FASTFI: Accelerating Software Fault Injections

Oliver Schwahn, Nicolas Coppik, Stefan Winter, Neeraj Suri
Department of Computer Science
Technische Universität Darmstadt
64289 Darmstadt, Germany
Email: {os, nc, sw, suri}@cs.tu-darmstadt.de

***Abstract*—Software Fault Injection (SFI) is a widely used technique to experimentally assess the dependability of software systems. To provide a comprehensive view on the dependability of a software under test, SFI typically requires large numbers of experiments, which leads to long test latencies. In order to reduce the overall test duration for SFI, we propose FASTFI, which (1) avoids redundant executions of common path prefixes for faults in the same injection location, (2) avoids test executions for faults that do not get activated, and (3) utilizes parallel processors by executing SFI tests concurrently. FASTFI takes patch files that specify source code mutations as an input, conducts an automated source code analysis to identify the function they target, and then automatically parallelizes the execution of all mutants that target the same function. Our evaluation of FASTFI on four PARSEC benchmarks shows a SFI test latency reduction of up to a factor of 26.**

***Index Terms*—Software fault injection; Parallelization; Software testing; Efficiency; Dependability assessment**

1. Introduction

Modern software stacks are increasingly complex, due to the increasingly sophisticated application scenarios they are used in. To cope with this increase in complexity, many software projects re-use existing so-called "off-the-shelf" software components. While software re-use is cost-effective, it can pose a risk for system reliability, as even correct software can malfunction if it is used in a different operational environment than originally anticipated[1]. To test whether software faults in some part of the software stack are critical to its overall reliability, software fault injection (SFI) [2]–[4] is a widely used method.

SFI creates a number of faulty software versions, executes them, and monitors their effects on the execution environment. How SFI generates faults is commonly specified in terms of code patterns that are referred to as fault models (e.g., [5]–[8]) or mutation operators (mostly in the mutation testing community, e.g., [9]–[11]). As these patterns can be applied more often for larger code bases, more complex software yields higher numbers of faulty versions and higher numbers of faulty versions result in longer SFI test latencies.

The most common approach to deal with the increasing test complexity is downsampling, i.e., a reduction of the tests to execute based on some heuristic or random sampling. Such a reduction is obviously unsound, as it may miss relevant (i.e., failing) tests. We propose FASTFI as an alternative solution that preserves soundness and reduces SFI test latencies by

1) avoiding redundant re-executions of code paths shared between different tests,
2) reducing the number of tests with non-activated faults, and
3) exploiting the performance speed-up potential of modern parallel processors.

FASTFI achieves this by analyzing the faults that are created by an existing fault injection tool, grouping them per injection location, and generating a library of all faulty versions that allows to integrate all faulty versions into one single executable. The parallel execution of the different faulty versions is managed *on demand* at run time by control logic that is additionally inserted. Applying FASTFI to four benchmarks from the PARSEC suite [12], we achieve an SFI execution time reduction of up to a factor of 26, which corresponds to an execution time reduction of over 6 hours for the respective benchmark. In addition to the speed-up in test execution, FASTFI also reduces the required build time to generate all faulty versions by a factor up to about 14.

The remainder of the paper is structured as follows: We discuss related work in Section 2. Section 3 introduces FASTFI and we present our evaluation of the approach in Section 4. Section 5 concludes the paper.

2. Related Work

FASTFI achieves higher FI test throughput (1) by test parallelization, (2) by avoiding redundant code execution.

2.1. Work on FI Test Throughput

A number of studies have advocated the potential benefits of parallelizing FI experiments [13]–[16] using virtual machines [13], [14] or OS processes [16] to isolate the experiments. Although virtual machines provide execution

1. For instance, the inertial reference system that was safe for the Ariane 4 launcher turned out to be unsafe for Ariane 5, which exhibited a higher horizontal acceleration during the first 40 seconds after lift-off [1].

environments with stronger isolation, the run time overheads that virtual machines incur can cause performance interferences, which can equally distort the results of fault injection experiments [17]. As a consequence, we chose to restrict FastFI's isolation for concurrently executing experiments to lightweight processes.

2.2. Work on Test Parallelization

Until recently, many approaches parallelized test executions under the assumption that these tests are independent and do not influence each other [18]–[21]. This assumption has proven incorrect for a number of test suites [22], [23]. Newer approaches take possible test dependencies into consideration and use this information to determine which tests need to execute in sequence to prevent spurious results [24]–[26]. In FastFI, concurrently executing program versions do not interfere as external resources are carefully handled by the runtime. As soon as a faulty version is selected for execution, a new process is forked to guarantee memory protection via address space isolation. Possible interference on shared persistent file storage are prevented by means of I/O redirection. Thus, the isolation across parallel SFI tests is stronger than what is commonly assumed for parallel correctness tests, but weaker than the state-of-the-art in parallel fault injections to reduce the risk of performance interference documented in [17].

2.3. Work on Avoiding Redundant Code Execution

FastFI saves execution time by avoiding redundant and unnecessary code executions, as we will detail in Section 3.2.2. We are only aware of one work that makes a similar attempt to reduce test suite execution latency. VmVm [27] analyzes which data is modified by each individual test case in a test suite and makes sure that the test suite executor only resets that part of the system state between tests, so that heavier isolation mechanisms can be avoided. The authors report an average execution time reduction of 62 %. In contrast to VmVm, FastFI avoids (a) the execution of code paths that are redundant for many tests and (b) the execution of faulty program versions, for which the fault would not get activated. These redundancies are peculiarities of FI tests and usually do not apply for other types of tests, such as unit tests targeted by VmVm. FastFI also does not attempt to reduce isolation between tests, but utilizes this isolation to safely execute tests concurrently to gain additional speed-up from parallel hardware.

3. FastFI

FastFI reduces the execution latencies for large SFI test suites that are commonly required for the robustness assessment of complex software. FastFI achieves the reduced execution latencies by (1) not re-executing redundant code paths, (2) reducing the number of tests without fault activation, and (3) parallel execution of tests.

Section 3.1 gives an overview of FastFI and its workflow while the following Section 3.2 details the FastFI execution model. Section 3.3 provides a detailed discussion of the employed parallelization strategy and the required control logic. Sections 3.4 and 3.5 discuss the required static analysis and technical limitations of the approach. Section 3.6 provides a brief overview of our prototype implementation.

3.1. Overview

For SFI tests, faulty versions of a given software are generated, which are then executed separately and the outcome of their execution is monitored. These test executions typically require external experiment logic for controlling which faulty versions get executed and for monitoring test outcomes. Typical test outcomes include successful execution, execution with error indication, and aborted (crashed) execution. Note that each faulty version typically contains only one single fault to allow for the isolated observation of the fault's effects.

With FastFI, the generated faulty versions of the software are not built and executed separately, but they are integrated into one test executable. For that purpose, FastFI groups the faulty program versions by the functions in which the faults are injected. Each fault is then included in the program as a faulty version of the function that it modifies, rather than creating faulty versions of the complete program as in existing SFI tools. Although the fault grouping granularity could be changed from the function level to, e.g., basic block or even statement level, function-based grouping appears to be the natural choice for procedural languages and proves effective in our evaluation (see Section 4). The FastFI runtime controls *on demand* which of the integrated faulty versions are executed once the test execution reaches a point where a faulty function version can be selected for execution. The executions of the different faulty versions are isolated from each other by forking a new process for each faulty version. The FastFI runtime includes all control and monitor logic needed to conduct SFI tests, i.e., no external logic is required to conduct a full set of tests with all generated faulty software versions.

Figure 1 provides an overview of the FastFI tool chain that generates a FastFI-enabled test executable for a given software. The FastFI tool chain takes the original source code and the code mutation patches from an SFI tool as input. We use SAFE [8] as SFI tool in our evaluation, but FastFI is independent of the actual SFI tool used. The only constraints are that the code patches generated by the SFI tool modify only one source code function at a time, as FastFI groups faults on a per-function level, and that the patch files adhere to the commonly used (unified) diff format as understood by the GNU `patch`[2] tool. The FastFI tool then performs the following steps on the provided inputs:

1) Static source code analysis for function extraction and fault grouping
2) Generation of a code library with all faulty function versions

2. http://savannah.gnu.org/projects/patch

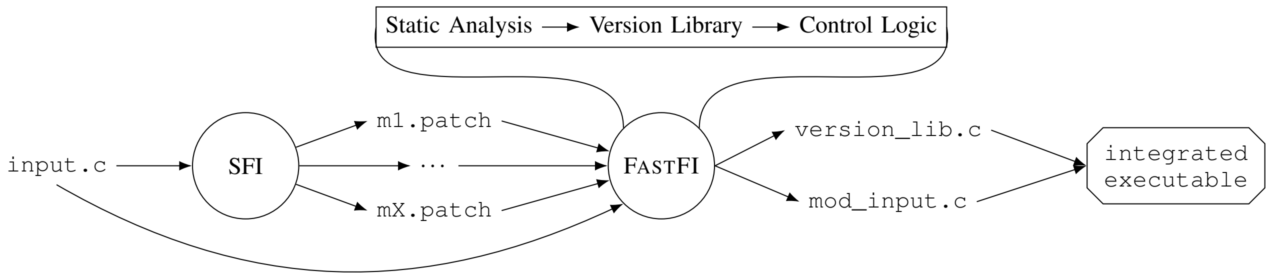

Figure 1. Overview of the FASTFI workflow. The input is the original source code and SFI mutation patches. The final output is a FASTFI-enabled integrated executable.

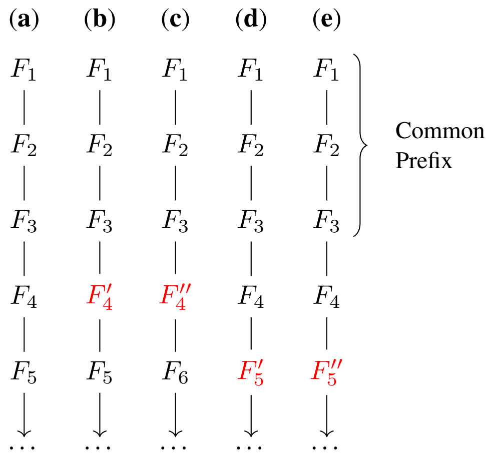

Figure 2. Traditional Execution Model. F_i denote functions and F_i' denote faulty versions of a function.

3) Insertion of the FASTFI fork server control logic into the original functions

The output is a modified version of the original source code with the FASTFI fork server control logic inserted and a library of all faulty function versions as well as copies of the original, unmodified functions. The final output after the usual software build process is the integrated FASTFI-enabled test executable.

3.2. FASTFI Execution Model

FASTFI introduces a novel, more efficient execution model for SFI tests that is enabled by the integration of all faulty software versions into one integrated executable. Both the traditional and the novel FASTFI model are discussed and contrasted in the following.

3.2.1. Traditional Execution. In the traditional execution model for SFI tests, each faulty software version is an executable of its own that has to be compiled and executed separately. Figure 2 illustrates an example for the execution of 5 tests in the form of function-level execution traces. The F_i are the functions executed. Faulty versions are marked with prime symbols, e.g., F_4' denotes a faulty version of function 4 and F_4'' denotes another faulty version of the same function. Trace (a) represents the execution of the original, fault-free software whereas traces (b) and (c) represent executions with faulty versions of F_4 and traces (d) and (e) with faulty versions of F_5. Each execution trace can contain only one faulty version of any function. However, the same faulty version of a function can obviously be invoked more than once during an execution. All the different traces share a common execution prefix up to the point where a faulty function is invoked for the first time. In the illustrated example, F_1 to F_3 is the common execution prefix for all 5 traces. For traces (a), (d), and (e), the common prefix is F_1 to F_4 as the first invocation of a faulty function happens later in the execution. After the invocation of a faulty function, the different executions may deviate drastically depending on the injected fault type and on whether it is activated during execution of the faulty function, as faults may arbitrarily change the program state. For instance in trace (c), F_6 is invoked after the fault in F_4'' was activated instead of F_5 as in the fault-free execution (a) or after execution of the faulty function F_4' in (b). Hence, although there is a common execution prefix between tests, there is generally no common postfix once a fault has been activated. However, re-executing the common prefix for each individual test is time-consuming redundant work that can be avoided using FASTFI as detailed in the following section.

3.2.2. FASTFI Execution. The essential difference to the traditional execution model is that FASTFI does not re-execute the common execution prefixes for all faulty versions and selects the faulty versions to be executed *on demand* during runtime. The FASTFI execution model is enabled by the integration of all faulty versions into one executable. Figure 3 illustrates an example for the execution of 7 tests as function-level execution traces similar to the illustration for the traditional model in Figure 2. The common prefix F_1 to F_3 is only executed once in this model by the master process, which is represented by the leftmost execution trace. The master process controls the execution of the faulty versions but never executes a faulty version of any function itself. Instead the master process creates, i.e., forks, (illustrated by dashed arrows) new child processes that execute the faulty versions on its behalf. Each faulty version is executed in its own process. In the example, once the master execution

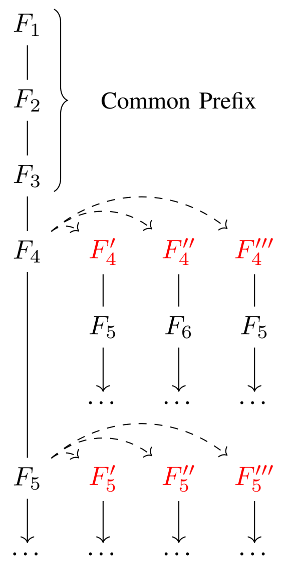

Figure 3. FASTFI Execution Model. F_i denote functions and F'_i denote faulty versions of a function. Dashed arrows represent process forks.

reaches F_4, for which three faulty versions exist, the FASTFI runtime forks a new process before the faulty version F'_4 is invoked. Since the `fork` system call creates an exact copy of the calling process, the new process starts its execution right were the master process called `fork` and invokes F'_4. Because F'_4 now executes in its own process, it cannot interfere with the execution of the master process. Both processes are isolated from each other by means of operating system process isolation, which, for instance, guarantees memory isolation such that the fault executing child process cannot write to the master process' memory. However, there is still the possibility of interference via external resources that are not covered by OS process isolation such as the shared file system. Therefore, the FASTFI runtime does not only perform a `fork` but also takes actions to minimize the chance for interferences via open files by file descriptor manipulation and I/O redirection upon forking.

Once the child process that executes the faulty version has finished, the master process either continues with the execution of the next faulty version, if available, or proceeds with its own fault-free execution. In the example, the master continues with the execution of F''_4 and afterwards F'''_4 in their own processes and then proceeds in its own execution by invoking the original F_4. When the master process eventually finishes, the master's execution corresponds to the execution of a fault-free software version and all faulty versions that were reachable have been executed. Since software functions that are not reachable during execution, for instance, if the provided program input does not trigger all of them, are never executed by the master process, they do not impose additional test latencies. This is an improvement over the traditional execution model, because it is generally not known a priori which faulty versions are reachable during execution. Hence, FASTFI automatically reduces the amount of faulty

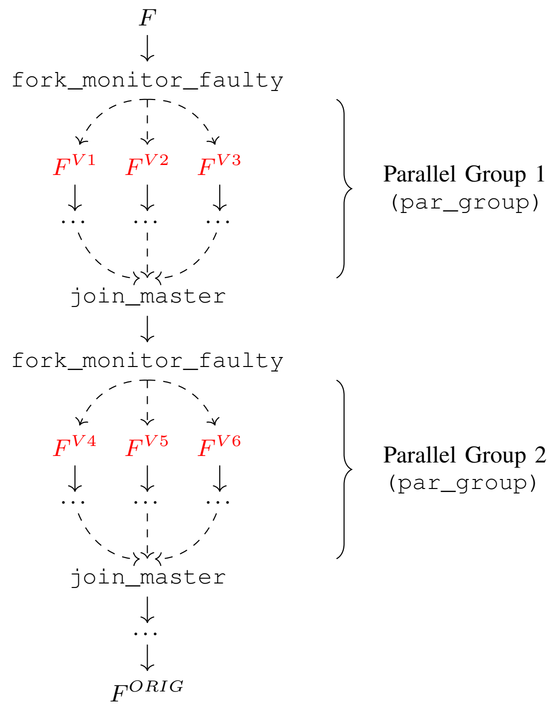

Figure 4. FASTFI Parallel Execution. The example illustrates the execution of all versions of function F using parallelism degree $P_n = 3$ from the perspective of the master process.

software versions to execute to the amount that is reachable and, thereby, avoids the execution of superfluous tests.

In addition to the test latency reduction due to the efficient execution of common prefixes and the automatic reduction of the number of faulty versions that need to be executed, FASTFI reduces latencies further by allowing for the parallel execution of faulty versions of the same function. In the example in Figure 3, all faulty versions of F_4 can be executed in parallel. The same is true for F_5. As faulty versions need to be executed in their own processes in any case, there is no additional cost associated in executing them in parallel. The following section details the parallelization strategy employed as well as the control logic required to implement the FASTFI execution model and the monitoring of the child processes executing faulty versions.

3.3. FASTFI Fork Server: Control & Monitoring of Faulty Versions

We denote the control logic that is responsible for implementing the FASTFI execution model, which we described in the previous section, as the FASTFI fork server. The fork server replaces the function body of all functions for which faulty versions exist. For each such function, distinctive fork server code is generated that controls which version gets executed and which degree of parallelism is employed.

3.3.1. Parallelization Strategy. FASTFI parallelizes the execution of faulty versions by grouping all versions of each function into groups of size P_n where P_n denotes the degree of parallelism used, i.e., the number of faulty versions that may execute in parallel. P_n is a runtime parameter that can be chosen by the user for each run of the integrated FASTFI executable. Each of the parallel groups is executed concurrently by forking all P_n versions at once. Before executing the next group, the previous group has to finish. Figure 4 illustrates an example for the execution of some function F, which is executed with parallelism degree $P_n = 3$. Once function F is invoked by the master process for the first time, all its versions need to be executed before the master process can eventually execute the fault-free version F^{ORIG}. Hence, the master executes all parallel groups for F sequentially until all groups have been executed. However, the members of each group are executed in parallel.

The faulty version groups are generated by dividing the list of all versions into consecutive non-overlapping chunks of size P_n. The last group generated may be smaller than P_n if the number of versions is not evenly divisible by P_n. The total execution time of all faulty versions of a function is determined by the longest execution times among the versions executed within each parallel group. The total execution time is minimized if all members of a group have similar execution times. Therefore, the list of all versions should ideally be ordered according to expected execution times before chunking. Since this information is generally not know ahead of time, i.e., before actual execution, we order the version list according to the mutation operators that were applied to generate the faulty versions. In our experience, faulty versions that have been generated by the same mutation operators often show a tendency to result in similar execution times.

3.3.2. Control Logic. The FASTFI fork server control logic replaces the function body of each function for which faulty versions exist. The original function versions are saved in the version library for each function and can still be invoked by both the master process as well as fault executing processes.

The listing in Figure 5 provides a simplified description of the FASTFI fork server logic for some function `foo` in the form of C-like pseudo code. The fork server logic starts in lines 2 to 5 by verifying if the FASTFI execution model should be used or if the user requested a traditional execution in which only one version, which is chosen by the user, gets executed. Both the execution mode and the requested version to execute are runtime parameters that can be configured by the user upon each execution of the integrated FASTFI executable. This feature allows testers to investigate the behavior of individual faulty versions in detail without the overhead of re-compilation. In the traditional execution mode (discussed in Section 3.2.1), there is no distinction between a master and a fault executing process and no integrated monitoring is in place. In order to actually invoke a requested function version, the version is looked up in the version library and called dynamically as shown in lines 3 and 4.

The FASTFI execution model is implemented by the logic in lines 6 to 22. The logic has to distinguish between

```
 1 ret_type foo(args) {
 2   if (in_single_version_mode) {
 3     return call_version(
 4         foo, args, requested_version);
 5   }
 6   if (forked) { // in faulty execution (1)
 7     if (is_active(foo))
 8      return call_version(foo, args, CUR_ACT);
 9     else
10       return call_version(foo, args, ORIG);
11   } else if (!forked && already_done(foo)) {
12     // master: all versions done (2)
13     return call_version(foo, args, ORIG);
14   } else {
15     // master: exec faulty versions (3)
16     for (par_group in parallel_groups(foo)) {
17       fork_monitor_faulty(par_group);
18       join_master(par_group);
19     }
20     set_already_done(foo);
21     return call_version(foo, args, ORIG);
22   }
23 }
24
25 ret_type fork_monitor_faulty(par_group) {
26   for (cur_version in par_group) {
27     if (fork() == MONITOR) {
28       if (fork() == MUTANT) {
29         // run faulty version
30         forked = true;
31         CUR_ACT = cur_version;
32         setup_env();
33         return call_version(
34             foo, args, CUR_ACT);
35       } else {
36         // monitor faulty version
37         results = observe_wait(cur_version);
38         log(results);
39         exit_monitor();
40       }
41     }
42   }
43 }
```

Figure 5. Pseudo Code of the FASTFI Fork Server Control and Monitor Logic.

three execution states as the master and all forked processes share the same code:

(1) in fault executing process (lines 6 to 10),
(2) in master process after all faulty function versions have been executed (lines 11 to 13), and
(3) in master process upon the first function invocation (lines 14 to 22).

In state 1, the invocation of the correct version within a fault executing process is implemented; in state 2, the invocation of only the original, fault-free versions is guaranteed for the master process; in state 3, the actual selection and forking of faulty versions takes place.

In state 1, the logic has to distinguish whether a faulty version of the function (`foo` in the example) is active in the current process (line 7). If so, the correct faulty version from the library is invoked; if not, the original version is called. This guarantees that each fault executing process executes only one faulty version of any function.

State 3 corresponds to the situation exemplified in Figure 4 and discussed in Section 3.3.1, i.e., the actual forking of the parallel version groups happens here. The master process iterates over all version groups `par_group` (lines 16 to 19) and invokes the helper function `fork_monitor_faulty` for each of them, resulting in the execution of the faulty versions. Each loop iteration waits at the end until the execution of the current group finishes before starting the next iteration. After all faulty versions have been executed, the master process marks the function as done (line 20) to prevent redundant re-executions. As last step, the original function version is invoked (line 21) which finishes the fork server execution and advances the fault-free execution of the master process.

The actual forking logic is implemented in `fork_monitor_faulty` (lines 25 to 43). The function iterates over all P_n members `cur_version` of the current version group `par_group`. For each version `cur_version`, *two* processes, `MONITOR` and `MUTANT`, are created via `fork` calls. The `MONITOR` process, which has not been discussed so far, is required to perform reliable monitoring of the fault executing process. This monitoring needs to occur in a separate process since the fault executing process itself may behave erratically and, for instance, crash or hang indefinitely. The `MONITOR` process is created first (line 27) such that the `MUTANT` process becomes its child (line 28). Therefore, `MONITOR` can exercise process control over `MUTANT`. For instance, it can terminate `MUTANT` and it can observe crashes and exits of `MUTANT`. The `MONITOR` logic is shown in lines 36 to 39. `MONITOR` waits until the `MUTANT` process, which executes `cur_version`, finishes execution, or terminates it if execution takes longer than a user-specified timeout to ensure progress, fetches observed results and logs them for later analysis.

The `MUTANT` control logic is shown in lines 29 to 34. First, `MUTANT` marks itself as fault executing process (line 30) and remembers which version it is supposed to execute (line 31). Next, it performs additional environment setup steps (line 32) such as I/O redirection. As last step, `MUTANT` finally invokes the faulty function version for the first time (line 33). At this point, `MUTANT` continues with the independent execution using the faulty version `CUR_ACT` upon each function invocation (`foo` in this example).

3.4. Static Analysis & Version Library Generation

FastFI requires knowledge about the static structure of both the input source code as well as the SFI mutation patches in order to be able to correctly replace function bodies, generate the FastFI fork server code, and to generate the library of faulty versions. To that end, FastFI relies on an existing static analysis framework to extract the necessary information about all functions present in the input source code. FastFI requires information about where functions reside in the source code, their function signature, and function parameter names. In a static analysis step, FastFI builds an analysis database with the required information for later use in the workflow as described in Section 3.1.

The mutation patches are parsed and information about modified source code lines are extracted. This information is then used to search the analysis database to match mutation patches to the functions that they mutate, i.e., the faults are grouped according to the source code function where they will reside. Once the grouping is complete, FastFI generates the library of faulty source code functions. For that purpose, each mutation patch is applied to the source code and the resulting faulty function is extracted, given a unique name, and added to the library. After each patch application, the original source version is restored to produce faulty versions that contain exactly one fault. As a final step, an unmodified version of each function is added to the library as well.

3.5. Limitations

We discuss technical limitations that may impede the application of FastFI in the following. Since FastFI relies on the `fork` system call as specified by POSIX, FastFI can be used only in environments where `fork` or a compatible system call is available. Moreover, an invocation of `fork` must leave both the calling and the created child process in a well defined state from which independent executions of parent and child processes are possible. This is not the case for multi-threaded processes as only the calling thread survives a `fork` invocation and the created process has only limited abilities to invoke further system services.

Software may behave differently under the FastFI execution model under certain circumstances. If the software's behavior depends on explicit process attributes, such as the process identifier (PID), its behavior may change as FastFI creates new processes with possibly changed attributes (e.g., different PIDs). Software that relies on explicit time information, e.g, by using timers or explicit time duration, may behave differently as FastFI effectively pauses the execution of the master process while faulty versions are executed. Moreover, software that contains severe defects such as invalid memory accesses in the original program may have different effects in FastFI as the memory layout between the generated executables differs.

FastFI isolates the execution of faulty software versions by means of OS process isolation. This leaves external resources that are not covered by process isolation as possible sources of interferences. While FastFI handles open files, additional measures need to be taken to also handle hardware devices or network connections.

3.6. Implementation

We developed a prototype of FastFI for software that is written in the C language and executes in a POSIX compliant environment. Our prototype relies on Coccinelle[3] as static analysis framework for C source code. It is mainly developed in Python and can, as the evaluation in Section 4 demonstrates, efficiently handle real world software despite the fact that it is not yet optimized for performance.

3. http://coccinelle.lip6.fr

Table 1. OVERVIEW OF THE PARSEC APPLICATIONS USED IN THE EVALUATION.

Application	Description	Mutants
`blackscholes`	Numerical financial computations	416
`dedup`	Data stream compression	662
`ferret`	Content-based image similarity search	6157
`x264`	Video stream encoding and compression	13368

Please note that, although our prototype currently only supports software written in C, FASTFI itself is not limited to C software. Software written in other languages, such as C++ or Rust, can also benefit from FASTFI.

4. Evaluation

In order to evaluate the applicability and performance of FASTFI for real world software, we investigate the following research questions using our prototype implementation for C software.

RQ 1 How much can FASTFI reduce overall test execution latencies for sequential SFI tests?

RQ 2 How does the execution speedup achieved by FASTFI develop with increasing degree of parallelism?

RQ 3 Do SFI test results remain stable across runs with increasing degree of parallelism when using FASTFI?

RQ 4 How large is the build time overhead of integrated FASTFI builds compared to traditional separate builds?

4.1. Experimental Setup

Execution Environment. We conduct our experiments on a machine with up to date Debian Buster (Linux 4.16, x86_64) as operating system. The machine is equipped with an AMD Ryzen 7 CPU with 8 physical and 16 logical cores running at 3.40 GHz, 32 GiB of main memory, and a 1 TiB SSD.

Evaluation Targets. We apply FASTFI to four applications from the widely used PARSEC benchmark suite 3.0[4] provided by Princeton University [12]. Table 1 gives a brief overview of the selected applications. We selected these four applications since they are representative for different application domains and they are written in C, which our current prototype implementation targets. We use the "simmedium" workloads that ship with PARSEC to exercise the applications. These workloads are of a moderate size, which allows us to execute our experiments within a reasonable time frame (within days).

Execution Steps. To investigate our research questions, we take the following steps for all selected evaluation targets.

We first apply the SAFE[5] software fault injection tool [8] to generate mutation patches. SAFE applies 13 different mutation operators to generate representative software faults. An overview of the generated mutants is given in Table 1. Each mutant creates a faulty software version that needs to be executed for SFI tests.

Next, we perform the static analysis of the input source code using Coccinelle to generate the analysis database as described in Section 3.4. We then analyze the generated mutation patches and perform the function level fault grouping. Afterwards, we generate the library of faulty versions by applying the mutation patches and extracting the resulting modified functions as well as saving the original, unmodified function version. Then, the original function bodies are replaced with the generated FASTFI fork server code as described in Section 3.3. As final step, we build the integrated executable with the PARSEC default build configuration "gcc-serial" that results in non-multithreaded executables.

We perform our experiments using the generated integrated executables in our execution environment. We repeat each experiment 3 times and report averages.

4.2. RQ 1: Sequential Speedup

To determine the impact of FASTFI on sequential SFI execution latency, we compare the performance of FASTFI without any parallelization ($P_n = 1$) to the performance achieved by separately executing each faulty version. For the separate executions baseline, we make use of our single version mode as described in Section 3.3.2, i.e., we still use the integrated executables generated by FASTFI. However, the faulty version to execute is picked prior to execution, and only one faulty version is chosen for each program execution. Consequently, executions in this mode do not benefit from the ability of FASTFI to avoid redundant code execution and the execution flow corresponds to a traditional SFI execution model as described in Section 3.2.1.

As shown in the leftmost column of Figure 6, FASTFI can achieve speedup factors from 1.3 to 3.6, depending on the benchmark. In the absence of parallelization, these speedups are the result of avoiding redundant code execution. FASTFI avoids redundant code execution in two ways: (1) By efficiently executing common prefixes and (2) by automatically reducing the number of faulty versions that need to be executed. The reduction in the number of faulty versions is shown in Figure 7. For three out of four benchmarks, FASTFI automatically executes fewer faulty versions than the traditional execution model as unreachable faulty versions are not executed. The maximum reduction can be observed for `ferret` where FASTFI reduces the number of faulty versions down to 47.9 %. This substantial reduction is also reflected in `ferret`'s speedup factor of 3.6. Moreover, despite executing the same number of faulty versions, FASTFI achieves a speedup of 1.3 over the traditional execution model for the `blackscholes` benchmark. This reduction is the effect of FASTFI's efficient common prefix execution.

4. http://parsec.cs.princeton.edu/parsec3-doc.htm
5. http://wpage.unina.it/roberto.natella/tools.html

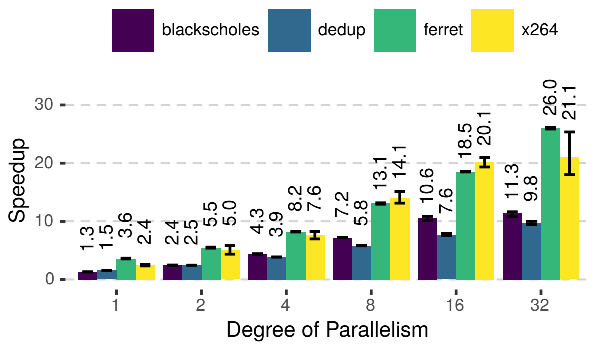

Figure 6. FASTFI speedup relative to traditional execution model for increasing degrees of parallelization (P_n). Error bars indicate minimum and maximum speedup.

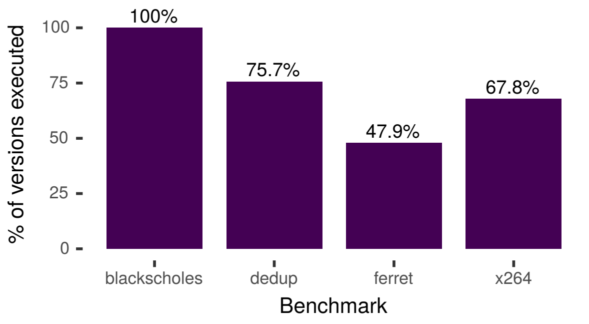

Figure 7. Percentage of faulty versions executed during (sequential) FASTFI execution. The reduction is due to FASTFI's ability to avoid execution of unreachable versions.

4.3. RQ 2: Parallel Speedup

To investigate how the speedup achieved by FASTFI develops with increasing degrees of execution parallelism, we configure FASTFI to run up to 32 faulty versions in parallel. Note that changing the degree of parallelism is handled by the FASTFI runtime code and does not require recompilation (see Section 3.3.2). The speedups relative to traditional execution for the different degrees of execution parallelism are shown in Figure 6. FASTFI achieves increasing speedups with an increasing degree of parallelism. When executing 16 faulty versions in parallel, which corresponds to the number of logical cores on the machine we use for our evaluation, FASTFI achieves a speedup of 7.6 to 20.1 compared to the traditional execution model. Relative to FASTFI execution without parallelism, the speedups range from 5.0 to 8.4. When going beyond the number of available cores by executing 32 faulty versions in parallel, FASTFI achieves speedups ranging from 9.8 to 26.0 relative to the traditional execution model, or 6.5 to 8.8 over FASTFI execution without parallelism. These results show that parallel FASTFI execution enables significant speedups over traditional SFI execution as well as over FASTFI execution without parallelization. By optimizing the FASTFI fork server architecture to allow for dynamic parallel groups (see Section 3.3), we believe that even higher speedups can be achieved.

4.4. RQ 3: SFI Result Stability

To determine whether increasing degrees of parallelism affect SFI result stability, we configure FASTFI to run up to 32 faulty versions in parallel and compare SFI test outcomes. We consider the common four classes of SFI test outcomes: "Crash": application crash, "Error": termination with error indication, "Success": termination without error indication, and "Timeout": application did not finish in time. From an application perspective, these failure modes match the crash and hang oracles that are most commonly applied for SFI and robustness tests [28]. We set the timeout values to 3 times the duration of a fault-free execution for each benchmark to account for increased individual execution latencies in parallel testing [17].

Figure 8 shows the SFI test outcomes for different degrees of parallelism. The rightmost columns labeled with "s" show results from the sequential single version execution mode that corresponds to a traditional execution. The higher count of successful tests for this mode is due to the fact that all faulty versions are executed independent of whether the faults are reachable. Such "dead" versions always result in success as their execution always corresponds to a fault-free execution. Since FASTFI avoids the execution of such "dead" versions, the success count for FASTFI runs is lower.

For all benchmarks, the results are stable for up to 16 parallel executions. When executing 32 faulty versions in parallel, results remain stable for the `blackscholes` benchmark. For the other three benchmarks, the number of crashes and errors remain stable but the number of successful tests drops and the number of timeouts increases compared to lower degrees of parallelism. Moreover, for the `x264` benchmark, the number of successful executions and timeouts varies between test runs at this degree of parallelism. As this effect only occurs when running at a degree of parallelism well in excess of the available computational resources on the machine we use for our experiments, we expect that spurious timeouts at this degree of parallelism can be avoided by choosing a higher timeout threshold, at the cost of increased SFI test latency (cf. [17]).

4.5. RQ 4: Build Time Overhead

To investigate how large the overhead for creating integrated FASTFI executables is, we build the same set of faulty versions twice: once with FASTFI and once by building separate executables for each faulty version. In the latter case, we utilize incremental compilation. Therefore, for each faulty version, one compilation unit is recompiled and the final executable is linked. This is a typical approach for building faulty program versions for SFI tests. User times for building with FASTFI relative to the traditional model are shown in Figure 9. Note that the recorded times include the application of the mutation patches and, for FASTFI, code generation. FASTFI offers substantially lower build

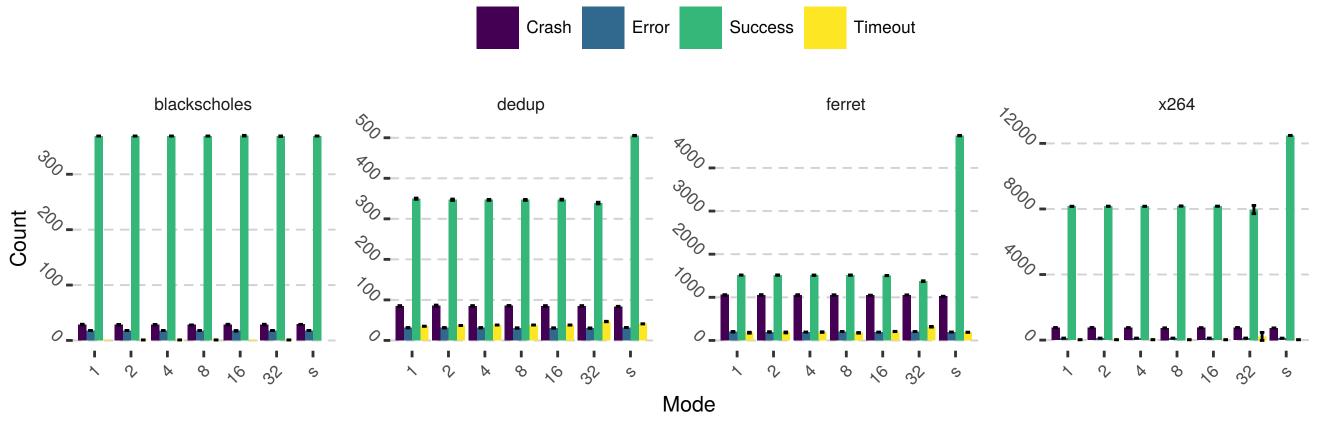

Figure 8. SFI test results for different modes of execution and degrees of parallelism. The x axis labels indicate the employed degree of parallelsims (P_n) for FASTFI execution. The "s" label indicates the sequential single version mode execution. Error bars indicate standard deviation.

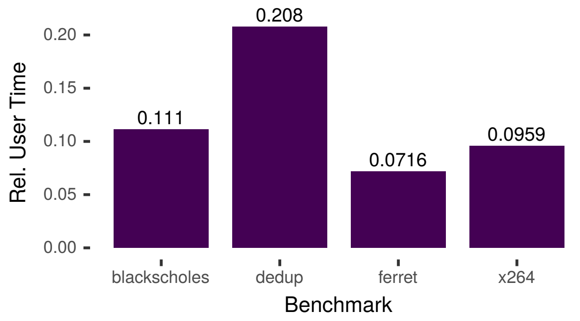

Figure 9. FASTFI user build times relative to user build times for separate executables.

times compared to the traditional approach: FASTFI builds take between 7.2 % and 20.8 % of the user time required for building separate executables for each faulty version. This corresponds to a speedup between 4.8 and 13.9. For `x264`, this speedup reduces the build time from almost 6 h to 35 min. The reason for this advantage is that FASTFI avoids redundant recompilation: The traditional approach incurs substantial overhead due to always recompiling entire compilation units, even though only a single function differs from the fault-free version. Since FASTFI works at function granularity, it avoids this overhead by design.

4.6. Discussion

Our investigation of FASTFI with regard to our four research questions shows that FASTFI can be applied to real world software and it is effective at avoiding redundant code re-execution, enabling sequential speedups of up to 3.6 over a traditional execution model. Our results also show that FASTFI enables further speedups through parallelization, which can be even further improved by using different parallelization strategies than the one implemented in our prototype. FASTFI therefore enables the effective utilization of modern parallel computing hardware for SFI tests. We find that neither sequential nor parallel FASTFI execution adversely affects SFI test result stability unless the degree of parallelism exceeds the available computational resources, in which case spurious timeouts commonly arise [17]. Such issues can be addressed by adjusting timeout thresholds at the cost of potentially higher execution latencies. Finally, our investigation shows that FASTFI enables faster compilation of faulty versions due to the finer, function-level granularity our approach offers. Overall FASTFI reduces latencies for both the compilation of faulty software versions and their execution.

5. Conclusion and Future Work

In this paper, we introduced FASTFI, a novel approach for accelerating SFI testing by (1) avoiding redundant code execution, (2) avoiding the execution of "dead" faulty versions, (3) parallelization of test execution, and (4) reducing build times for faulty versions. Based on an evaluation of FASTFI on four widely used benchmark programs from the PARSEC suite, we conclude that FASTFI is applicable to real world software from various application domains, enables both sequential execution speedup as well as effective parallelization, and substantially reduces build times.

FASTFI achieves speedups of up to 3.6 in sequential execution and up to 26 in parallel execution. The number of executed faulty versions is reduced by up to 52.1 %. FASTFI can reduce build times to as little as 7.2 % of conventional SFI approaches. FASTFI achieves these improvements while maintaining result stability and is therefore a viable approach for reducing SFI test latencies in real world settings.

In the future, we plan to extend this work in several directions. Different parallelization strategies, such as replacing the fixed chunks currently used by FASTFI with work stealing, may result in improved CPU utilization and a further reduction in SFI test latencies. Our current prototype is limited to programs written in C and we are planning to support C++ as well. Finally, the FASTFI execution model can be applied to other fault injection techniques beyond code mutations, such as interface error injections.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP at TU Darmstadt.

References

[1] J.-L. Lions *et al.*, *Ariane 5 flight 501 failure*, 1996.

[2] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, “Predicting How Badly “Good” Software Can Behave”, *IEEE Softw.*, vol. 14, no. 4, pp. 73–83, 1997.

[3] J. Durães and H. Madeira, “Emulation of Software faults: A Field Data Study and a Practical Approach”, *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, 2006.

[4] D. Cotroneo and R. Natella, “Fault Injection for Software Certification”, *IEEE Security Privacy*, vol. 11, no. 4, pp. 38–45, 2013.

[5] R. Chillarege and N. Bowen, “Understanding large system failures-a fault injection experiment”, in *Proc. FTCS*, 1989, pp. 356–363.

[6] W. I. Kao, R. K. Iyer, and D. Tang, “FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults”, *IEEE Trans. Softw. Eng.*, vol. 19, no. 11, pp. 1105–1118, Nov. 1993.

[7] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat, “MAFALDA: Microkernel Assessment by Fault Injection and Design Aid”, in *Proc. EDCC*, J. Hlavička, E. Maehle, and A. Pataricza, Eds., 1999, pp. 143–160.

[8] R. Natella, D. Cotroneo, J. Durães, and H. Madeira, “On Fault Representativeness of Software Fault Injection”, *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 80–96, Jan. 2013.

[9] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs”, in *Proc. POPL*, 1980, pp. 220–233.

[10] R. A. DeMillo and A. J. Offutt, “Constraint-based automatic test data generation”, *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[11] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing”, *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[12] C. Bienia, “Benchmarking Modern Multiprocessors”, PhD thesis, Princeton University, Jan. 2011.

[13] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, “D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology”, in *Proc. CCGRID*, 2010, pp. 631–636.

[14] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato, “Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems”, in *Proc. ICSTW*, 2010, pp. 428–433.

[15] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, “A whitebox approach for automated security testing of Android applications on the cloud”, in *Proc. AST*, 2012, pp. 22–28.

[16] R. Banabic and G. Candea, “Fast black-box testing of system recovery code”, in *Proc. EuroSys*, 2012, pp. 281–294.

[17] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, “No PAIN, No Gain?: The Utility of PArallel Fault INjections”, in *Proc. ICSE*, 2015, pp. 494–505.

[18] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado, “GridUnit: Software Testing on the Grid”, in *Proc. ICSE*, 2006, pp. 779–782.

[19] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, “Parallel Test Generation and Execution with Korat”, in *Proc. ESEC/FSE*, 2007, pp. 135–144.

[20] T. Parveen, S. Tilley, N. Daley, and P. Morales, “Towards a distributed execution framework for JUnit test cases”, in *Proc. ICSM*, 2009, pp. 425–428.

[21] M. Oriol and F. Ullah, “YETI on the Cloud”, in *Proc. ICSTW*, 2010, pp. 434–437.

[22] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, “Empirically Revisiting the Test Independence Assumption”, in *Proc. ISSTA*, 2014, pp. 385–396.

[23] J. Candido, L. Melo, and M. D’Amorim, “Test Suite Parallelization in Open-Source Projects: A Study on Its Usage and Impact”,

[24] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, “Efficient dependency detection for safe Java test acceleration”, in *Proc. ESEC/FSE*, 2015, pp. 770–781.

[25] W. Lam, S. Zhang, and M. D. Ernst, “When tests collide: Evaluating and coping with the impact of test dependence”, University of Washington Department of Computer Science and Engineering, Tech. Rep., 2015.

[26] A. Gambi, S. Kappler, J. Lampel, and A. Zeller, “CUT: Automatic Unit Testing in the Cloud”, in *Proc. ISSTA*, 2017, pp. 364–367.

[27] J. Bell and G. Kaiser, “Unit Test Virtualization with VMVM”, in *Proc. ICSE*, 2014, pp. 550–561.

[28] P. Koopman, K. DeVale, and J. DeVale, “Interface robustness testing: Experience and lessons learned from the ballista project”, *Dependability Benchmarking for Computer Systems*, vol. 72, p. 201, 2008.