Optimization Techniques for Fault-Injection Experiments

Adrian Böckenkamp
Technische Universität Dortmund, Department of Computer Science 12
adrian.boeckenkamp@tu-dortmund.de

Abstract. This article summarizes existing work in the field of optimization concepts to decrease the amount of time required for fault injection (FI) campaigns. First, we present several concepts to improve the execution speed of FI experiments. They should be applied when constructing a FI framework. Second, we explain a few fault-space pruning techniques to reduce the overall amount of experiments that need to be carried out. Finally, an excerpt of the results of Hari et al. is presented to justify the effectiveness of fault space pruning [4].

Keywords: optimization, fault-injection, fault space pruning, bit-flip

1 Introduction

Future technologies steadily tend to reduce the chip size as well as the supply voltage because small computer chips with lowered energy consumption are required for all types of embedded systems. On the downside, this increases the probability of so-called soft errors. Examples for soft errors are transient bit-flips in processor registers. Such a bit-flip causes a bit to be toggled either from one to zero or vice versa. Although there is a huge interest in new improved hardware designs, dependability of (embedded) software is a major concern as well. This leads to the problem that on the one hand software developers need to forearm against such soft errors, but on the other hand still want to utilize the hardware efficiently.

Fault injection (FI) aims at inducing faults in selected parts of the hardware to identify the overall impact of such faults on the software running on the system—the so-called target application (or target for short). Faults that propagate through the software stack and eventually change the system behavior reveal weak spots of the software. With this knowledge, a software developer is able to implement fault-tolerance mechanisms to protect against undesired behavior. Furthermore, FI can be used to examine the effectiveness of fault-tolerance mechanisms which have been implemented on top of a given software.

This article presents recent techniques to improve the speed of FI experiments. They can be classified into two categories: The first approach tries to improve the underlying framework which is used to perform FI. The second approach focuses on the number of experiments that need to be carried out.
A few techniques have been invented to reduce the number of FI experiments while still preserving the ability to extrapolate the impact of faults for which no experiments have been executed.

Section 1.1 explains the terminology used throughout this article and Sect. 1.2 justifies the need for optimizations as mentioned above. Finally, Sect. 1.3 summarizes the content of this article.

1.1 Preliminaries

We proceed with the introduction of several basic terms. The fault model refers to the kinds of faults under consideration, i.e., it comprises of “where”, “when” and “what” to inject. An example has already been given in the introduction: Transient bit-flips in processor registers specify “where” a bit-flip (“what”) should be injected in (assuming one FI per experiment, i.e., “when”). In particular, it is necessary to detail what kinds of bit-flips are considered. In this article, we restrict ourselves to transient single bit-flips because more complex models lead to a fault-set size which is practically infeasible to cover. In addition, most of the research papers constructed their optimization techniques for rather “simple” fault models. In the context of this article, we only consider faults in processor registers and the main memory.

A given high level source code can be translated into a sequence of (assembler) instructions by invoking a compiler. Each of these instructions can be uniquely identified by its instruction address (also called the program counter). In the following, we call an assembler instruction the static instructions $I_s$. When executing the program, static instructions may be executed several times. We call the (multiple) executions of a corresponding static instruction, the set of dynamic instances $I_d$. For example, consider a for loop that iterates 100 times. The (static) instructions generated for the body of the loop will be executed 100 times\(^1\), yielding a 1:N-relation.

Many optimization techniques require a fault-free execution trace of the target application. This is also termed golden run. For applications that require a specific input, a predefined test input is used. Obviously, the golden run can be used to determine the dynamic instances $I_d$ by executing the target application in the absence of any FI and tracing the sequence of executed instructions. The elements of $I_d$ cannot be uniquely identified by their addresses due to the aforementioned 1:N-relation. Therefore it is convenient to number the elements in ascending order according to their occurrence in the trace.

We refer to the fault space $\mathcal{F}$ as the initial set of faults. The space is spanned by the dynamic instances $I_d$ (first dimension) and the locations in the unreliable hardware components (second dimension). The first dimension can be associated as “when” and the second dimension as “where” to inject a fault. If we think of a large application and a huge amount of memory (second dimension), the fault space becomes very large. To make this clear, consider the following example: Assume we have an application that computes the Fast Fourier Transform (FFT)

\(^1\)For simplicity, we assume the absence of compiler optimizations.
for a given test input with 548,000,000 dynamic instruction instances \( I_d \). Assume further, we only have nine 32-bit registers. In terms of our fault model, we need to perform about 158 billion FIs.

Fault effects can be classified in the following way. A fault in a register that is not used at all or gets overwritten immediately does not have any effect; such a fault is masked. On the other hand, a fault that corrupts a part of the application in a detectable manner is detected. The problem of software developers in the context of unreliable hardware stems from the third type of faults: silent data corruptions (SDCs). Faults resulting in SDCs change the overall behavior of the system which may cause a system to crash. It is important to note the inability of an application to recognize SDCs without any further information (e.g., output of the golden run). Section 3 describes a fault space pruning technique by exploiting the nature of detectable faults.

We have not yet explained how to measure the robustness of a software. An example is the total SDC rate [4]. Given a static instruction \( I_s \), we first compute the SDC rate for all dynamic instances \( I_d \) separately. Therefore, we count the number of FIs in all bits for all registers/memory cells used by \( I_d \) that result in SDCs. This count is accumulated over all dynamic instances of \( I_s \) and divided by the cardinality of the subset \( F_{I_s} \subset F \) which itself consists of all faults related to \( I_s \). Summing up the counts for all static instructions \( I_s \) divided by the size \(|F|\) of the initial fault set yields the total SDC rate for the application. The SDC rate should always be computable regardless of any optimizations applied (either to the FI framework or by pruning the initial fault set).

1.2 Problem

The example in the previous section already demonstrates the problem that arises in the context of FI: The fault space is much too large. In practice, 158 billions of FIs are clearly infeasible. Döbel et al., for example, performed FIs for about 170 million faults “to cover all possible register/bit/instruction-offset coordinates in [their] fault space, occupying [their] faculty’s computing cluster for several days” [3].

The numbers give rise to the need for optimizations, but where and how can optimizations be applied? The papers summarized in this article exploit two key ideas to improve FI experiments. At first, optimizations should speedup the experiment execution within the chosen FI framework itself. Second, pruning techniques should be exploited to reduce the number of experiments requiring further simulation. All optimizations should try to respect the primary objectives of fault injection, as mentioned in the introduction. Furthermore, the SDC rate should not be distorted by optimization.

1.3 Outline

In this section, we give a brief overview of this article. Section 2 briefly introduces the generic structure of a simulation-based FI framework and continues with possible concepts to optimize such a framework. Section 3 elaborates several
fault-space pruning techniques and their effect on the initial fault set. Afterwards, Sect. 4 evaluates effectiveness and accuracy of the presented pruning techniques. Finally, Sect. 5 briefly summarizes the core ideas and concludes with an outlook regarding further research topics.

2 Optimizations in FI Frameworks

This section deals with the general structure of fault injection frameworks and presents several optimization concepts to speedup the execution of experiments. We begin with some general classifications.

According to Benso et al., there are three different types of FI techniques \cite{1}: hardware-based, software-implemented and simulation-based. Hardware-based FI requires injecting faults directly into the hardware units, e.g., with the assistance of a hardware debugger. When referring to software-implemented FI, the FI logic is implemented in the target application itself. At last, simulation-based FI denotes the simulation and observation of the target application with the help of a simulator. A simple example for this technique is the use of a debugger which executes the target application and can be instructed to change the registers of the running target with debugger commands (e.g., for \texttt{gdb: set $eax=66}). In the following, we will only be concerned with \textit{simulation-based techniques}.

In the previous section, we already introduced the (hardware) locations of FI, i.e., “where” to inject a fault. We distinguish between three abstraction levels \cite{4}: gate, microarchitectural, and architectural level. The gate level considers (low level) gates—such as \texttt{AND}, \texttt{OR} or \texttt{NOT} gates—for fault injection. The microarchitectural level, for example, considers the components of the processor design (e.g., the arithmetic logical unit) for FI. The architectural level considers processor registers and memory cells for FI. Generally speaking, the lower the abstraction level, the worse is the overall simulation speed. Here, we will only focus on the \textit{architectural level}.

2.1 Generic Structure

Fault-injection frameworks are the base for executing FI experiments in a simulation-based environment. They are typically composed of a set of inputs (target binary, predefined test input and additional constraints), optimization algorithms to be applied on the initial fault set, a storage, the simulator and some post processing. Figure 1 illustrates these components and their interaction. At first, the target is executed without any FI (golden run) to create the initial set of faults. This set is either written to a database or directly forwarded to a pruning algorithm (see Sect. 3). The reduced fault set is then used for the simulations. With a subsequent analysis of the results, further statistics (e.g., the SDC rate) can be computed which may take into account the output of the pruning algorithm (see dashed line in Fig. 1). \textit{Experiments} and the superior \textit{campaign} \cite{2} form the major parts of such a framework. The experiments define

\footnotesize
\begin{itemize}
  \item Both are not explicitly shown in Fig. 1.
\end{itemize}
how the FI takes place and the campaign is responsible for supplying the experiments with appropriate parameter sets. The experiments can be part of the simulator or defined separately. The campaign usually has access to the database and initiates the experiment executions. It depends on an input fault set which may be pruned by applying some pruning techniques.

Two examples of FI frameworks are Fail* [5] and Relyzer [4]. The Fail* framework “provides carefully-chosen abstractions simplifying both the implementation of different simulator/hardware target backends and the reuse of experiment code, while retaining the ability for deep target-state access for specialized FI experiments” [5]. Its structure is similar to the one sketched in Fig. 1 except for the storage component which can be chosen freely by the experiment developer.

Relyzer is composed of a “full system simulation environment” (Wind River Simics) and a “microarchitectural and memory timing simulator” (GEMS) [4]. The simulation environment enables the simulation in functional mode which improves the overall execution speed. However, detailed FI requires a precise timing simulation which is provided by the additional timing simulator (but at lower speed). The timing simulator can be switched on and off as needed. During the simulation, a symptom detector checks for detectable faults (e.g., fatal traps or kernel panics).

Both frameworks allow the injection of faults at arbitrary (dynamic instances of) static instructions. Relyzer mainly aims at injecting faults in integer registers and address output latches while Fail* is designed to be much more versatile regarding FI locations.
2.2 Speed Improvements

In this section, we discuss several ideas to improve the execution speed of experiments. Note that this does not include the idea of reducing the amount of experiments which is discussed in the next section (cf. Sect. 3).

Experiment Parallelization. The first idea is to use parallel—and wherever possible distributed—execution of experiments [3–5]. This idea stems from the observation that FI is highly parallelizable. It is therefore very obvious to execute multiple experiments at the same time. Exploiting the potential availability of a computing cluster leads to an increased throughput of experiments as well. As mentioned previously, the campaign’s job is to distribute the parameters to the experiments required for FI. After all FIs have finished, the results have to be joined locally. Both of the aforementioned examples use this idea in an intensive way.

Simulator Checkpoints. A time-consuming phase in simulation-based FI frameworks is to bring the simulator to the instruction of interest, i.e., to wait until the simulation reaches the dynamic instruction $I_d$ where we want to inject a specific fault. To improve this, we can exploit a simulator’s capability of saving the current simulator state—the so-called checkpoints $C$—and restoring the state when desired. This allows us to skip the tedious simulation phase prior to the instruction of interest by simply loading the state of the golden run prior to $I_d$. However, saving the simulator state takes a huge amount of space on the hard disk which has, in turn, rather slow access rates. We therefore have to trade off the number of checkpoints against the storage footprint. A possible approach is to generate many equidistant checkpoints [2]. Figure 2 shows the idea with a small example. Let $f_i$ denote the fault that we want to inject at some time $t$ (shown as a red dot). The green arrows indicate the equidistant checkpoints $C$ that were created by using the golden run. Without using the checkpoints, the simulator needs to simulate all instructions from zero to $t$. In contrast to this, by using the checkpoints, we only need to load the latest checkpoint $c \in C$ at time $t'$ with $t' < t$ and simulate the remaining instructions up to $t$ (i.e., $t - t'$ instructions). The cardinality of $C$ is proportional to the required storage footprint and inversely proportional to the size $s$ of the checkpoint intervals (cf. Fig. 2). If, for example, the dynamic instruction instance $t$ uses two 32-bit registers and we have to inject faults in each bit, we can save the previously mentioned simulation time for all 64 FIs. Clearly, this optimization concept requires a “checkpoint capability” of the simulator being used.

Experiment Termination. Another point to consider is the question of when an experiment is allowed to terminate at all. In principle, each experiment must be given the possibility to run to completion because an injected fault can still affect the system behavior at any later point in time. Unfortunately, this would
take much to long. Consequently, a premature termination of experiment executions is desired [2, 4]. In contrast to the exact solution (running to completion), there are two heuristical approaches [2, 4]. The first approach continues the simulation after a FI at some time \( t \) for \( n \) instructions where \( n \) is a heuristical parameter (e.g., \( n = 500 \)). If a fault affects the system behavior after \( t + n \), it is not recognized by the framework. The second approach is based on the observation that most of the faults either affect the system immediately after the FI or after a relatively long period of time. It is therefore convenient to compare the output of the target application with the golden run in exponential steps. Thus, the first comparison is made directly after the injection, the second comparison is made after two instructions, and so forth. It should be noted that this approach was originally used at the gate level but could be incorporated on a higher abstraction level, too.

Code Modifications. The last optimization concept refers to the idea of modifying the simulator code [5]. The modifications at least contain the FI and observation logic for the experiments. Obviously, this idea requires the simulator’s source code to be available. Due to the fact that the modifications are directly implemented in the simulator’s source code, this concept typically offers a good performance. However, the changes may also lead to unreadable code because they scatter across the simulator code. If typical imperative programming languages like C++ or Java are being used, a possible solution could be the use of aspect-oriented programming (AOP). AOP enables one to modularize the so-called cross-cutting concerns. In the context of this optimization concept, the cross-cutting concerns are all code snippets that need to be incorporated at different positions of the simulator source code. Thus, AOP allows us to encapsulate the required modifications in separate modules (the so-called aspects) which are automatically woven into the simulator code. As a consequence, no explicit changes are required. The FAll* framework, as briefly introduced in Sect. 2.1, uses AspectC++ (an AOP extension for C++) to cope with the problem of cross-cutting concerns.

In conclusion, it can be said that many of the above-mentioned concepts are featured in current FI frameworks to make the simulation of a huge amount of experiments practically feasible. However, if we only build on the optimization concepts presented in this section, the initial fault sets of many target applica-
tions are still much too large. This is roughly justified as follows: The formula

\[
\frac{\text{time(experiment)} \cdot \#\text{experiments}}{\#\text{machines}}
\]

approximately computes the run time for a FI campaign containing \#experiments. They are executed on \#machines (in parallel) and the running time of each experiment is \text{time(experiment)}. According to Sect. 1, \#experiments can be of the order of 158 billion. If, e.g., the checkpoint approach reduces the execution time of an experiment by a factor of two, then the resulting run time for the FI campaign is still too large. Hence we need more effective optimizations that address the size of the initial fault set (cf. Fig. 1, “fault space pruning algorithm”). The next section introduces such optimizations; they are usually termed fault-space pruning techniques.

3 Fault-Space Pruning Techniques

Fault space pruning deals with the reduction of the initial given faults that still need further simulation to estimate their outcomes. It is important to note that fault space pruning does not decrease the actual number of faults; instead, it decreases the number of experiments.

We classify pruning techniques into two major categories [4]: Known-outcome-based techniques are able to predict the effects of a fault without requiring any simulation. They are usually implemented using static code analysis. Equivalence-based techniques divide the total set of faults into equivalence classes requiring only one FI for a representative of each class. Equivalence-based techniques are further subdivided into exact and heuristical techniques. The classification is motivated by the definition of equivalence: Exact equivalence-based techniques use precise definitions for the equivalence of faults whereas heuristically equivalence-based techniques define the equivalence upon empirical observations. The core ideas of pruning techniques can be summarized as follows:

1. Do not simulate any faults if it is possible to predict their outcomes without simulation.
2. Try to find as many equivalent faults as possible and limit the simulation to their representatives.

It should be noted that the following techniques are more or less independent of the underlying architecture although some may provide more beneficial results than others.

3.1 Random Downsampling

This simple pruning technique samples the fault space (randomly) using an uniform probability distribution [1]. By varying the distribution density, the reduction factor (of the initial fault set) can be adjusted as needed. For all elements
of the resulting subset of faults, a FI is performed. Figure 3 visualizes the fault space (gray) and an exemplary distribution (red). The vertical axis denotes the memory addresses in bytes and the horizontal axis denotes the dynamic instances of the static instructions (retrieved from the golden run). To cover the entire (initial) fault space, we would inject faults at any combination of instructions and memory addresses (gray). By applying this pruning technique, this can be reduced to the points shown in red (each point represents a FI). Note that instructions are ignored if they do not access the memory at all [1].

Fig. 3. Example for a given fault space (gray) and a subset of faults (red) retrieved by random downsampling. The red points visualize the (uniform) distribution that was used to sample the fault space.

Clearly, this does not cover all possible faults and we are therefore not aware of the effects of faults that have not been selected. As a consequence, we may miss some weaknesses of the target application—a major drawback of this approach. Figure 3 sketches this problem with the highlighted yellow point, indicating a potentially overlooked fault whose outcome is unknown. In addition, it is non-deterministic due to the underlying probability distribution. An important advantage of this technique is its scalability. Furthermore it is fast and easy to implement. Regarding the introductory classification, random downsampling is a heuristically, equivalence-based technique because it assumes that the sampled subset of faults sufficiently represent the entire fault set.

3.2 Address Bounding

This approach exploits the nature of detectable faults as mentioned in Sect. 1 [1,4]. Consider a single bit-flip in a memory address stored in a register. The bit-flip can have exactly two outcomes. First, the memory address becomes invalid. When the system tries to access the invalid address, it enters an exceptional

The technique is applicable on registers as well.
state (e.g., throws a segmentation fault). In fact, the system is able to detect the fault even though it may not be able to recover a working state. Second, the bit-flip changes the register value to a valid address which involves a valid memory access, too. The first case allows us to prune exactly those faults that result in detectable system behavior whereas the second case requires further FI.

The technique proceeds as follows: A dynamic profiling of the golden run traces all memory accesses to determine the valid address range $B$. More specifically, $B$ contains all addresses pointing to the heap and the stack. With the help of this information, we declare all faults as detectable if they produce a new address not contained in $B$. Address bounding is applicable to all Load and Store instructions.

The benefit of this technique is obvious: We do not need to perform FI for all those faults which have been declared as detectable. Consequently, it is a known-outcome-based approach. The technique may be extended to jump instructions whereby faults producing invalid instruction addresses are pruned away. However, this involves the problem of distinguishing between calls to the current .text section and calls to shared libraries. The latter distinction might be difficult on some platforms [4].

### 3.3 Def-Use Analysis

Def-use analysis only considers the subset $F_r \subset \mathcal{F}$ of dynamic instruction instances$^4$ that access a register $r$. We look at an instruction that writes into a specific register $r$ at some time $t \in F_r$ (def) and another succeeding instruction that reads the value of $r$ (use) at some later point $t' \in F_r$ (with $t' > t$). The following two instructions explain this scenario through an example (with $r = r_1$):

$t: \quad [r_1] = r_2 + r_3; \quad \quad t': \quad [r_4] = [r_1] + r_5;$

The first instruction writes the sum of $r_2$ and $r_3$ to register $r_1$ (write access, red). The second instruction reads the value of $r_1$ to compute the sum of $r_5$ and $r_1$ (read access, green). The idea of this technique is to prune the faults in the write accesses of a register because their effects are equivalent to faults in the read accesses. Thus, the faults at instruction $t \in F_r$ for register $r_1$ can be pruned because they are equivalent to faults in the read access of $r_1$ at $t' \in F_r$.

More generally, consider an instruction that reads a specific register $r$ at time $t \in F_r$. In the context of FI, a previous write access to $r$ (at time $t' \in F_r$ with $t' > t$) is assumed to be equivalent to the subsequent read access of $r$. Additionally, consider an instruction that writes into a specific register $r$ at time $t$. Again, when going backwards in the instruction trace, all write accesses to $r$ can be ignored until the occurrence of a read access to $r$. The latter case can be categorized as known-outcome-based while the former case is equivalence-based.

Figure 4(a) shows an example for an access pattern on a hypothetical platform with three registers $r_1, r_2$ and $r_3$. The $x$-axis denotes the (dynamic instances

---

$^4$In principle, this approach is applicable to memory cells as well.
of) static instructions and the y-axis denotes the FI locations. A green “R” represents a read access to the given register and time whereas a red “W” represents a write access. Figure 4(b) shows the pruned result. Black crosses encircled in a gray rectangle indicate faults in write accesses that have been pruned due to equivalent faults in the read access encircled in the same rectangle. This equals the first case, as discussed above. The write access at \( t_4 \) to register \( r_2 \) has been pruned due to the second case, i.e., we begin at instruction \( t_6 \) and prune away all faults in write accesses to \( r_2 \) until the occurrence of a read access (to \( r_2 \)). This way, we stop at \( t_1 \) and prune faults in the write access at \( t_4 \).

![Fig. 4. Example for fault space pruning using def-use analysis on a hypothetical 3-register machine.](image)

Def-use analysis needs to be employed on each register of the target architecture whereby each iteration considers a different subset \( F_r \). The use of this technique allows us to divide many of the faults into equivalence classes or ignoring them completely. Clearly, faults that can be ignored do not require FI. As mentioned in the introduction, one representative of each equivalence class is taken for FI which reduces the number of experiments as well (compared to the initial set). This technique can be classified as (exact) equivalence-based because the first case (see above) is the crucial part of the approach.

### 3.4 Control Flow Equivalence

This technique is much more sophisticated and based on the following observations [4]:

1. Similar faults propagate through similar code sequences.
2. Multiple executions of a code sequence increase the probability of faults.

Keeping these observations in mind, let us consider a static instruction \( I_s \) with many dynamic instances \( I_d \). The idea of this technique is similar to the one in Sect. 3.3. We divide the dynamic instances into equivalence classes and only perform FI for their representatives. The sophisticated part is the (heuristical) definition of equivalence which uses the observations (1) and (2). We enumerate all control flow paths \( P \) up to a length of \( n \) using dynamic code analysis of the golden run. For each dynamic instance \( I_d \), the enumeration for a path \( p \in P \)
starts at the basic block containing $I_d$. The path $p$ consists of basic blocks along the actual control flow in the golden run whereas the maximum length $n$ is a heuristical parameter (e.g., $n = 5$). Note that there are several paths for a single dynamic instance. Figure 5 shows an example for a control flow graph (CFG) on the left and the set $P$ of possible control flow paths on the right. Each node represents a basic block and the edges determine the possible control flows. Basic block four is never visited according to the golden run. The paths start at basic block one (containing our dynamic instance $I_d$ of interest) and each of them represents an equivalence class for FI at $I_d$.

Fig. 5. Left: exemplary control flow graph (CFG) on the basic block level. Right: list of all possible paths within the CFG up to length 5 (based on [4]).

With the help of the dynamic application profile, we can determine the number of executions of each path. We call the set of executions of a path $p_i$ its population $Q_i$. Actually, the semantics of a population equals the semantics of an equivalence class. For example, Fig. 5 has five populations depicted on the right. We randomly select a so-called pilot $p_i^*$ for each population $Q_i$. Again, the semantics of a pilot equals the semantics of a representative of an equivalence class. Finally, the FI is only performed on the pilot paths whereas all other paths $Q_i \setminus \{p_i^*\}$ are discarded in terms of FI. Thus, the pilots represent the paths of their population regarding their FI results. If the first observation is not or only partially satisfied, the technique becomes inaccurate.

An exemplary reason for a huge population $Q_i$ could be a for loop that iterates many times. The paths in $Q_i$ are composed of the basic blocks for the body of the loop. The outcome of a FI in the pilot of $Q_i$ is assumed to be the outcome of FIs in all paths of $Q_i$ as well. However, the promising advantage is the reduced number of FIs. Now, recall the SDC rate explained in Sect. 1. In order to correctly compute this rate when applying this pruning technique, we need to weight the number of pilots resulting in SDCs with the cardinality of their population.

In a nutshell, the approach exploits the first observation to define a heuristic for equivalent control flow paths and the second observation to be still able to
compute the SDC rate. It is obviously a heuristically equivalence-based tech-
nique. Furthermore it should be noted that the technique is applicable to all
instructions except for Load and Store instructions and those who depend on
a Load or Store. This is because the fault propagation of these instructions
depends not only on the control flow paths but also on the addresses of the Load
instructions. The described version does not cover this issue. Hari et al. also de-
scribe an extension to this technique, the store equivalence, which covers control
flows consisting of Load and Store instructions [4].

In conclusion, it can be said that pruning techniques are the most promising
optimizations to make larger FI campaigns practically feasible. The next section
treats the effectiveness and the accuracy of such (heuristical) pruning techniques.

4 Appraisal and Discussion

This section presents evaluation results about the fault-space pruning techniques
of the previous section.

First of all, we describe the data basis that was used for evaluation [4]. The
target platform is SPARC v9—a RISC architecture. The authors selected twelve
applications from three different benchmark suites. The applications arise from
different areas such as physics, computer science and mathematics. All of them
have been simulated using the FI infrastructure briefly described in Sect. 2.1.

The number of dynamic instances for the twelve applications vary from 22.3
million to 4.57 billion for predefined test inputs (shipped with the benchmark
suites). Hence the underlying target binaries are relatively large. It is therefore
not very surprising that the number of total faults $|F|$ varies from 1.9 to 500.4
billion. For example, the 500.4 billion faults occurred for the gcc compiler which
was instructed to compile some predefined test code for an AMD Opteron pro-
cessor. Some other exemplary applications are an implementation of the Fast
Fourier Transformation (FFT) algorithm, the Ocean simulator to simulate wa-
ter currents and Water, an evaluation platform for the interaction of forces and
potentials in a system of water molecules.

The results of Benso et al. do not have been considered in this section because
they are less representative compared to those of Hari et al.

4.1 Effectiveness of Pruning Techniques

We begin with the treatment of the overall pruning effectiveness. The total prun-
ing amount averaged over all techniques and applications is 99.78 %, i.e., 99.78 %
of all faults could be pruned away. This corresponds to a decrease by three to
six orders of magnitude. Figure 6 shows the percentage distribution of the total
amount. The most impressive technique is control flow equivalence (see Sect. 3.4)
with 48 %. Surprisingly, address bounding—a rather simple technique—has a
share of 27 %. A very important result is that 99 % of all faults are represented

\[\text{Random downsampling (cf. Sect. 3.1) is not included at all.}\]
by only 0.004 %. This reveals a high potential for optimizations in terms of pruning techniques.

The results of Hari et al. offer another insight: Compiler optimizations have a non-neglectable impact on the effectiveness of pruning (and the initial fault set). The following example demonstrates the effects on the pruning efficiency using the results of the FFT application. In the optimized version, the initial number of faults was 48.7 billion whereas the pruning decreases this value to 300,000. Moreover, in the optimized version, the initial fault set was 61.18 billion. With the application of all pruning techniques, this degrades to only 160,000 residual faults. Thus, the reduction rates for unoptimized target binaries are (much) higher than for optimized binaries. It should be noted, that the values and observations are emerged from further pruning techniques that were only applied in case of the unoptimized target. They have not been described here.

4.2 Accuracy of Heuristics

This section appraises the accuracy of the heuristical pruning techniques (see Sect. 3.4). First of all, we need to explain the model that was used to measure the accuracy. We do this by giving an example: Let us assume the outcome of a FI for a given pilot $p^\star$ is a masking effect. Suppose that we are able to perform FI for all elements of the population $Q$ of $p^\star$. Assume further, 98 % of the faults are masked and only 2 % result in silent data corruptions (cf. Sect. 1.1). Then we say the accuracy for population $Q$ in this example is 98 %. The total accuracy for a pruning technique is computed by the weighted average of all pilots whereby the weights are determined by the cardinalities of the populations.

In fact, the previous example is not feasible in practice because the FI for all elements of $Q$ would take much too long. Therefore, the following heuristics is used: Each population is sampled randomly and only each eighth bit of the registers is taken into account. In addition, the number of pilots are truncated to perform approximately one million FI experiments “such that it [is] feasible to simulate all of them [...] in the available time” [4].

By this means, the averaged accuracy for control flow and store equivalence is 96 % (across all twelve optimized applications). Note that this also includes address bounding (cf. Sect. 3.2) and def-use analysis (cf. Sect. 3.3) which, in turn, are assumed to have 100 % accuracy. In particular, control flow equivalence achieves 95.7 % accuracy for FI in integer registers and 94.5 % accuracy for FI in address output registers. The accuracy for each application—averaged across

![Fig. 6. Percentage distribution of pruning efficiency regarding different (kinds of) pruning techniques for the optimized versions of the twelve applications (from [4]).](image-url)
all pruning techniques—is always higher than 91%. A major weakness of control flow equivalence is its restriction on the basic block level. This is a good starting point for further improvements.

5 Summary

In conclusion, the optimization purposes of this article can be summarized as follows. On the one hand, Sect. 2 introduced optimizations to increase the throughput of FI campaigns, i.e., to accelerate experiment executions. These optimizations need to be incorporated when implementing a FI framework. On the other hand, Sect. 3 presented several pruning techniques to reduce the total number of experiments that need to be executed.

The core ideas of fault space pruning can be recapitulated in the following way: First, try to predict the outcome of a fault to obviate its FI experiment. Second, divide the faults into populations which only involve the FI for the pilots.

The results of Sect. 4 approve the effectiveness of the approaches—especially the total reduction rate of 99.78% is very impressive. Control flow equivalence makes up the largest part of this reduction (48%). It is important to note that the pruning effectiveness crucially depends on the used compiler, target application and necessarily on the pruning technique itself.

In summary it can be said that pruning techniques and efficient FI frameworks are beneficial and strongly required which also strengthens the need for further research activities.

References