sum_{i=1}^{n} a_{mi} x_n = b_m \\ \text{and } & x_1 \geq 0, ..., x_n \geq 0. \end{aligned}$$

Here $\mathbf{b} \in \mathbb{R}^m$ is an m-dimensional vector and $\mathbf{c} \in \mathbb{R}^n$ is an n-dimensional vector of real numbers. $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a $m \times n$ matrix. The values in $\mathbf{b}$, $\mathbf{c}$ and $\mathbf{A}$ are constants and part of the description of the given problem. The vector $\mathbf{x} \in \mathbb{N}^n$ contains the variables to be determined in the optimization of the Integer Linear Program. The variables in $\mathbf{x}$ can only take positive integer values. To determine a solution for the ILP, different classes of algorithms can be used, such as the cutting plane method or the branch and cut method. Since an ILP is NP-hard, heuristic methods must also be used to some extent. Since state-of-the-art ILP solvers can choose from many of these methods, depending on the characteristics of the ILP, this thesis will not discuss specific solution methods in detail.

To create an Integer Linear Program for finding the optimal configuration, known properties of the optimization problem from section 4.1 can be used. It is explained how the number of SDCs of a configuration can be calculated. Necessary for this is the knowledge about how many SDCs d_a are detected by an assertion a. Furthermore, it is also known how many SDCs o_a additionally occur

by the execution of the assertion. The difference s of detected and additional SDCs then describes the total reduced number of SDCs, if the assertion is considered in isolation:

$$s_a = d_a - o_a \tag{4.2}$$

Since an assertion a can be either active or inactive, it is $a \in \{0, 1\}$, where 0 represents an inactive assertion and 1 represents an active one. If an assertion a is active, then s_a shall be included in the equation to be optimized, and if this assertion is inactive, then s_a shall not be included. This results in the following term:

$$s_a \cdot a \tag{4.3}$$

However, it must also be noted that assertions cannot be considered independently of each other. If several assertions occur, the redundantly detected SDCs r_{A_r} must be added, if one the assertions $A_r = \{a_l, ..., a_k\}$ are active. To achieve the selection when more than on assertion is active. The following term can be used:

$$(1 - \prod_{a \in A_r} (1 - a)) \tag{4.4}$$

This results in the following term for the selection of the redundant SDCs:

$$r_{A_r} \cdot (1 - \prod_{a \in A_r} (1 - a)) \tag{4.5}$$

The combining of the terms 4.3 and 4.5 with the extension to any number of assertions $A = \{a_1, ..., a_n\}$ is shown in the following, where M contains the set of all redundant occurrences of assertions:

$$\sum_{a \in A} a \cdot s_a + \sum_{A_r \in M} r_{A_r} \cdot (1 - \prod_{a \in A_r} (1 - a)) \tag{4.6}$$

The resulting term describes the total number of reduced SDCs depending on the state of the assertions in A. This term is the cost function of the proposed ILP, which should be maximized for the highest possible number of reduced SDCs subject to the variables in A. From these variables, the optimal configuration $[a_1, ..., a_n]$ is obtained after optimization.

The constraints are defined in a first approach as follows:

$$\begin{aligned} a_1 &\leq 1 \\ &\vdots \\ a_n &\leq 1 \end{aligned}$$

The idea is to take into account that an assertion a can only be active or inactive and $a \in \{0, 1\}$ applies. However, it is noticeable that the cost function 4.6 is not

linear in its variables because of term 4.5. Products of variables can be present. Since only binary decision variables occur in this problem, the problem can still be linearized with cleverly chosen linearization rules. For each product $\prod_{i=k}^{l} a_i$ a new variable z_j is introduced, which is representing this product. For each variable z_j the following linearization rules must then be added [6]:

$$z_j \leq a_i \text{ for } i = k, ..., l$$
$$z_j \geq (\sum_{i=k}^{l} a_i) - (l - 1)$$

These linearization rules cause the representation of the product z_j to be 1 only if all factors $a_k, ...a_l$ are also 1, otherwise z_j is 0. This matches the desired behavior.

An additional optimization target should be the number of activated assertions. The fewer assertions a program contains, the faster it can be executed. Therefore, the number of active assertions in a configuration should be minimized. For this, the cost function from 4.6 must be adapted. Each variable can be subtracted from the cost function, so that many active assertions would make the cost function smaller compared to many inactive assertions. However, this makes the cost function ambiguous. Therefore, it has to be multiplied by the number of all variables beforehand so that there are enough non-ambiguous places for the added variables. The resulting cost function is shown in the following:

$$\text{maximize } n \cdot (\sum_{a \in A} a \cdot s_a + \sum_{A_r \in M} r_{A_r} \cdot (1 - \prod_{a \in A_r} (1 - a))) - \sum_{a \in A} a \tag{4.7}$$

It can be shown that to determine an optimal configuration, an ILP can be designed. Accordingly, since an ILP can be solved exactly, the optimal configuration can be found reliably if solving the ILP is done in a reasonable amount of time. Furthermore, it is not entirely certain whether the calculation method from section 4.1 used as a basis for the ILP leads to a robust result when those are compared with real configurations that have been executed in reality. These questions will be examined in more detail in the evaluation in chapter 6.

4.3. Complexity Reduction

In this chapter, it will be investigated how the optimization problem can be subdivided into smaller subproblems that can presumably be solved in a shorter time. These subproblems could then be solved one after the other, for example by Integer Linear Programming, and combined to an overall optimum. In section 4.3.1 it will be considered how partitions of different assertions can be formed, which allow to optimize only a selection of assertions. It has to be ensured that the optimum found by this method does not differ significantly from the original one, where all assertions are included in the optimization at the same time.

4.3.1. Partition Assertions

In this section it is described how the assertions occurring in a program can be partitioned so that the optimization problem can be simplified. In section 4.3.1.1 the properties of a partition are defined and assumptions are made to justify that such a partitioning can be legitimate. Afterwards, in section 4.3.1.2, 4.3.1.3 and 4.3.1.4 examines how to handle dependencies between assertions in order to partition them correctly. In section 4.3.1.5, heuristics are proposed to enable further partitioning. Finally, in section 4.3.1.6 the collected findings are summarized.

4.3.1.1. Definitions and Assumptions

In the following a partition is defined:

Definition 4.3.1 (Partition)
A partition contains different disjoint assertions which have a relation to each other with respect to the fault tolerance. A partition is of the form $\{a_k, ..., a_l\}$ *with the assertions* $a_k, ..., a_l$.
In another representation of a partition, it also includes the occurrences of the corresponding assertions of the partition. The number of occurrences corresponds to the number of injections that trigger the corresponding assertions. Such a partition is of the form $\{a_k \times n_{a_k}, ..., a_l \times n_{a_l}\}$ *with the assertions* $a_k, ..., a_l$ *and the respective occurrences of these assertions in the partition* $n_{a_k}, ..., n_{a_l}$.

The idea in partitioning is to reduce the complexity of the optimization problem. This exploits the fact that $2^n > 2^k + ... + 2^l$ for $k + ... + l = n$ and $k \geq 1, ..., l \geq 1$ holds. Which means that the number of configurations 2^n for n assertions is always greater than the summed number of all configurations of the respective partitions. Accordingly, fewer configurations would have to be evaluated.

The goal here is to reduce the computational effort required to find an optimal configuration that minimizes the number of SDCs in a fault injection campaign.

- In complex systems, there are partitions of assertions that are independent of each other.
- These independent partitions can be considered separately in respect to susceptibility to faults.

These assumptions are considered and theoretically justified in the following chapters.

4.3.1.2. Transitivity

As described in section 4.1, the fault-injection step records for each injected bit flip which assertions are triggered by it. Thus, assertions $\{a_k, ..., a_l\}$ can be triggered by one injection. For the following program section `assert1` is a_1, `assert2` is a_2 and `assert3` is a_3:

```
bool a = 0;
assert1(a == 0);
bool b = 0;
assert2(a == 0 && b == 0);
assert3(b == 0);
```

It was experimentally examined that the assertions $A_1 = \{a_1, a_2\}$ and in the same program, assertions $A_2 = \{a_2, a_3\}$ are triggered by an injection. Thus, sets of assertions can overlap, while the union of these sets $A_{1\cup2} = \{a_1, a_2, a_3\}$ in this program excerpt is not triggered by a single injection.

The question arises whether, for the purpose of partitioning, the assertions of the set A_1 can be considered independently of the assertions of the set A_2, or whether the assertions may only be seen together in the union $A_{1\cup2}$. The following thoughts can be made in this regard: It is first assumed that the sets A_1 and A_2 can be considered independently. That means the program contains in one run only the assertions from A_1 and in another run only the assertions from A_2. On each of the program runs, the number of SDCs would be minimized and the best configuration determined. After the independent optimization, the optimal configuration with respect to all assertions a_1, a_2, a_3 of the program would then be determined. In this example, it is then assumed that for A_1 the optimal configuration would be $[0, 1]$, so only assertion a_2 would be active and contained in the program section for a minimum number of SDCs. For A_2, the optimal configuration would be $[0, 1]$, so only assertion a_3 would be active. This creates the contradiction that the optimization on A_1 activates assertion a_2, but according to the optimization on A_2 it should be deactivated. This contradiction could also not be resolved easily, since under the assumption that assertion a_2 should be activated, the optimization on A_2 would also have to be considered under this aspect. It follows that the assertions in A_1 and A_2 must not be viewed independently, but the unified set of A_1 and A_2 has to be considered. The sets must be unified if they overlap. Therefore, sets of assertions have a transitive relation to each other. Thus, for a correct optimization, the sets of assertions must be transitively joined. For the example above, the assertions a_1, a_2, a_3 must be considered together in one optimization process.

4.3.1.3. Independent Assertions Regarding Redundancy

In section 3.4.1 it is found that assertions must not be considered independent if they partially or completely protect the same memory area. If assertions are to be partitioned, this dependency must be taken into account. The goal are partitions which can be considered independently with respect to redundancy. Only within these partitions assertions should protect the same memory areas. In this section we will discuss how this dependency can be extracted from the recorded data from section 4.2.

In order to determine which assertions protect the same memory areas, it is possible to take advantage of the fact that in the recording as described in section 4.2, triggering an assertion does not cause the program to terminate. The program will continue to run and it will only be recorded that the corresponding assertion was triggered on that injection. If further assertions are triggered after this assertion was triggered, this is also recorded for this injection. Thus, each assertion triggered by an injection is recorded. Since these assertions are all triggered by one injection, i.e. the injection into a memory bit before a CPU cycle, these assertions must all protect this memory bit and thus overlap in it. In the record, all the information necessary to determine all redundant dependencies are in this form and can be read out piece by piece from the Injections which trigger more than one assertion.

4.3.1.4. Independent Assertions Regarding Time Overlap

In section 3.4.2 it is recognized that assertions are not only dependent of each other if they protect the same memory areas, but are also dependent if at a certain point in time these assertions are triggered from injections in different memory areas. This dependency is also called dependency through time overlap in this thesis. The reason for this is that the instructions of an assertion increase the attack surface also in other memory areas related to other assertions. It then depends on whether these other assertions are active, so that the additional SDCs, caused by the increased attack surface, are detected by these assertions and do not affect the fault tolerance negatively.

This dependency through time overlap is to be determined from the recorded assertions from section 4.2. To do this, the injections and the triggered assertions recorded for them are grouped by their dynamic instructions. The dynamic instructions are the instructions executed by the CPU. Thus, all assertions that are triggered in one of the respective dynamic instructions are selected as related in time. Assertions that are triggered by injections that occur at the same time are related.

Independent assertions regarding time overlap are assertions that are not triggered by injections at the same time. If partitions are constructed with them, only

within a partition the assertions should fulfill the dependency through time overlap between each other.

4.3.1.5. Further Heuristics

In addition to the exact partitioning of the previous sections, heuristics will be presented, that have the potential to further subdivide the assertions into smaller partitions while without ignoring of the connection between the assertions too much. In other words, the goal must be to get as close as possible to the optimal configuration and at the same time reduce the computation time needed to find this optimum. Since these are heuristics, it cannot be directly proven that this goal is met by every application to every program. It is dealt in chapter 6 in the evaluation exemplary on programs and operating systems, for which the following heuristics are used. It will be considered how well these heuristics reduce the computation time and how far the result is from the minimum number of SDCs. In the following the heuristics are explained.

. **Ignore assertions with small influence** In this heuristic, assertions with little influence should be ignored, since their presence or absence probably contributes insignificantly to finding the minimum number of SDCs.

To determine the influence of an assertion on this optimum, the difference between minimum detected and maximum added SDCs could be used. The smaller the difference, the smaller could be the influence, since this difference is essentially the benefit of an assertion, which influences the determination of the minimal number of SDCs in the ILP. In addition, the number of injections by which an assertion is triggered must also be considered. If an assertion is triggered by injections a large enough number of times, then even a small difference can have an significant impact. To determine the influence i of an assertion, the product of the two factors difference between minimum detected and maximum added SDCs d and the number of injections n triggering this assertion is calculated as follows:

$$i = d * n \tag{4.8}$$

Furthermore, it must be judged whether the influence of an assertion is small enough not to be considered. For this it is probably useful to relate this influence to the one of the other assertions. For example, all assertions could be sorted according to their influence and then the assertions with the smallest influence that together have a certain small percentage of the total influence of all assertions could be removed. For example, assertions with a combined influence of 5% of the total influence could be removed.

. **Combine assertions with large influence** In this heuristic, occurrences of an assertion are to be combined with other occurrences of the same assertion,

which has a significantly greater influence. Assertions from partitions containing more than one assertion may be detached from the partition if the influence of the same assertion is significantly higher for all injections where only this assertion is triggered. The extracted occurrences of the assertion are then combined with the individual occurrences of the assertion. The idea here is that while an assertion may be related to other assertions, individually it may have a significantly greater impact that the relation plays only a minor role. Individually means in this context that exactly one assertion is triggered by one injection. The number of this single occurrences are then the number of injections that trigger only this assertion. Only merging with single occurrences are considered for this heuristic and not also occurrences in other partitions with more than one assertion, since an assertion in another partition would otherwise be assigned an occurrence that is too high compared to the other assertions of a partition. Since each partition leads to an individual optimal configuration, it does not seem logical to mix partitions containing such relations.

The influence is calculated in the same way as in equation 4.8. However, the difference between minimum detected and maximum generated SDCs is identical in this case. Accordingly, it is sufficient to compare the number of occurrences of an assertion directly, since the ratio between them is interesting. It is suggested to detach an assertion from a partition if the proportion of single occurrences of an assertion exceeds a threshold of all occurrences of this assertion in all partitions. Such a threshold could be for example 95%. That means, if the single occurrences contain 95% of all occurrences of this assertion over all partitions, this assertion is removed from all partitions and merged with the single occurrences. As an example, this procedure can be shown with the following partitions, where P represents an arbitrary partition and S_{a_1} represents the single occurrences for assertion a_1 and S_{a_2} for a_2 and S_{a_3} for a_3, respectively:

$$\begin{aligned} P_1 &= \{a_1 \times 1000, a_2 \times 20, a_3 \times 20\} \\ S_{a_1} &= a_1 \times 990 \\ S_{a_2} &= a_2 \times 10 \\ S_{a_3} &= a_3 \times 10 \end{aligned} \tag{4.9}$$

In this case, a_1 has 990 single occurrences versus a total of 1000 occurrences. Thus, the single occurrences have a 99% share of all occurrences and according to a threshold of 95%, a_1 would be separated from P_1 and then forms its own partition. The single occurrences for a_2 and a_3, respectively, account for only 50% and accordingly none of these assertions would be detached. The result after this

step is shown below:

$$\begin{aligned} P_1 &= \{a_2 \times 20, a_3 \times 20\} \\ P_2 &= \{a_1 \times 1000\} \\ S_{a_1} &= a_1 \times 990 \\ S_{a_2} &= a_2 \times 10 \\ S_{a_3} &= a_3 \times 10 \end{aligned} \tag{4.10}$$

In the optimization step, only this assertion would then be evaluated individually as an own partition.

4.3.1.6. Conclusion

In this chapter methods are presented, which should make it possible to find the optimal configuration without having to execute all possible combinations, since this is practically not possible for programs with 100 assertions and more. It is explained first, how the number of SDCs for an arbitrary configuration can be computed. For this computation it is necessary to execute one fault injection campaign with all assertions active. This allows a faster determination of the number of SDCs for a configuration. However, even with this method it is hardly feasible to calculate the number of SDCs of all possible configurations, since this is also not practical for a program with, for example, 2^{100} possible configurations.

Therefore Methods are investigated that are intended to find or approximate the optimal configuration, without considering all possible configurations. Randomization is proposed first, but it browses the search space non specifically and without prior knowledge of the problem, and is likely to yield insufficient and uncertain results. Then, the possibility of designing an evolutionary algorithm for the problem is discussed. This variant has the advantage that the search space for an optimal configuration can be arbitrary, but at the same time an evolutionary algorithm converges slowly to a local optimum and needs tuning of its parameters. It would be a slow and inflexible method since it would have to be applied to each program separately to find the specific optimal configuration. Subsequently, the use of Integer Linear Programming is suggested. It can be exploited that the number of SDCs of a configuration can be calculated. By using an ILP, an optimal configuration can be found if the ILP can be solved in acceptable time. However, even this method becomes slower as the number of assertions increases. For example, the Linux kernel has at least 8000 assertions. With this amount, the ILP could take an impractically long time. Therefore, it is suggested to partition the assertions and utilize the dependencies between the assertions. The goal are partitions that are as independent of each other as possible and can be optimized individually. This should reduce the complexity of the optimization problem. The extent to which partitioning is suitable for this is evaluated in chapter 6.

5. Realization and Implementation

This chapter describes how the proposed solutions from chapter 4 were implemented. In section 5.1 the implementation of the DETOx tool will be discussed. Finally, section 5.2 describes how the embedded operating systems used for evaluation in this thesis were adapted and which benchmarks were selected for them.

5.1. DETOx

In this section the implementation of the DETOx tool and its extensions will be discussed. Therefore, in section 5.1.1 some preprocessing is done to prepare for the use of FAIL* and DETOx. Then, in section 5.1.2, the recording step is discussed and it is shown how to determine whether assertions are triggered during an injection. Then, in section 5.1.3, the profiler step is shown, where the captured data is further processed so that it can be turned into an ILP in the analyzer step. The analyzer is implemented like described in 4.1, 4.2.3 and 4.3.1 which includes also the determination of partitions. The workflow with the most important aspects is shown in figure 5.1 and discussed in more detail in the following sections.

5.1.1. Preparation

Before the execution of an fault injection campaign and further processing by DETOx, the code to be injected must be prepared. The notable preparations are listed below:

- **Prepare for FAIL***
  In the code used, markers must be set at certain points that give FAIL* signals. This markers are realized by symbols which are the reference to a function or a global variable.

  There must be a symbol in the code where FAIL* can register that the program should be terminated because all instructions important for the program have been executed. There are also symbols that can indicate to FAIL* that an fault was detected. In this case there are symbols by which

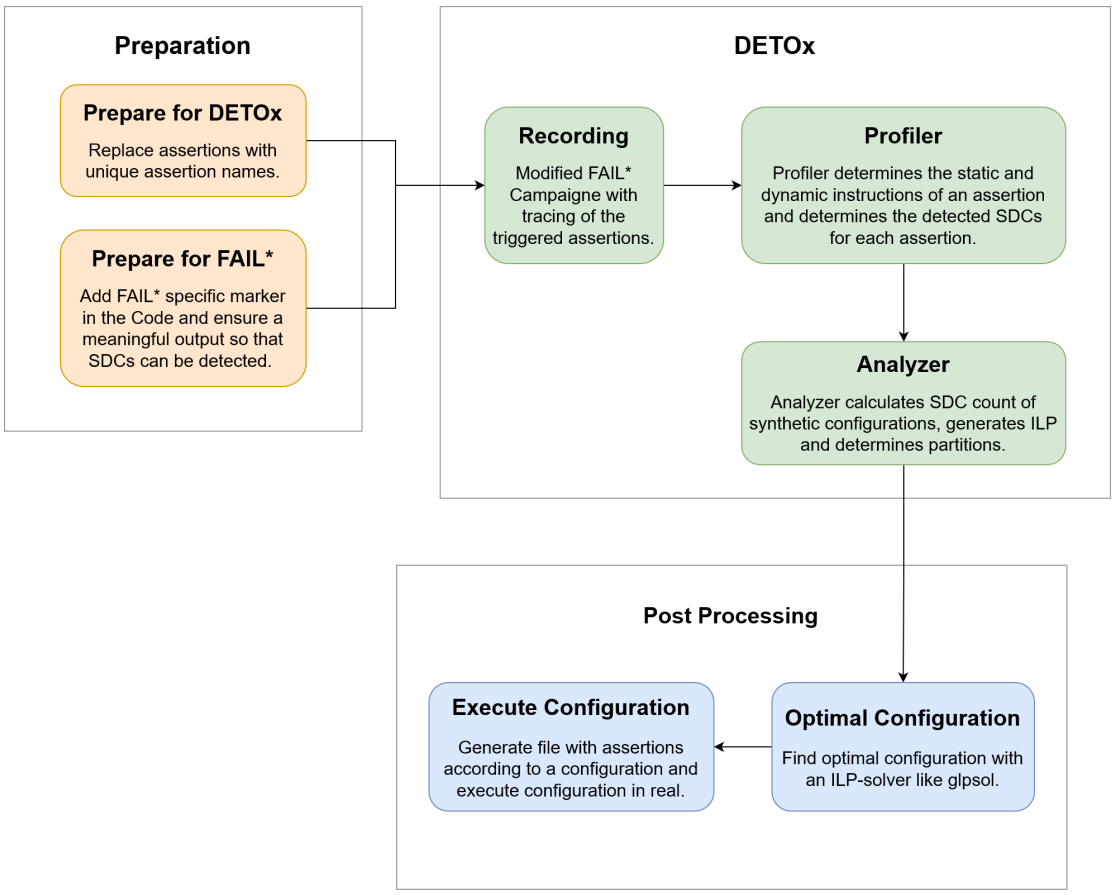

Figure 5.1.: Visualization of the workflow used in this thesis.

FAIL* terminates the program, as well as those by which FAIL* lets the program continue to run, but saves that an error condition was triggered at the corresponding injection. In such a case, after the occurrence of such a symbol, others can be detected during the same fault injection. These symbols are used in combination with the recording of assertions while a fault injection. Triggering an assertion would then no longer terminate the program, but call a function that indicates FAIL* that an assertion was triggered. The insertion of these symbols was done manually or in the case of the symbols, that are used in combination with assertion, also automated with Python scripts.

In addition, the program must produce a suitable output, which is used to register an SDC. If the output for an injection differs from the output of the golden run, but otherwise the program has terminated normally, then the result for that injection is a SDC. The specification of the output is done manually.

- **Prepare for DETOx**

  For DETOx, adjustments must be made to the assertions present in the program. These assertions all need an unique name so that FAIL* in the fault injection campaign knows which assertions exactly are triggered by an injection. This is done using the found assertions for eCos and FreeRTOS from section 3.1. Each occurrence of an assertion in the code is automatically replaced by an unique assertion through a Python script. In addition, an assertion must no longer cause the program to terminate. For recording which assertions are triggered by which injection, the program must be able to continue running. For this purpose an assertion only calls a symbol that tells FAIL* that an assertion was triggered and does nothing else.

5.1.2. Recording

This section explains how to implement the recording of assertions for an injection in FAIL*. The recording step is an extension of the FAIL* fault injection campaign. This means that during the injections of the faults, assertions are monitored for triggering and if one is triggered, it is recorded for the injection. The fault injection is performed in detail by a FAIL* client, which in turn receives the individual fault injection experiments from a server, injects this fault into the program, and then sends the result back to the server. This client can be implemented in different ways. In the default implementation, this client is part of the so-called `generic-experiment`, which implements a common fault injection campaign. Figure 5.2 visualizes the relationship between the FAIL* components and the user-defined experiment. Bochs is an open source IA-32 x86 PC emulator

[1] with which the FAIL* components can communicate. For the user-defined experiment different interfaces are available. It is possible to define listeners that can listen for user-definable symbols. This is particularly interesting for the recognition of assertions. In this case a listener monitors the call of the function `ASSERTION_DETECTED`. This function was automatically placed after the call of each assertion, so that it is called when the assertion is triggered. In addition, the following callback functions are interesting in that context:

- `cb_before_resume()`: This function is called after the fault was injected. At this point the listener should be activated to look for the function `ASSERTION_DETECTED` which means a triggered assertion.
- `cb_during_resume(BaseListener *event)`: This callback function is entered when the function `ASSERTION_DETECTED` is called after the fault injection and the listener catches it. In the callback function the return address for `ASSERTION_DETECTED` is added to the current running injection. After that the listener is reactivatet and the injection resumes unaffected.
- `cb_after_resume(BaseListener *event)`: This callback function is called for example when a symbol is reached that indicates FAIL* to stop. The running injection is finished at this point and the detectors which are collected until then are linked with the injection.

5.1.3. Profiler

The profiler is based on the work of Lenz and Schirmeier [10]. The profiler determines the static and dynamic instructions of an assertion and determines how many failures an assertion detects and how many are added by it. This is done in particular by appropriate database queries. The database contains all relevant data of a FAIL* fault injection campaign.

To get the dynamic instructions, the so-called `fulltrace` table in the database is used, which contains an entire trace of the program execution, in which dynamic instructions are mapped to static ones. Using a disassembler, the unique name of an assertion can be used to find the static instruction where it is located in the assembly code. The `fulltrace` can then be used to find the dynamic instructions of that assertion. Relevant to this approach is that it may be that a function is called within an assertion, such as the `is_sorted` function in the quicksort example from section 3.2. This function would not be recognized as part of the assertion, but must be counted towards it for the correct calculation of SDCs. If this is the case, it is recognized that an instruction of the assertion contains a call of a function. This call gap is bridged and the bridged instructions are added to the dynamic instructions of the assertion.

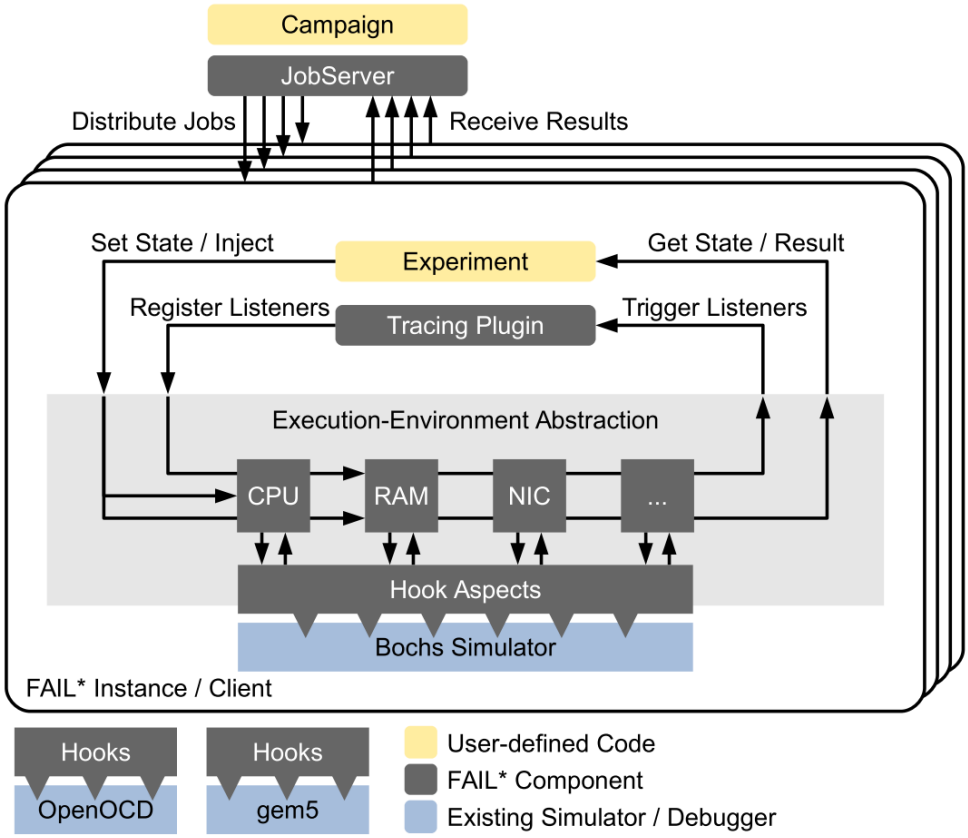

Figure 5.2.: Visualization of the Plumbing Layer of FAIL* [17].

5.2. Embedded Operating Systems

In this thesis, the embedded operating systems FreeRTOS [5] and eCos [3] are used as more complex programs for evaluation. These have been chosen because they have a sufficient number of assertions already existing in the code, as shown in section 3.1, and thus could push the presented solutions to their limits. In particular, eCos contains a lot more assertions compared to FreeRTOS and should therefore be evaluated more intensively.

Both embedded operating systems were prepared according to the preperation step from section 5.1.1. This means that all occurrences of assertions found were automatically replaced with unique assertions in each case, which still check the same condition but no longer terminate the program. Instead, a function called `ASSERTION_DETECTED` is called which indicates FAIL* that an assertion was triggered. In addition, other necessary symbols have been included, for example to indicate the end of the injection.

Different tests are considered for FreeRTOS and eCos. All tests are written in C or C++ and are compiled with the optimization level `-O0`.

The demo used for FreeRTOS is explained below:

- `Blinky_Demo:` For FreeRTOS, a demo for the FreeRTOS IA32 (x86) flat memory model port is used and is pre-configured to run on the Galileo Gen 2 single board computer. This variant is required for FreeRTOS to run in the Bochsemulator of FAIL*. The demo has been modified and extended in this thesis and consists of two tasks which are communicating through a queue. Additionally a mutex was added. On the board the demo would let a led blink. During the FAIL* Campaign the state of the led is written to a file.

For eCos, the benchmarks `kalarm0` and `cnt_sem1` from the eCos kernel test suit, which is supplied with the source code, were used. In addition, a custom test was written, which will be called `kernel_test` in the following. In the following the tests are explained:

- `kalarm0:` This test includes three alarms and one counter. The alarms are all triggered at different timings. The number of alarm triggers was reduced to achieve an acceptable runtime of the fault injection campaign. The test essentially checks that the counter and alarms are working correctly.

- `cnt_sem1:` This test includes three counting semaphores which are used between two tasks. The test essentially checks that the counting semaphores are working correctly.

- `kernel_test:` This is a self created test which should target many different components of the kernel in one test. This can not be achieved by the existing test, because they focus usually on one component only. The kernel_test

includes two tasks, one counter, one alarm, one mailbox (which is essentially a queue) and one semaphore.

The first task sorts an array with bubblesort while after each swap the current value is send over the mailbox to a second task which prints the data to the output. A binary semaphore is used to block the first task until the value is printed. After that a counter is incremented on which an alarm is listening and every four increments a character is printed to the output. The code is added in the appendix in A.

6. Evaluation

In this chapter, the presented solutions for finding the optimal configuration from chapter 4 will be evaluated with the implemented infrastructure from chapter 5.

6.1. Evaluation of a Simple Program

In this section, simple programs are used to investigate whether the method of Integer Linear Programming (ILP) from section 4.2.3 is suitable to find the optimal configuration of assertions for the given programs. In doing so, the programs are chosen to allow the execution of any possible configuration and thus to verify that for these programs the ILP determines the optimal configuration.

As a simple program that does not contain too many assertions so that every configuration can be executed, the quicksort example from section 3.2 is chosen. There are five assertions in this program, resulting in $2^5 = 32$ configurations. First, it will be checked how much the deviation of the real executed configurations, which are already determined in section 3.2, to the calculated synthetic configurations is. This is to determine whether the calculation method from section 4.1 is likely suitable for determining the number of SDCs for a configuration without actually having to run it. Since the basics of the calculating method for the synthetic configurations are also used for the creation of the ILP, it is central for the further procedure to show that the calculation is as close as possible to reality. In the paper of Lenz and Schirmeier [10] it is already stated that the calculation method can be very accurate with an average deviation of about 0.2%, but a larger example shall also be tested at this point. In table 6.1 the number of SDCs of the synthetic configurations $SDC_{synthetic}$ are compared to the real executed SDC_{real} and a deviation is given. The entries are sorted in ascending order by the SDCs number of real executed configurations. The deviation varies from -1.29% to 1.95%. This is small enough that the order of the synthetic configurations largely matches the order of the real executed configurations. The optimal configuration is the same for both. Accordingly, it is further assumed that the computational method can be suitable to calculate the number of SDCs of synthetic configurations. Since the calculation procedure is based on the same fundamentals as the creation of the ILP, the assumption is also made that the procedure is suitable for finding the optimal configuration. In the further course of the evaluation, this assumption will be verified with further real executed con-

figurations.

When creating an ILP with DETOx for the quicksort algorithm and solving it with glpsol from the GLPK package which is intended for large-scale ILP [7], the solution for the optimal configuration is $[0, 0, 0, 0, 1]$. This is the expected result. Therefore the ILP method is working for this simple example. In the next section ILPs for more complex programs are created and evaluated to further verify the method.

6.2. Complex Software: Embedded Operating Systems

After it can be shown in the preceding section that the method of the ILP for simple examples is able to determine the optimal configuration, the procedure is to be extended in this chapter to a more complex program example. For this purpose the embedded operating systems eCos and FreeRTOS are used. Since these contain 224 and 4489 assertions as in section 3.1 determined, not all configurations can be executed or computed for these operating systems. It must be considered that the determined number of assertions is an estimate, since it is not clear which assertions are really covered and executed during a program execution. However, it can be anticipated here that a sufficient number of assertions are executed during an execution of FreeRTOS and eCos, so that not all configurations can be executed or calculated. To verify that the optimal configuration determined by the ILP really has the smallest number of SDCs among all possible configurations, the following methods are proposed:

- **Randomization:** This method is already explained in section 4.2.1. Here, the number of SDCs of a random configuration is computed or executed and, if possible, a high number of random configurations is used to show that the optimal configuration determined by the ILP is not outperformed by a randomly chosen one.

- **Small Hamming Distance:** All possible configurations that are a Hamming distance of one away from the opimal configuration are evaluated. The aim is to investigate whether the direct neighbors of the optimal configuration do not have fewer SDCs compared to it.

In order to validate the calculated configurations, a subset of configurations is executed in real. Furthermore, the runtime of the ILP solver is determined until an optimal solution is found. If the runtime is significant, we also investigate to what extent the assertions of the two embedded operating systems can be partitioned.

[assert1, assert2, assert3, assert4, assert5]	SDC_{real}	$SDC_{synthetic}$	$error$
[0, 0, 0, 0, 1]	667 792	671 174	0.50%
[1, 0, 0, 0, 1]	688 507	680 590	−1.16%
[0, 1, 0, 0, 1]	697 033	696 201	−0.12%
[0, 0, 0, 1, 1]	701 957	693 018	−1.29%
[0, 0, 1, 0, 1]	703 802	698 904	−0.70%
[1, 0, 0, 1, 1]	704 466	702 434	−0.29%
[1, 1, 0, 0, 1]	707 584	705 617	−0.28%
[1, 0, 1, 0, 1]	715 638	708 320	−1.03%
[0, 0, 1, 1, 1]	720 954	720 759	−0.03%
[0, 1, 1, 0, 1]	722 079	723 981	0.26%
[0, 1, 0, 1, 1]	726 452	718 050	−1.17%
[1, 1, 0, 1, 1]	730 459	727 466	−0.41%
[1, 0, 1, 1, 1]	735 990	730 175	−0.80%
[1, 1, 1, 0, 1]	741 354	733 397	−1.08%
[0, 1, 1, 1, 1]	744 953	745 836	0.12%
[1, 1, 1, 1, 1]	755 252	755 252	0.00%
[0, 0, 0, 0, 0]	1 539 162	1 569 759	1.95%
[0, 0, 0, 1, 0]	1 568 798	1 551 790	−1.10%
[1, 0, 0, 1, 0]	1 592 419	1 580 798	−0.74%
[0, 1, 0, 0, 0]	1 606 239	1 604 603	−0.10%
[1, 0, 0, 0, 0]	1 606 692	1 599 027	−0.48%
[0, 1, 0, 1, 0]	1 613 835	1 601 115	−0.79%
[0, 0, 1, 1, 0]	1 626 834	1 623 432	−0.21%
[0, 0, 1, 0, 0]	1 638 825	1 649 377	0.64%
[1, 1, 0, 0, 0]	1 639 217	1 633 677	−0.34%
[1, 0, 1, 1, 0]	1 647 153	1 652 408	0.32%
[1, 1, 0, 1, 0]	1 650 336	1 630 095	−1.24%
[0, 1, 1, 1, 0]	1 670 647	1 673 952	0.20%
[0, 1, 1, 0, 0]	1 672 429	1 685 132	0.75%
[1, 0, 1, 0, 0]	1 679 553	1 678 613	−0.06%
[1, 1, 1, 1, 0]	1 702 363	1 702 928	0.03%
[1, 1, 1, 0, 0]	1 714 920	1 714 202	−0.04%

Table 6.1.: Table with the occurrences of SDCs of a quicksort algorithm with five assertions and correspondingly 32 possible synthetic and real configurations. The error column describes the deviation from the real configuration to the corresponding synthetic configuration. The entries are in ascending order of the number of SDCs from the real configurations.

The evaluations took place on a four Intel Xeon E5-4640 CPU System with 252GB of main memory. A FAIL* fault injection campaign consisted of 64 injection clients which where executed on one logical core each.

6.2.1. Case Study: FreeRTOS

First, the embedded operating system FreeRTOS is evaluated. For this, `Blinky_Demo` is used, which is already provided with the source code and uses the FreeRTOS IA32 (x86) flat memory model port. The characteristics of the demo and the necessary preparations for the fault injection campaign are described in more detail in section 5.2.

In the first step the recording is performed. This means that the extended fault injection campaign is executed for the demo and it is recorded which assertions are triggered by which injections. Then the profiler determines the dynamic and static instructions of the assertions. From this, it determines the added SDCs caused by the execution of the assertions and the detected SDCs. In the process, the profiler detected 60 different assertions that were triggered during the fault injection campaign. In the subsequent step, the analyzer creates an ILP from this data. There is basically no significant runtime required for this. The ILP consists of 38 single variables included in the optimization function, and 20 variables from single variable products, as well as 60 linearization rules. Not all 60 variables are in the optimization function, since in this case only SDCs are optimized for, but assertions are captured for all failures. For solving the ILP, glpsol is used to determine the potential optimal configuration. This has 14 active assertions for the demo used in FreeRTOS.

From this possibly optimal configuration, all 60 configurations are determined that are one Hamming distance away from it. In addition, for a further comparison, the respective edge cases are calculated in which all assertions are active or inactive in a configuration. In addition, 3000 random configurations are also sampled. The runtime of the previous steps is shown in table 6.2. In figure 6.1 all these computed configurations are shown in a plot. In this, it can be seen that the calculated optimal configuration is not outperformed by any other. Among the hamming configurations, however, there are some that generate just as few SDCs. If the respective edge cases are considered, it is noticeable that the configuration with all assertions active belongs to the better configurations and is thus clearly better than the configuration in which all assertions are inactive. In addition, it can be seen from the random configurations that there are two clearly separable clusters of configurations. If in addition the worst hamming configuration is considered, then it can be seen that this is in the worse cluster in contrast to all other hamming configurations. If it is now considered that this differs from the optimal configuration by one assertion, then this assertion can be determined. The assertion is active in the optimal configuration and inactive in the hamming con-

Step	Runtime (in hours)
Recording	1:10
Profiler	0:30
ILP solver	0
Calculate hamming (60)	0:01
Calculate random (3000)	0:08

Table 6.2.: Table with the step in the workflow for the FreeRTOS Demo on the left side and the corresponding runtime on the right side.

figuration. This means that the presence of these assertions has a strong positive influence. In the following the assertion is shown:

```
assert(pxQueueSetContainer->uxMessagesWaiting < pxQueueSetContainer->uxLength);
```

This assertion is located in the `queue.c` file at line 3032. It can be determined from the data of the fault injection campaign that the assertion is not executed in the golden run, i.e. the non-injected program. However, since this was triggered during fault injections, it is reasonable to assume that it was triggered by a corruption of the control flow. Accordingly the assertion would have in the normal fault-free program run no negative influence on the number of SDCs, since this is only then executed, if at certain places faults occurred. However, the positive influence is considerable. This also questions the objective from section 4.2.3 that a configuration with as few active assertions as possible should be found. According to the current knowledge, with the same minimum number of SDCs, it could also make sense to have a configuration with a particularly large number of assertions, which could be triggered by a slightly changed input at fault injection and would otherwise have essentially no negative influence. However, it shows at least that it could be argued for both variants.

In the previous section it can be shown, as far as this is possible under the given constraints, that the calculated optimal configuration cannot be surpassed by any other of the calculated ones. This means for `Blinky_Demo` that the method of the ILP apparently works. In the following, the calculation method itself will be verified by executing real configurations. For a comparison, the optimal configuration, as well as three of the best, three average and the three of the worst hamming configurations are selected. In addition, the confiurations of each of the edge cases and the best, one average and the worst random configuration are selected from the existing dataset. The results are presented in table 6.3. The deviation between synthetic and real configuration varies from -0.06% to 0.66% and is therefore very small. However, it can be seen that one hamming configuration is slightly better in reality. However, these differ only by about one thousandth of a percentage point. The difference between the configuration with all assertions active and the

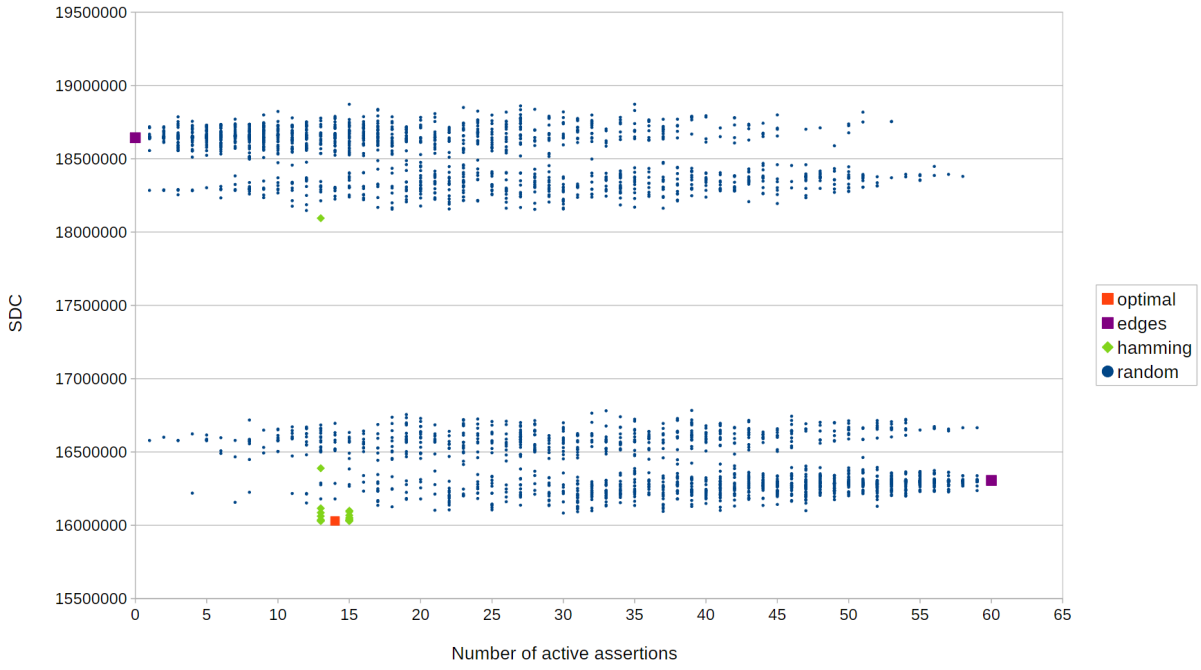

Figure 6.1.: Plot of calculated specific and random configurations of the `Blinky_Demo` test. On the x-axis the number of active assertions of a configuration is plotted. On the y-axis is the amount of SDCs a configuration produces. The calculated optimal configuration is in red, the configurations with hamming distance one to the optimal are green, the edge case configurations were all assertions are active or inactive are shown in purple and the additional random configurations are blue.

Selected from	Amount of active assertion	SDC_{real}	$SDC_{synthetic}$	*error*
Hamming best	15	15 964 992	16 030 016	0.41%
Calculated Optimum	14	15 965 432	16 030 016	0.40%
Hamming best	15	15 965 432	16 030 016	0.40%
Hamming medium	15	15 972 126	16 032 600	0.38%
Hamming best	13	15 989 551	16 030 016	0.25%
Hamming medium	13	16 000 659	16 037 743	0.23%
Hamming medium	13	16 024 704	16 031 409	0.04%
Hamming bad	13	16 049 040	16 115 872	0.41%
All active	60	16 315 968	16 305 786	−0.06%
Hamming bad	13	16 381 011	16 388 842	0.05%
Random best	30	16 026 035	16 083 568	0.36%
Random medium	12	18 027 846	18 147 254	0.66%
Hamming bad	13	18 067 755	18 094 468	0.15%
All inactive	0	18 597 916	18 643 401	0.24%
Random worst	35	18 875 625	18 871 621	−0.02%

Table 6.3.: Table with the occurrences of SDCs of the `Blinky_Demo` with 60 assertions. A comparison is given between special configurations. The error column describes the deviation from the real configuration to the corresponding synthetic configuration. The entries are in ascending order of the number of SDCs from the real configurations.

synthetic optimal configuration is 2.15% in reality and 14.15% for the configuration with all assertions inactive. Accordingly, the improvement in fault tolerance with the optimal configuration is comparatively small to the configurations which would be normally used in practice.

Since the ILP can be solved practically immediately, further optimizations to the runtime, such as partitioning, are not necessary. It can also be seen that the recording step in particular is time consuming. This is due to the difficult to solve circumstance that the fault injection must be injected into all relevant points in order to collect reliable data about the triggering of the assertions.

6.2.2. Case Study: eCos

An evaluation is also to be provided for the embedded operating system eCos. The existing tests `cnt_sem1` and `kalarm0` that come with the source code of eCos are used. In addition an own test was written, which is called here `kernel_test`. In `kernel_test` a wider range of kernel functionalities are used within one test than in the existing tests. At least compared to those that can be injected in a reasonable time in a FAIL* campaign. The characteristics of the benchmarks are

Step	Runtime (in hours)
Recording	0:21
Profiler	0:02
ILP solver	0
Calculate hamming (238)	0:03
Calculate random (3000)	0:23

Table 6.4.: Table with the step in the workflow for the `kalarm0` test on the left side and the corresponding runtime on the right side.

explained in section 5.2. The evaluation procedure used here is the same as in the previous section.

For the eCos tests, the first step is to record the assertions. The profiler then pre-processes the resulting data and the analyzer uses it to create the ILP. This then solves glpsol and with the calculated optimal configuration all further configurations with hamming distance one are created, as well as the edge case configurations and random configurations. In the following sections, the results will be discussed in more detail.

6.2.2.1. eCos test: `kalarm0`

The runtimes for the according steps are shown in table 6.4. Due to the shorter program runtime, `kalarm0` is significantly faster in the recording and profiler step. But `kalarm0` contains significantly more assertions, which are more complexly interrelated. Thus the computation of new assertions takes longer. The profiler has detected 238 triggered assertions after the fault injections. The ILP consists of 39 single variables included in the optimization function, and 148 variables from single variable products, as well as 444 linearization rules. The computed optimal configuration contains 12 active assertions.

As for FreeRTOS, the selected configurations are illustrated in Figure 6.2. Again, the optimal configuration calculated from the ILP is at least as good as any other configuration shown. It is also evident from the edge case configurations and the plotted trend in the random configurations that there is a clear tendency towards higher susceptibility to faults as the number of active assertions in a configuration increases. In addition, two separable clusters of configurations are also recognizable here. It is to be seen just like with FreeRTOS that there is exactly one hamming configuration, which is in the worse cluster. After the same method as with FreeRTOS the assertion with this high influence is determined. The assertion is inactive in the optimal configuration. This means that this assertion appears to add significantly more SDCs than it detects. The assertion is shown below:

```
CYG_ASSERTCLASS(alarm, "Bad alarm in counter list" );
```

The assertion is located in the file `clock.cxx` in line 230. The assertion takes a relatively long time to be evaluated. At the same time, the detected SDCs cannot compensate for the additional ones. Figure 6.3 shows the faultspace for `kalarm0` with this assertion. From this it can be seen that the assertion is called frequently. This adds up to the negative effect, resulting in the significantly negative effect on fault tolerance when this assertion is included in a configuration. Assertions with a significantly negative effect could also be responsible for the visible trend in Figure 6.2, since the probability increases with an increasing number of active assertions that they are also included in the configuration.

The ILP also seems to provide the best solution for `kalarm0` when compared with the collected data. To validate the computed configuration, real executed configurations are again compared with the computed ones, as in the evaluation of FreeRTOS. This comparison is shown in table 6.5. A relatively small difference of -1.03% to 1.23% can be observed between real and synthetic configurations. However, it is noticeable that there are three better configurations than the calculated optimal configuration. This could be explained by the fact that the optimal configuration does not particularly stand out and there are very many configurations close to the optimum, as can also be seen in Figure 6.2. In this case, the optimal configuration is also close to the configuration where all assertions are inactive, with only a deviation of 0.1%. Despite all this, the configuration determined here is probably also a good solution in reality, although the best configuration determined here is 1% better. In comparison to the configuration where all assertion are active which is seemingly one of the worst configurations, the calculated optimal configuration is 22% better.

For this test, the ILP was set up to also optimize for the smallest possible number of active assertions. From the table 6.5 it could be concluded that this was not successful, since for example one of the best hamming configurations generates the same number of calculated SDCs, but contains one assertion less. It could not be determined exactly where the inaccuracy came from, however, an ILP was also created without this optimization and the result was an optimal configuration with 17 active assertions. So 4 active assertions more compared to the original optimal configuration. Possible causes could be in the glpsol, which could have a tolerance for the termination criterion, whereby no information was discovered for this, or the ILP is not set up completely correctly.

6.2.2.2. eCos test: `cnt_sem1`

As evaluated in the sections before the runtimes for the several steps are shown in table 6.6. It is striking that the runtime for the calculation of a configuration is quiet high. The reason might be that there are more assertions to be considered in

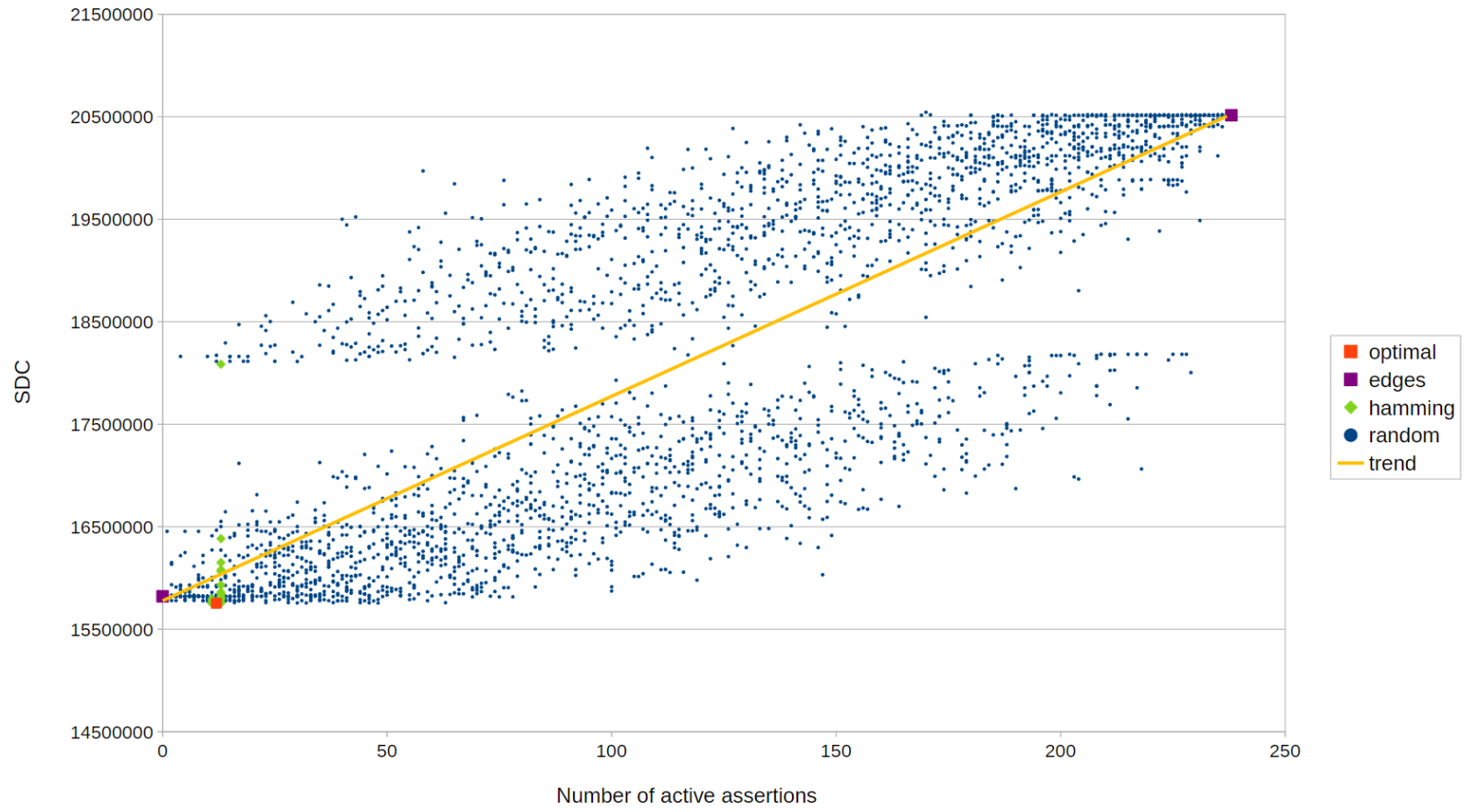

Figure 6.2.: Plot of calculated specific and random configurations of the `kalarm0` test. On the x-axis the number of active assertions of a configuration is plotted. On the y-axis is the amount of SDCs a configuration produces. The calculated optimal configuration is in red, the configurations with hamming distance one to the optimal are green, the edge case configurations were all assertions are active or inactive are shown in purple and the additional random configurations are blue. A trend line through the random configurations is yellow.

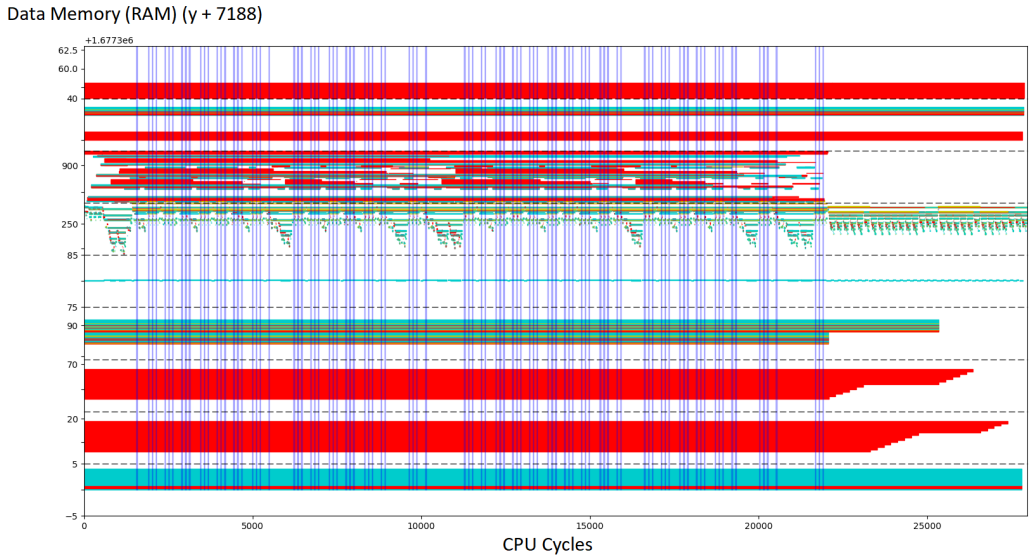

Figure 6.3.: Plot of the faultspace of the `kalarm0` test. On the x-axis the CPU cycles and on the y-axis the data memory addresses are plotted. The resulttypes have different colours. SDCs are red, traps are cyan, timeouts are yellow, ok marker are white and detected injections are green. The execution of the significant assertion is in transparent blue. The y-axis is squashed to focus on the important parts.

Selected from	Amount of active assertion	SDC_{real}	$SDC_{synthetic}$	*error*
Random best	48	15 658 833	15 754 753	0.61%
Hamming medium	11	15 761 018	15 754 137	−0.04%
Hamming best	11	15 800 959	15 753 217	−0.30%
Calculated Optimum	12	15 818 712	15 753 217	−0.42%
Hamming best	13	15 818 712	15 753 217	−0.42%
Hamming best	13	15 819 974	15 753 217	−0.42%
All inactive	0	15 836 755	15 821 577	−0.1%
Hamming medium	13	15 849 348	15 793 217	−0.36%
Hamming medium	13	15 947 378	15 931 811	−0.10%
Hamming bad	13	16 201 012	16 150 495	−0.31%
Hamming bad	13	16 402 500	16 384 249	−0.11%
Hamming bad	13	18 107 146	18 085 447	−0.12%
Random medium	125	18 277 171	18 090 057	−1.03%
Random worst	170	20 290 547	20 543 376	1.23%
All active	238	20 320 097	20 514 704	0.95%

Table 6.5.: Table with the occurrences of SDCs of the `kalarm0` test with 238 assertions. A comparison is given between special configurations selected from the synthetic configurations. The error column describes the deviation from the real configuration to the corresponding synthetic configuration. The entries are in ascending order of the number of SDCs from the real configurations.

Step	Runtime (in hours)
Recording	0:52
Profiler	0:15
ILP solver	0
Calculate hamming (239)	0:14
Calculate random (3000)	3:21

Table 6.6.: Table with the step in the workflow for the `cnt_sem1` test on the left side and the corresponding runtime on the right side.

`kalarm0` when looking on SDCs. The profiler has detected 239 triggered assertions after the fault injections. The ILP consists of 65 single variables included in the optimization function, and 103 variables from single variable products, as well as 309 linearization rules. The computed optimal configuration contains 17 active assertions.

As before, the selected configurations are illustrated in Figure 6.4. Again, the optimal configuration determined by the ILP is the best among the synthetic ones. As with `kalarm0`, a trend towards more SDCs with more active assertions can be observed.

The synthetic configurations are compared in table 6.7. From this, a variation of -2.19% to -0.67% is observed. It is noticeable that there could be a bias in the deviation, as it appears to be quite regular. It can be seen that, again, the calculated optimal configuration is outperformed in real execution by two other configurations selected here. However, the real best one is only 0.13% better than the synthetic optimal configuration. This in turn is 11% better than the configuration in which all assertions are active and 0.7% better than in the configuration in which all assertions are inactive.

Also for `cnt_sem1`, both the computation of the synthetic configurations and the determination of the optimal configuration by the ILP seem to be useful.

6.2.2.3. eCos test: `kernel_test`

As evaluated in the sections before the runtimes for the several steps are shown in table 6.8. Here the runtime for calculating on configuration is extremely high. Much more assertions are used in the optimization function and the assertions are related to each other in a much more complex way compared to the previous programs. The profiler has detected 278 triggered assertions after the fault injections. The ILP consists of 92 single variables included in the optimization function, and 6 185 variables from single variable products, as well as 18 555 linearization rules. The computed optimal configuration contains 23 active assertions.

As before, the selected configurations are illustrated in Figure 6.4. Again, the

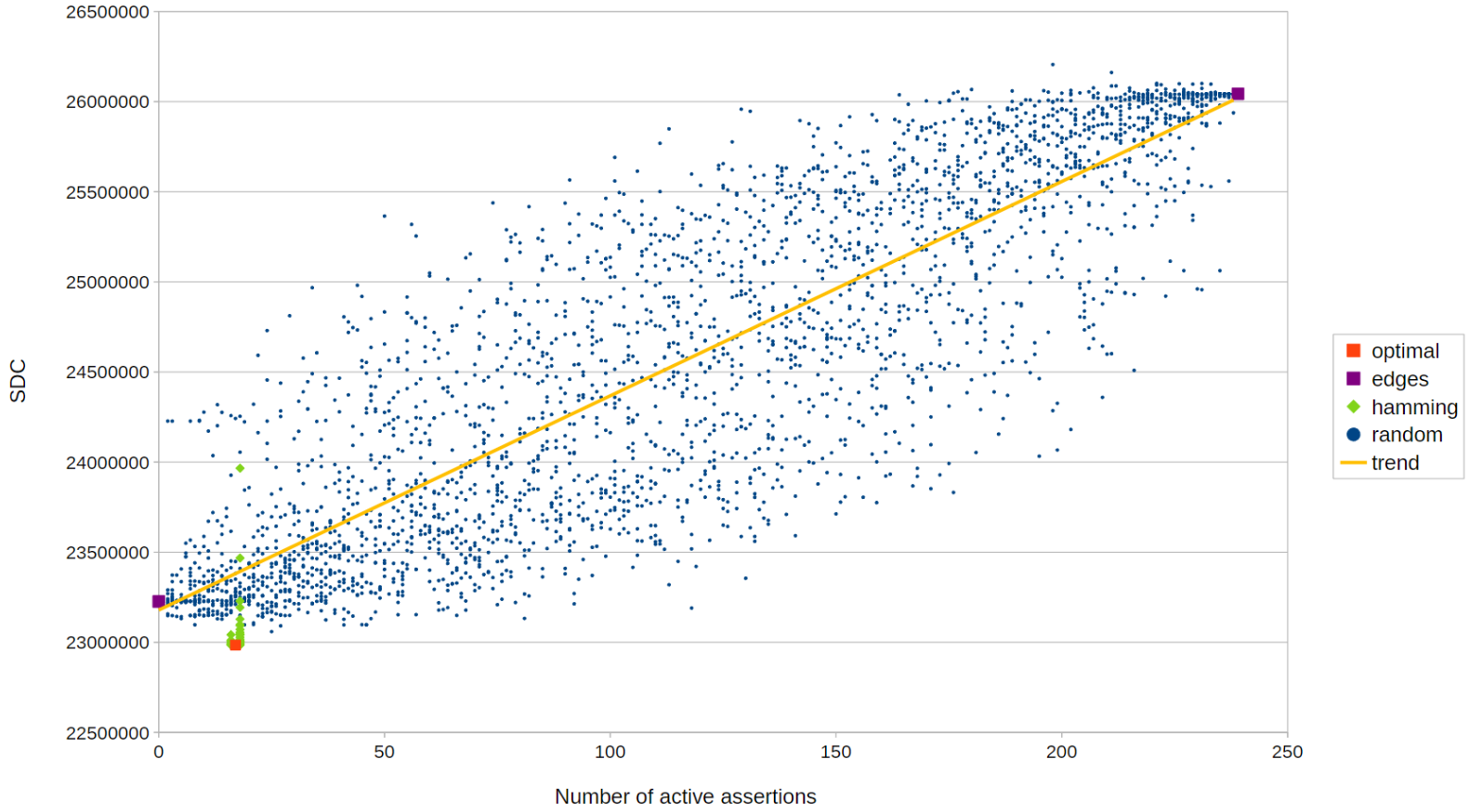

Figure 6.4.: Plot of calculated specific and random configurations of the `cnt_sem1` test. On the x-axis the number of active assertions of a configuration is plotted. On the y-axis is the amount of SDCs a configuration produces. The calculated optimal configuration is in red, the configurations with hamming distance one to the optimal are green, the edge case configurations were all assertions are active or inactive are shown in purple and the additional random configurations are blue. A trend line through the random configurations is yellow.

Selected from	Amount of active assertion	SDC_{real}	$SDC_{synthetic}$	*error*
Hamming best	18	23 391 314	22 984 935	−1.77%
Random best	25	23 402 911	23 059 427	−1.49%
Calculated Optimum	17	23 421 065	22 984 935	−1.90%
Hamming best	18	23 421 065	22 984 935	−1.90%
Hamming best	16	23 433 781	22 984 935	−1.95%
Hamming medium	18	23 451 499	23 001 405	−1.96%
Hamming medium	18	23 479 371	23 097 325	−1.65%
Hamming medium	18	23 495 313	22 992 425	−2.19%
Hamming bad	18	23 578 524	23 228 682	−1.51%
All inactive	0	23 589 612	23 226 525	−1.56%
Hamming bad	18	23 923 761	23 467 631	−1.94%
Hamming bad	18	24 348 891	23 965 937	−1.60%
Random medium	170	24 942 424	24 471 938	−1.92%
All active	239	26 218 387	26 043 142	−0.67%
Random worst	198	26 543 290	26 205 590	−1.29%

Table 6.7.: Table with the occurrences of SDCs of the `cnt_sem1` test with 239 assertions. A comparison is given between special configurations selected from the synthetic configurations. The error column describes the deviation from the real configuration to the corresponding synthetic configuration. The entries are in ascending order of the number of SDCs from the real configurations.

Step	Runtime (in hours)
Recording	1:40
Profiler	2:30
ILP solver	0:54
Calculate hamming (278)	3:10
Calculate random (3000)	18:45

Table 6.8.: Table with the step in the workflow for the `kernel_test` on the left side and the corresponding runtime on the right side.

Selected from	Amount of active assertion	SDC_{real}	$SDC_{synthetic}$	error
Hamming best	24	54 850 327	55 004 278	0.28%
Calculated Optimum	23	55 287 299	55 004 278	−0.51%
Hamming best	24	55 287 299	55 004 278	−0.51%
Hamming best	22	55 320 957	55 004 278	−0.58%
Hamming medium	24	55 587 640	55 403 839	−0.33%
Hamming medium	24	55 655 636	55 111 732	−0.99%
Random best	62	55 935 942	55 557 162	−0.68%
Hamming medium	24	56 062 364	55 182 299	−1.59%
Hamming bad	24	56 110 173	56 510 425	0.71%
Hamming bad	22	57 510 510	57 035 128	−0.83%
Hamming bad	24	58 771 788	58 044 047	−1.25%
All inactive	0	61 147 513	61 064 559	−0.14%
Random medium	96	63 042 356	62 779 626	−0.42%
All active	279	69 826 996	69 821 480	−0.01%
Random worst	272	70 344 475	69 953 977	−0.56%

Table 6.9.: Table with the occurrences of SDCs of the `kernel_test` with 279 assertions. A comparison is given between special configurations. The error column describes the deviation from the real configuration to the corresponding synthetic configuration. The entries are in ascending order of the number of SDCs from the real configurations.

optimal configuration determined by the ILP is the best among the synthetic ones. As with `kalarm0` and `cnt_sem1`, a trend towards more SDCs with more active assertions can be observed. Overall, the optimal configuration stands out much more than in the other benchmarks.

In table 6.9 the synthetic configurations are compared with the real ones. A deviation of −1.59% to 0.71% is observed. In this case, a hamming configuration is 0.8% better than the synthetic optimal configuration in reality. The synthetic optimal configuration is 20.8% better than the configuration with all assertions active and 9.6% better than the configuration with all assertions inactive.

From the table 6.8 it can be seen that the runtime of the glpsol to solve the ILP was 54 minutes. This is a sufficiently long runtime to justify the use of partitions. These are determined according to chapter 4.3.1. This resulted in 4 partitions, with 152 of the 155 relevant assertions in one alone. Accordingly, the approach does not seem to be suitable in this case. To reduce the runtime of the ILP anyway, it is suggested to limit the computation time of the glpsol and to use the result computed up to this point. A corresponding time series is visualized in figure 6.6. Here, the glpsol was started with the time limit plotted on the x-axis and the optimal configuration calculated up to this point was determined. The number of

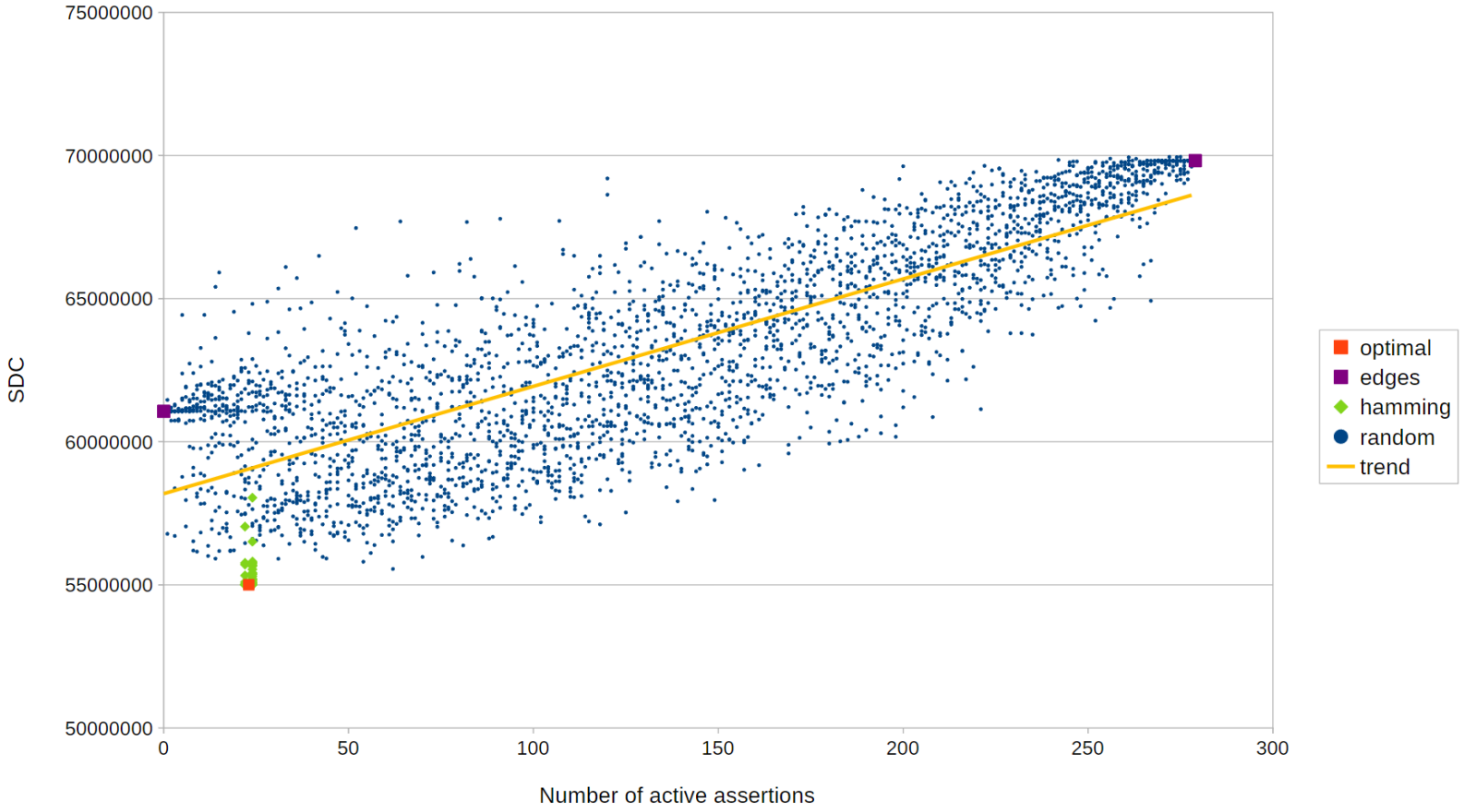

Figure 6.5.: Plot of calculated specific and random configurations of the `kernel_test`. On the x-axis the number of active assertions of a configuration is plotted. On the y-axis is the amount of SDCs a configuration produces. The calculated optimal configuration is in red, the configurations with hamming distance one to the optimal are green, the edge case configurations were all assertions are active or inactive are shown in purple and the additional random configurations are blue. A trend line through the random configurations is yellow.

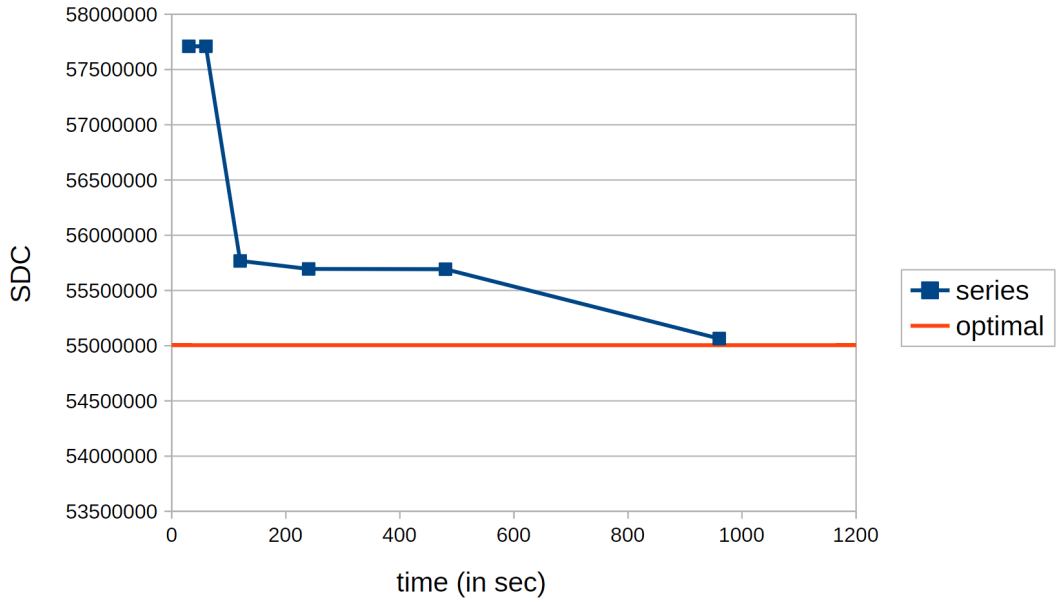

Figure 6.6.: ILP time series of `kernel_test`. On the x-axis is the time limit to solve the ILP for `kernel_test` and on the y-axis is the number of SDCs which where the result for the corresponding configuration determined with the ILP.

SDCs it generates is plotted on the y-axis. It shows that already after 2 minutes a configuration was calculated, which was 1.4% worse than the optimum. After 16 minutes, a configuration was calculated that was 0.1% worse than the optimal configuration calculated after 54 minutes. The problem with reducing the time for the ILP solver is that it is not clear exactly how good a configuration calculated by it really is, since full confidence would require calculating to the end.

6.3. Influence of Non-Protected Application Data

In this section it will be examined to what extent large unprotected memory areas can have an effect on the result of the optimal configuration. For this purpose, an additional array was added to the quicksort algorithm from section 3.2, which is created at the beginning of the program and written to the output at the end. Nothing else happens to this array. The memory of the array is not protected by any assertion. A fault injection into this array results in an SDC. The program was tested with such an array with 100 elements and in another run with 1000. The results of all synthetic configurations were determined. The results are presented

Configuration	$SDC_{synthetic}$
[0, 0, 0, 0, 1]	6 636 910
[1, 0, 0, 0, 1]	6 723 446
[0, 0, 0, 1, 1]	6 843 736
[0, 1, 0, 0, 1]	6 874 854
[0, 0, 1, 0, 1]	6 877 207
[1, 0, 0, 1, 1]	6 930 272

(a) With array of 100 elements.

Configuration	$SDC_{synthetic}$
[0, 0, 0, 0, 0]	166 578 704
[1, 0, 0, 0, 0]	167 361 618
[0, 0, 0, 1, 0]	168 410 020
[0, 1, 0, 0, 0]	168 698 035
[0, 0, 1, 0, 0]	168 760 105
[0, 0, 0, 0, 1]	168 997 128

(b) With array of 1000 elements.

Table 6.10.: Table with the occurrences of SDCs of a quicksort algorithm with five assertions and the six best configurations sorted after the number of SDCs. In addition an array with 100 and 1000 elements was added to the program.

in table 6.10 and include the six best configurations of each run. The comparison is made with the synthetic configurations from table 6.1. It can be seen that with the array containing 100 elements, the optimal configuration is preserved, as well as largely the original ranking by number of SDCs. The distance to the second best configuration with 1.29% is about the same compared to the variant without the array with a distance of the best to the second best of 1.38%. With the array with 1000 elements it is obvious that the order has changed. Also, the configuration with all assertion inactive is now the best configuration.

This confirms the considerations from section 3.2. In this, it is stated that during the execution of an assertion, additional SDCs can arise if faults are injected into another unprotected memory area in the meantime. The larger the unmonitored memory becomes, the worse most assertions become, since they add more and more additional SDCs, but are detecting not more. This shows a fundamental problem in determining an optimal configuration, because it also depends on the size of the unproteced memory, which could change dynamically in real applications. As a result, it is not possible to conclude directly from the optimal configuration of a program to the optimal configuration of a similar program.

6.4. Different Optimization Levels

In this section, it will be investigated to what extent different optimization levels affect the accuracy of the calculation of the synthetic configurations. Up to now, all programs were compiled with the optimization level `-O0`. In addition, the level `-O2` is now to be examined exemplarily. For this, the quicksort algorithm from section 3.2 is considered, as well as the `kernel_test`.

Table 6.11 shows the results of the quicksort with `-O2`. It can be seen that

there is a significant difference of −6.64% to 14.58% between the synthetic and real executed configurations. It is noticeable that in particular there is a high deviation when `assert5` should be inactive. When the data generated by the profiler is analyzed, it becomes apparent that DETOx could not determine dynamic instructions for this assertion. This leads to the problem that the SDCs, which were created during the execution of the assertion, are not subtracted, if this assertion is supposed to be inactive in a configuration. This seems to be a major problem in this case, since the number of SDCs in the synthetic configuration is greatly overestimated especially when `assert5` is not included. The exact reasons for the lack of identification of the dynamic instructions are presently unknown. However, an improvement could be achieved before, since originally DETOx was not able to recognize `assert5` at all in the quicksort example. There were problems with the processing of the text of the source code of the injected programs by FAIL*, which caused that some lines of the code were not recognized as such. This problem could be solved.

Furthermore, table 6.12 shows the results of `kernel_test` with `-O2`. For this the configuration with all assertions active and random configurations were evaluated. It is noticeable that the deviation varies from −2.19% to 0.88% and is considerably smaller than in the quicksort example. Possibly the problem of the unrecognized dynamic instructions does not play such a large role, since it is distributed over many assertions. However, at least in this example, it does not seem to fundamentally change the result and therefore possibly only occurs in certain less frequent cases.

In principle, it would be desirable if other optimization levels would also work safely, since the use of these alone, as can be seen in the quicksort example, can significantly reduce the number of SDCs and improve the fault tolerance.

6.5. Limitations of the Approach

As shown in section 6.3, an optimal configuration for a particular program cannot be applied generally to similar program types. The method is highly dependent on, for example, the size of the unprotected memory, but possibly also on different program inputs. At this point a problem comes to play, which applies with fault injection in general: Each fault injection campaign is applicable in principle first only to the program into which it was injected. To analyze and perhaps mitigate the problem, different variants of a program with different representative inputs would have to be checked.

Another limitation is the quality of the existing assertions. If they are not able to detect SDCs sufficiently or if they are basically not available at all, then this method might provide trivial solutions, such as that all assertions should be inactive. Basically, this could be the case in eCos, where there seems to be a clear

Configuration	SDC_{real}	$SDC_{synthetic}$	error
[0, 0, 0, 0, 1]	241 990	250 765	3.50%
[0, 0, 0, 1, 1]	252 026	266 713	5.51%
[0, 1, 0, 0, 1]	252 488	260 522	3.08%
[0, 1, 0, 1, 1]	256 270	276 470	7.31%
[0, 0, 1, 0, 1]	260 966	269 569	3.19%
[0, 0, 1, 1, 1]	263 994	285 513	7.54%
[0, 1, 1, 0, 1]	271 150	279 292	2.92%
[0, 1, 1, 1, 1]	274 176	295 236	7.13%
[1, 0, 0, 1, 1]	289 863	289 995	0.05%
[1, 1, 0, 0, 1]	290 324	283 804	−2.30%
[1, 1, 0, 1, 1]	291 261	299 752	2.83%
[1, 0, 0, 0, 1]	292 250	274 047	−6.64%
[1, 0, 1, 0, 1]	296 534	292 851	−1.26%
[1, 0, 1, 1, 1]	301 795	308 795	2.27%
[1, 1, 1, 0, 1]	310 649	302 574	−2.67%
[1, 1, 1, 1, 1]	318 518	318 518	0.00%
[0, 1, 0, 0, 0]	544 239	605 410	10.10%
[0, 0, 0, 0, 0]	545 203	605 484	9.96%
[0, 0, 0, 1, 0]	546 062	626 313	12.81%
[0, 1, 0, 1, 0]	551 799	645 946	14.58%
[0, 0, 1, 1, 0]	575 619	669 341	14.00%
[0, 1, 1, 0, 0]	586 799	648 340	9.49%
[0, 0, 1, 0, 0]	593 743	651 132	8.81%
[0, 1, 1, 1, 0]	595 729	688 758	13.51%
[1, 1, 0, 0, 0]	655 207	672 639	2.59%
[1, 1, 0, 1, 0]	661 451	713 175	7.25%
[1, 0, 0, 1, 0]	668 162	693 542	3.66%
[1, 0, 1, 1, 0]	685 187	736 570	6.98%
[1, 0, 0, 0, 0]	695 293	672 713	−3.36%
[1, 1, 1, 0, 0]	701 843	715 569	1.92%
[1, 0, 1, 0, 0]	705 957	718 361	1.73%
[1, 1, 1, 1, 0]	707 979	755 987	6.35%

Table 6.11.: Table with the occurrences of SDCs of a quicksort algorithm with five assertions and correspondingly 32 possible synthetic and real configurations. The program is compiled with `-O2`. The error column describes the deviation from the real configuration to the corresponding synthetic configuration. The entries are in ascending order of the number of SDCs from the real configurations.

Amount of active assertion	SDC_{real}	$SDC_{synthetic}$	error
37	24134449	24330815	0.81%
3	25729509	25178613	−2.19%
150	27459016	27684347	0.81%
141	27528308	27772403	0.88%
186	28172922	28283010	0.39%
215	29271654	29264785	−0.02%
248	29742339	29480267	−0.89%
256	29899800	29767065	−0.45%
282	30787696	30606906	−0.59%

Table 6.12.: Table with the occurrences of SDCs of the `kernel_test` compiled with `-O2` with 282 assertions. A comparison is given between random configurations and the configuration with all assertions active. The error column describes the deviation from the real configuration to the corresponding synthetic configuration. The entries are in ascending order of the number of SDCs from the real configurations.

trend towards a worse and worse SDC value, the more assertions are active in a configuration. Despite all this, the presented method creates a significantly greater certainty of having really found the optimal configuration, since, for example, the `kernel_test` also follows this trend, but the optimal configuration can be significantly different. In the end there is in principle no trivial solution, because it is simply unknown which solution could be more optimal when only trying is a choice.

7. Conclusion and Future Work

This chapter concludes the thesis in section 7.1 and gives an outlook on future work in section 7.2.

7.1. Conclusion

The goal of the thesis was to investigate the use of existing assertions on fault tolerance and then to advance a method that is able to find the optimal configuration of assertions. This was preceded by the realization in chapter 3 that the best fault tolerance cannot necessarily be achieved by using as many assertions as possible. In principle, an optimal configuration can be any possible configuration, provided there is no prior knowledge about it.

In the following, the causes were explained and how the assertions are interrelated. It was stated that assertions can detect not only failures, but also add additional ones by their own execution. By the execution the run time and accordingly also the attack surface of the program increase. An assertion must therefore detect more failures than add new ones. However assertions are also related to each other. There on the one hand the dependency through redundancy was shown, which brings assertions in connection, that protect the same parts of the memory. And on the other hand, the dependency through time overlap was also identified, where assertions are related that occur in injections at the same time.

Proposed solutions are then discussed in chapter 4. The DETOx tool on which this thesis is based has a significant role to play here. With this tool it is possible to calculate configurations by recording a run with all assertions active. In this run, all triggered assertions for the respective injections are recorded. This makes it possible to calculate the number of failures of a configuration much faster than executing them in reality. However also this method is too slow for programs with a larger number of assertions. Therefore an Integer Linear Program is derived, which should be able on similar bases as the computation of configurations to find a global optimal configuration. In the following it is suggested with the help of partitions to divide the optimization problem into smaller parts, in order to simplify it. Among other things the dependencies between assertions are considered.

In chapter 6 both the ILP and the calculation method could then be validated for a total of five different programs and two embedded operating systems.

However, the partitioning idea was not very successful, since the assertions could hardly be separated. However the method could possibly be suitable for substantially larger programs with clearly more assertions, like the Linux Kernel. It has already been shown for the `kernel_test` that the time required by an ILP for the solution can grow over-proportionally with increasing complexity. Furthermore, it was evaluated how accurate the calculation of configurations is in the context of different optimization levels. It could already be shown by the previous evaluation that for the optimization level -O0 the calculation has a high accuracy, but there is room for improvement for -O2 and presumably also for further optimization levels.

Over all programs from FreeRTOS and eCos an average improvement for the optimale configurations to the configuration with all assertions active of 14% was achieved. And the average improvement for the optimale configurations to the configuration with all assertions inactive was 6.6%. While these improvements are relatively moderate it should be considered that these improvements comes at virtually no cost for the program performance. With less active assertion in the program it becomes not only more fault tolerant, but also faster.

In conclusion, it can be stated that the method for calculating an optimal configuration using an ILP, as well as the calculation of the configurations themselves, worked for the considered programs. However, limitations must also be pointed out. On the one hand it can be observed that when adding or removing unprotected memory areas, the optimal configuration can change. The more unprotected memory is present, the worse an assertion tends to become and produces eventually more failures than it detects. Accordingly, a determined optimal configuration cannot be used for similar programs and not even for the same program if the state in it has changed or different inputs are used. Beyond that the fault tolerance method presented here stands and falls with the presence of assertions.

In principle the procedure developed in this thesis could be applied also to other detector mechanisms for fault detection. The problem that for a detector conditions must be examined, which increase the run time of the program and thus its attack surface, is a general one. However, for these detectors it must then be possible to determine what benefit they bring and how much they cost. So there must be, for example, clues that an instruction is that of a detector and which one. In addition, the detector should be compact. This means that all instructions necessary for it should be executed in one piece if possible. Otherwise it becomes considerably more complicated to link the scattered instructions, which belong to the detector, with it.

7.2. Future Work

A problem highlighted in this thesis for the computation of configurations is the inaccuracy at optimization level `-O2` and others except `-O0`. The problem must be solved that assertions sometimes cannot be associated with their dynamic instructions even though they have been executed.

Furthermore, the runtime to compute configurations with many assertions could be improved. This has proven to be the biggest bottleneck in the evaluation. The current method relies on deleting the assertions to be removed from each injection individually. This is an enormously time consuming procedure when many injections and many assertions occur. The considerations from the creation of the ILP could be used for a faster procedure that is based on the addition and subtraction of occurrences of assertions. In addition several computations are parallelizable and thus also multi threading would be suitable.

In order to extend the evaluation the procedure could be applied in the future for example to the Linux Kernel, since it contains substantially more assertions than the programs considered here. In this regard also the partitioning of assertions could be considered again, since it is foreseeable that an extensive ILP would emerge. Possibly, also another previous work [11] could be used, which is to produce a code coverage as high as possible in the Linux Kernel. For the processing of large programs sampling is then presumably necessary, which would have to be intigrated.

The developed procedure could be used as well in combination with assertion generating methods. For example, a redundant variable could be created after each potential write operation of a variable, which is then checked with an assertion before a potential read access. Thus assertions are aggressively added to the program. With the method developed in the thesis then the optimal selection could be determined from these assertions.

Bibliography

[1] *bochs*. https://bochs.sourceforge.io/. Accessed: 2022-02-24.

[2] *Die assert-Anweisung*. https://dbs.cs.uni-duesseldorf.de/lehre/docs/java/javabuch/html/k100044.htmll. Accessed: 2022-02-24.

[3] *eCos*. https://ecos.sourceware.org/. Accessed: 2022-02-24.

[4] A. E. Eiben and J. E. Smith. 'What Is an Evolutionary Algorithm?' In: *Introduction to Evolutionary Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 25–48. ISBN: 978-3-662-44874-8.

[5] *FreeRTOS (Real-time operating system for microcontrollers)*. https://www.freertos.org/. Accessed: 2022-02-24.

[6] Fred Glover and Eugene Woolsey. 'Further Reduction of Zero-One Polynomial Programming Problems to Zero-One linear Programming Problems'. In: *Operations Research* 21.1 (1973), pp. 156–161. DOI: 10.1287/opre.21.1.156. eprint: https://doi.org/10.1287/opre.21.1.156. URL: https://doi.org/10.1287/opre.21.1.156.

[7] *GLPK (GNU Linear Programming Kit)*. https://www.gnu.org/software/glpk/. Accessed: 2022-02-24.

[8] Daniel Gomez Toro. 'Temporal Filtering with Soft Error Detection and Correction Technique for Radiation Hardening Based on a C-element and BICS'. PhD thesis. Dec. 2014.

[9] M. R. Guthaus et al. 'MiBench: A Free, Commercially Representative Embedded Benchmark Suite'. In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC '01. USA: IEEE Computer Society, 2001, 3–14. ISBN: 0780373154.

[10] Michael Lenz and Horst Schirmeier. 'DETOx: Towards Optimal Software-based Soft-Error Detector Configurations'. In: *Proceedings of the 12th European Dependable Computing Conference (EDCC '16)* (Gothenburg, Sweden). Fast abstract. Sept. 2016.

[11] Alexander Lochmann, Robin Thunig and Horst Schirmeier. 'Improving Linux-Kernel Tests for LockDoc with Feedback-driven Fuzzing'. In: *CoRR* abs/2009.08768 (2020). arXiv: 2009.08768. URL: https://arxiv.org/abs/2009.08768.

[12] David G Luenberger, Yinyu Ye et al. *Linear and nonlinear programming*. Vol. 2. Springer, 1984.

[13] B. Nicolescu, Y. Savaria and R. Velazco. 'Software detection mechanisms providing full coverage against single bit-flip faults'. In: *IEEE Transactions on Nuclear Science* 51.6 (2004), pp. 3510–3518. DOI: 10.1109/TNS.2004.839110.

[14] Nahmsuk Oh, Subhasish Mitra and Edward J. McCluskey. 'ED4I: Error Detection by Diverse Data and Duplicated Instructions'. In: *IEEE Trans. Comput.* (2002), 180–199.

[15] David Powell et al. 'Estimators for Fault Tolerance Coverage Evaluation'. In: *IEEE Transactions on Computers* 44 (Sept. 1995). DOI: 10.1109/12.364537.

[16] G.A. Reis et al. 'SWIFT: software implemented fault tolerance'. In: *International Symposium on Code Generation and Optimization*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[17] Horst Schirmeier. 'Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance'. Dissertation. Technische Universität Dortmund, July 2016. DOI: 10.17877/DE290R-17222.

[18] *Wenn Weltraumpartikel das Smartphone abstürzen lassen*. https://www.heise.de/tp/features/Wenn-Weltraumpartikel-das-Smartphone-abstuerzen-lassen-3630560.html. Accessed: 2022-02-24.

List of Figures

List of Tables

A. Source Code

The source code of the `kernel_test` is given in the following:

```
#include <cyg/hal/hal_arch.h>
#include <cyg/kernel/kapi.h>
#include <cyg/infra/testcase.h>
#include <cyg/kernel/sema.hxx>

#include "cyg/kernel/fail.hxx"

static Cyg_Binary_Semaphore s0(false);

static cyg_mbox mbox;
static cyg_handle_t mbh;

#define NTHREADS 2
#define STACKSIZE CYGNUM_HAL_STACK_SIZE_MINIMUM

static cyg_handle_t handle[NTHREADS];

static cyg_thread thread_obj[NTHREADS];
static char stack[NTHREADS][STACKSIZE];

static cyg_counter counter_obj;
static cyg_handle_t counter;

static cyg_alarm alarm_obj;
static cyg_handle_t alarm;

// Outputs a character to Bochs's debug console.
void outportb(unsigned int port, unsigned char value)
{
  asm volatile ("outb %%al,%%dx": :"d" (port), "a" (value));
}

void putc(char c)
```

```
{
  outportb(0x3F8, c);
}

void print_data(char data[], unsigned len)
{
  unsigned i;
  for (i=0; i < len; i++) {
        outportb(0x3F8, data[i]);
  }
}

void swap(char *a, char *b)
{
  char tmp = *a;
  *a = *b;
  *b = tmp;
}

void
do_test_1(cyg_addrword_t data) {
  char input_data[] = {'4',')','-','m','c',':'};
  unsigned data_length = sizeof(input_data)/sizeof(*input_data);

  // do some work: sort input_data in place
  unsigned k, l;
  for (k=0; k < (data_length-1); k++) {
    for (l=0; l < (data_length-k-1); l++) {
          // swap elements, if need be
          if (input_data[l] > input_data[l+1]) {
                swap((input_data+l), (input_data+l+1));
          }
          char *message = new char;
          *message = input_data[l];
          cyg_mbox_put( mbh, (void *)message);
          s0.wait();
          cyg_counter_tick( counter );
        }
  }

  // print data and signal end to Fail*
  print_data(input_data, data_length);
```

```
  putc(s0.posted() + '0');

  putc((char) cyg_counter_current_value(counter));

  CYG_TEST_PASS_FINISH("Test OK");
}

void
do_test_2(cyg_addrword_t data) {
  while (1) {
        char *message;
        message = (char *)cyg_mbox_get(mbh);
        putc(*message);
        delete message;
        s0.post();
  }
}

void alarm_handler(cyg_handle_t alarmh, cyg_addrword_t data)
{
  putc('#');
}


externC void
cyg_start(void) {

  CYG_TEST_INIT();

  cyg_mbox_create( &mbh, &mbox );

  cyg_counter_create( &counter, &counter_obj );

  cyg_alarm_create( counter,
  alarm_handler,
  0,
  &alarm,
  &alarm_obj);
  cyg_alarm_initialize( alarm, 0, 4 );
  cyg_thread_create(3,do_test_1,0,"test_1",
  stack[0],sizeof(stack[0]),&handle[0],&thread_obj[0]);
```

```
  cyg_thread_create(4,do_test_2,0,"test_2",
  stack[1],sizeof(stack[1]),&handle[1],&thread_obj[1]);
  cyg_thread_resume(handle[0]);
  cyg_thread_resume(handle[1]);
  cyg_scheduler_start();
}
```