Check for updates

Received: 5 October 2022 | Revised: 16 August 2023 | Accepted: 8 September 2023

DOI: 10.1002/stvr.1865

RESEARCH ARTICLE

WILEY

Mutation testing optimisations using the Clang front-end

Sten Vercammen[1] | **Serge Demeyer**[1,2] | **Markus Borg**[3,4] | **Niklas Pettersson**[5] | **Görel Hedin**[4]

[1]Department of Informatics, University of Antwerp, Antwerp, Belgium

[2]Flanders Make, Antwerp, Belgium

[3]Digital Systems, RISE Research Institute of Sweden, Lund, Sweden

[4]Department of Computer Science, Lund University, Lund, Sweden

[5]Saab Aeronautics, Saab AB, Linköping, Sweden

Correspondence
Serge Demeyer, Department of Informatics, University of Antwerp, Antwerp, Belgium.
Email: serge.demeyer@uantwerpen.be

Present address
Serge Demeyer, Department of Computer Science, Universiteit Antwerpen, Middelheimlaan 1, BE-2020 Antwerp, Belgium.

Funding information
Fonds Wetenschappelijk Onderzoek; Research Foundation Flanders (FWO), Grant/Award Number: 1SA1519N; FWO-Vlaanderen and F.R.S.-FNRS, Grant/Award Number: 30446992 SECO-ASSIST

Abstract

Mutation testing is the state-of-the-art technique for assessing the fault detection capacity of a test suite. Unfortunately, a full mutation analysis is often prohibitively expensive. The CppCheck project for instance, demands a build time of 5.8 min and a test execution time of 17 s on our desktop computer. An unoptimised mutation analysis, for 55,000 generated mutants took 11.8 days in total, of which 4.3 days is spent on (re)compiling the project. In this paper, we present a feasibility study, investigating how a number of optimisation strategies can be implemented based on the Clang front-end. These optimisation strategies allow to eliminate the compilation and execution overhead in order to support efficient mutation testing for the C language family. We provide a proof-of-concept tool that achieves a speedup of between 2× and 30×. We make a detailed analysis of the speedup induced by the optimisations, elaborate on the lessons learned and point out avenues for further improvements.

KEYWORDS
C++, CLANG, mutant schemata, mutation testing

1 | INTRODUCTION

DevOps (Software Development and IT Operations) is defined by Bass et al. as *a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality* [1]. This allows for frequent releases to rapidly respond to customer needs. Tesla, for example, uploads new software in its cars once every month [2]; Amazon pushes new updates to production on average every 11.6 s [3]. The enabling factor for the DevOps approach is a series of automated tests that serve as quality gates, safeguarding against regression faults.

The growing reliance on automated software tests raises a fundamental question: How trustworthy are these automated tests? Today, mutation testing is acknowledged within academic circles as the most promising technique for assessing the *fault-detection capability* of a test suite [4, 5]. The technique deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. The more mutants the test suite can detect, the higher its fault-detection capability is—referred to as the *mutation coverage* or *mutation score*.

Case studies with safety-critical systems demonstrate that mutation testing could be effective where traditional structural coverage analysis and code inspections have failed [6, 7]. Google on the other hand reports that mutation testing provides insight into poorly tested parts of the system, but—more importantly—also reveals design problems with components that are difficult to test, hence must be refactored [8]. In a similar vein, a blog post from a software engineer at

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited and is not used for commercial purposes.
© 2023 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

NFluent, comments on integrating Stryker (a mutation tool for .Net programs) in their development pipeline [9]. There as well, mutation testing revealed weaknesses in the test suite, but also illustrated that refactoring allowed for simpler test cases, which subsequently increased the mutation score.

Despite the apparent potential, mutation testing is difficult to adopt in industrial settings. One of the reasons is because the technique—in its basic form—requires a tremendous amount of computing power. Without optimisations, the entire code base must be compiled and tested separately for each injected mutant [4]. During one of our experiments with an industrial code base, we witnessed 48 h of mutation testing time on a test suite comprising 272 unit tests and 5256 lines of test code for a system under test comprising 48,873 lines of production code [10]. Hence for medium to large test suites, mutation testing without optimisations becomes prohibitively expensive.

As a consequence, the last decades has devoted a lot of research to optimise the mutation testing process [5, 11]. One stream of work focuses on *parallelisation*, either on dedicated hardware [12] or in the cloud [10]. Another stream of work focuses on *incremental approaches*, limiting the mutation analysis to what has been changed since the previous run [13]. A third stream of work (and the inspiration for what is presented in this paper) focuses on techniques based on program analysis, for example *mutant schemata* [14] and *split-stream mutation testing* [15]. The former injects all mutants simultaneously, analysing the abstract syntax tree to ensure that the mutated version actually compiles. The latter forks the test execution from the mutation point via a combination of static and dynamic program analysis.

Mutation testing shines in systems with high statement coverage because uncaught mutants reveal weaknesses in code, which is supposedly covered by tests. Safety-critical systems—where safety standards dictate high statement coverage—are therefore a prime candidate for validating optimisation strategies. In a safety-critical software, C and C ++ dominate the technology stack [16]. Yet this is not represented in the mutation testing community: a systematic literature review on mutation testing from 2019 reports that less than 25% of the primary studies target source code from the C language family [5]. (In this paper, we use the term C language family to describe the C, Objective C, C++, Objective C++ and their variants.) This opens up opportunities as the C language family is a mature technology with considerable tool support available. In particular, the compiler front-end Clang that operates in tandem with the LLVM compiler back-end [17].

This paper presents a feasibility study, investigating to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. Mutation testing and mutation optimisations for statically typed languages like the C family are challenging to implement as they have many interacting features and unforgiving compilers. Utilising the Clang front-end might alleviate some problems, as many mutation optimisations require syntactically correct mutants, which might be difficult to achieve without relying on the compiler.

We present a proof-of-concept optimisation tool, featuring a series of representative optimisation strategies. These optimisation strategies are

1. *exclude invalid mutants*: avoid compilation overhead from mutants that would cause downstream compilation errors.
2. *exclude unreachable mutants*: avoid execution overhead from mutants that are not reached by the test suite.
3. *mutant schemata*: where all mutants get injected simultaneously and the project is only compiled once [14]. At test execution time, the appropriate mutant is selected via a boolean flag.
4. *reachable mutant schemata*: an extension of mutant schemata that reduces the test suite to only those test cases that reach the mutant.
5. *split-stream mutation testing*: where tests are executed from the mutation point itself, by forking the process, essentially avoiding redundant executions [15].

We validate the proof-of-concept tool on four open-source C++ libraries and one industrial component. These cases cover a wide diversity in size, C++ language features used, compilation times and test execution times. Hence, they may serve as a representative benchmark to validate mutation optimisation strategies. To illustrate the potential of Clang-based implementation of mutation optimisation strategies, we report the overhead induced and the speedup achieved, both in absolute as well as relative terms. Furthermore, we derive a series of lessons learned on the benefits and impediments of implementing mutation testing optimisations using the Clang compiler front-end.

As such, we make the following contributions to the state of the art in mutation testing optimisation.

- *Whole is more than the sum of the parts*: We are the first to study the *combined* effect of different optimisation strategies. We estimate the potential savings on both the compilation time and test execution time, while estimating the additional overhead induced by the Clang front-end analysis.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

- *Empirical Validation*: We empirically validate these estimates against four projects not considered in the previous optimisation studies. We report maximum and minimum speedups achieved and explain the properties of the projects under test that explain the large variation we observed.
- *Demonstrate Feasibility of Clang*: By means of a proof-of-concept tool, we demonstrated how the Clang compiler front-end can be used to implement a variety of optimisation strategies. We also list the limitations of the Clang compiler front-end for implementing advanced mutation testing strategies. In particular, we learned that split-stream mutation testing demands a test suite without any external dependencies as the Clang compiler front-end does not provide facilities to identify (and consequently revert) state-changes in dependent components and libraries. We list a series of lessons learned that allows the mutation testing community to build upon our observations and optimise even more.

The rest of the paper is structured as follows. In Section 2, we elaborate on the concept of mutation testing and list related work. In Section 3, we describe the design of our proof-of-concept tool, including the estimates for the combined effect of different optimisation strategies. In Section 4, we explain how we obtained the speedups induced by the different optimisation strategies. This naturally leads to Section 5 where we discuss the results and Section 6 where we derive the lessons learned. As with any empirical research validating proof-of-concept tools, our study is subject to various threats to validity, which are listed in Section 7. Finally, we draw conclusions in Section 8.

2 | BACKGROUND AND RELATED WORK

In this section, we elaborate on the concept of mutation testing, the different optimisation strategies as available in the related work and discuss existing work mutation optimisation based on the Clang front-end.

2.1 | Mutation testing terminology

For effective testing, software teams need strong tests that maximise the likelihood of exposing faults [18]. Traditionally, the strength of a test suite is assessed using code coverage, revealing which statements are poorly tested. However, code coverage has been shown to be a poor indicator of test effectiveness [19, 20]. Stronger coverage criteria, like full MC/DC coverage (Modified Condition/Decision Coverage, a coverage criterion often mandated by functional safety standards that target critical software systems, e.g., ISO 26262 and DO-178C) still do not guarantee the absence of faults [21, 22]. Today, mutation testing (sometimes also named *mutation analysis*, the terms are used interchangeably) is the state-of-the-art technique for assessing the *fault-detection capacity* of a test suite [4, 5].

Killed / survived mutants. Mutation testing deliberately injects faults (called mutants) into the production code and counts how many of them are caught by the test suite. A mutant caught by the test suite, that is, at least one test case fails on the mutant, is said to be *killed*. When all tests pass, the mutant is said to *survive*.

Mutation Coverage. A strong test suite should have as few surviving mutants as possible. This is expressed in a score known as the mutation coverage: the number of mutants killed divided by the total number of mutants injected. A high mutation coverage implies that most mutants get killed; 100% is a perfect score as the tests can reveal all small deviations. Mutation coverage is sometimes referred to as mutation score.

Mutation Operators. Mutation testing mutates the program under test by artificially injecting a fault based on a mutation operator. A mutation operator is a source code transformation that introduces a change into the program under test. Typical examples are replacing a conditional operator (e.g., $\leq$ into $<$) or an arithmetic operator (e.g., $+$ into $-$). The first set of mutation operators was reported in King et al. [15]. Later, special purpose mutation operators have been proposed to exercise errors related to specific language constructs, such as Java null-type errors [23] or C++11/14 lambda expressions and move semantics [24]. There are more than 100 mutation operators reported in the academic literature and there is no consensus of which ones are best for a specific language and code base. Therefore, mutation testing tools feature a diverse set of mutation operators that can be configured when performing the mutation analysis. Table 1 lists commonly used mutation operators for C and C++, which we will refer to later in this paper.

Invalid Mutants. Mutation operators introduce syntactic changes, hence may cause compilation errors in the process. If we apply the arithmetic mutation operator (AOR) to, for example, 'a * b', then we get four mutants as shown in Listing 1. However, the modulo operator ('%') will give an '`invalid operands to binary expression`' error, as the modulo operator is not defined for floating point data types. The mutant can thus not be compiled and is considered invalid.

Another frequently occurring invalid mutant occurs when changing the '+' into a '−' that works for numbers but does not make sense when applied to the C++ string concatenation operator. If the compiler cannot compile the

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 Overview of commonly used mutation operators for C and C++.

Code	Short	Description
ROR	Relational Operator Replacement	Replace a single operator with another operator. The relational operators are $<, <=, >, >=, ==, !=$
AOR	Arithmetic Operator Replacement	Replace a single arithmetic operator with another operator. The operators are: $+, -, *, /, \%$
LCR	Logical Connector Replacement	Replace a logical connector with the inverse. The logical connectors are: ||, &&, |, &
UOI	Unary Operator Insertion	Insert a single unary operator in expressions. Example unary operators are: increment (++), decrement (−−), address (&), indirection (∗), positive (+), negative (−), …
SDL	Statement Deletion	Selective deletion of code, including removing a specific function call, or replacing a method body by `void`
AMC	Access Modifier Change	Changes the access level for instance variables and methods to other access levels. Access levels are `private`, `protected`, `public`
ISI	Super Keyword Insertion (or Base keyword insertion)	Inserts the `scope resolution operator:` so that a reference to the variable or the method goes to the overridden instance variable or method
ICR	Integer Constant Replacement	Replaces every constant c with a value from the set $\{-1,0,1,-c,c-1,c+1\} \setminus \{c\}$

mutant for any reason, the mutant is considered invalid and is not incorporated into the mutation coverage. Preventing the generation of invalid mutants is one way to optimise the mutation testing process.

Listing 1: Mutation Example

```
1 float f(float a, float b) {
2     return a * b;    // original code
3 }
4     return a + b;    // mutant 1
5     return a - b;    // mutant 2
6     return a / b;    // mutant 3
7     return a % b;    // mutant 4
8            ~~~~~~~Invalid operands to binary expression
```

Unreachable mutants. The Reach–Infect–Propagate–Reveal criterion (a.k.a. the RIPR model) provides a fine-grained framework to assess weaknesses in a test suite that are conveniently revealed by mutation testing [25]. It states that an effective test should first of all *Reach* the fault, then *Infect* the program state, after which it should *Propagate* as an observable difference, and eventually *Reveal* the presence of a fault. When a mutant is injected in a statement that is never executed by the test suite, it can never be killed. Therefore, one can optimise the mutation testing process by explicitly excluding unreachable mutants. To increase its effectiveness, this should be done on the test case level instead of on the entire test suite.

Weak versus strong mutation coverage. Based on the Reach–Infect–Propagate–Reveal criterion, the literature also distinguishes between weak and strong mutation coverage. For weak mutation coverage, it is sufficient that the mutant only infects the program state but that it should not necessarily pass to the propagate and reveal states. For strong mutation coverage, the mutant should pass through all states. This can be used as another optimisation criterion: if a test reaches a mutant but does not infect the program state one can also exclude the test from the execution [26]. Just as for unreachable mutants, this should be done on a test case level.

Equivalent mutants. Injected mutants can be syntactically different from the original software system, but semantically identical. These mutants do not modify the meaning of the original program and can therefore not be detected by the test suite. Such mutants are called equivalent mutants. They yield false negatives and decrease the effectiveness of the mutation analysis. Additionally, they waste developer time, as they need to be manually labelled as equivalent because they show up as survived mutants, which can never be killed. Consequently, a big challenge of mutation is handling (and/or eliminating) these equivalent mutants. An overview of techniques to overcome the equivalent mutant problem has been provided by Madeyski et al. [27]. One of the frequently used techniques is called *trivial compiler equivalence (TCE)*: if the compiler emits the same low-level code (machine code) then the mutant is guaranteed to be equivalent.

Timed out mutants. Some injected mutants cause the test to go into an infinite loop. To prevent such infinite loops, mutants with an excessively long execution time need to be detected and stopped. This is often done using a time out, which terminates the test after a predefined period of time. A mutant that causes such a time-out is considered killed.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2.2 | Mutation testing optimisations

To explain the time-consuming nature of the mutation testing process, Algorithm 1 shows the essential steps of a mutation analysis without any optimisations. The software system needs to build without errors and all software tests should succeed before mutation testing can even begin; this is called the *pre*-phase. Then, the two main phases are executed: (*A*) the generate mutants phase and (*B*) the execute mutants phase. In phase *A*, mutants are generated for all source files. In phase *B*, for each mutant, all tests are executed and the result—whether or not it was killed—is saved. Finally, all the results are gathered and the final report is created in the *post*-phase.

Algorithm 1 Pseudocode mutation testing

```
1:  function MUTATIONTESTING(srcFolder src)
2:      ▷ Pre: verify build and if all tests succeed
3:      if INITIALBUILDANDTESTS() ≠ True then
4:          return
5:      end if
6:
7:      ▷ A: generate mutants
8:      mutants ← []
9:      for all srcFile f ∈ src do
10:         fMutants ← GENERATEMUTANTS(srcFile f)
11:         mutants ← mutants + fMutants
12:     end for
13:
14:     ▷ B: execute mutants
15:     for all mutant m ∈ mutants do
16:         COMPILEMUTANT(mutant m)
17:         result ← EXECUTEMUTANT(mutant m, testsuite t)
18:         STORERESULT(result, mutant m, testsuite t)
19:     end for
20:
21:     ▷ Post: process results
22:     PROCESSRESULTS()
23: end function
```

A lot of research is devoted to optimising the mutation testing process, summarised under the principle—*do fewer*, *do smarter* and *do faster* [28].

- *Do fewer* approaches minimise the execution time by reducing the total number of mutants to execute. Such an optimisation can be implemented by generating fewer mutants on line 10 in Algorithm 1 or by selecting a subset of all mutants on line 15. Incremental approaches [13], limiting the mutation analysis to code changed in a commit are a particularly relevant example of a 'do fewer' approach. A reduced set of mutants normally incurs an information loss compared to the full set of mutants; however, the effectiveness is often acceptable [4]. Nevertheless, when the mutation testing tool provides sufficient guarantees to identify *invalid mutants* or *unreachable mutants*, excluding these is always an effective optimisation.
- *Do smarter* approaches attempt to minimise the execution time by exploiting the computer hardware (e.g., distributed architectures, vector processors, fast memory access). The `for` loops in lines 9 and 15 in Algorithm 1 have few data dependencies, hence can be executed in parallel. Parallel execution of mutants, either on dedicated hardware [12] or in the cloud [10] is known to speed up the process by orders of magnitude. *Split-stream mutation testing* is the de facto representative for 'do smarter' optimisations [15]. By retaining state information between test runs, split-stream mutation testing avoids the redundant execution of statements up until the mutation point.
- *Do faster* approaches attempt to minimise the execution time by reducing the execution cost for each mutant (cf., line 17 in Algorithm 1). By design, a mutated program is almost identical to the original program that can be exploited during the compilation step. *Mutant schemata* [14] is the best know example. All mutants get injected simultaneously (guarded by a global switch variable), hence the project is compiled only once in line 16. During the mutant execution phase (in line 17) the global switch is used to select the actual mutant to execute. The execution time of a mutant can

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

also be reduced with *test prioritisation* techniques. By rearranging the test suite, the tests with the highest likelihood of failure will be executed first, reducing the test suite run time using early-failure [29].

- *Hybrid approaches*. Different approaches are often synergistic, where a combination of techniques becomes more than the sum of the parts. Note that some approaches are orthogonal to one another hence are easy to combine. Excluding unreachable mutants, for example, can be combined with any other optimisation. Other approaches, however, may depend on each other. Mutant schemata, for instance, requires that all invalid mutants are excluded because even a single invalid mutant will immediately invalidate the whole mutated program. Measuring the speedup of a given optimisation strategy should take these synergies into account.

2.3 | LLVM & Clang compiler infrastructure

LLVM is a collection of compilation tools designed around a low-level language-independent intermediate representation, the LLVM IR. The project includes front-ends that translate source code to LLVM IR, optimisers that rewrite the LLVM IR to become faster, and backends that generate machine code from the LLVM IR for different architectures. We visualised this in Figure 1. Clang is the most well-known front-end for LLVM and supports languages in the C family, like C, C++ and Objective-C, among others. Internally, it represents programs as abstract syntax trees (ASTs). Clang includes a semantic analyser that does type-checking and other compile-time checks, before generating the LLVM IR. Furthermore, Clang contains a number of libraries based on the visitor pattern, allowing more analyses or transformations to be added to the front-end.

LLVM and Clang serve as the de facto standard for building static analysis tools for the C language family. It should therefore come as no surprise that several C and C++ mutation tools exist that build upon these frameworks. These tools mutate the program either at the AST level or at the LLVM IR level. Both approaches have advantages and disadvantages. Doing the mutations at the LLVM IR level has the advantage that they will work for any front-end, but the disadvantage that mutants injected in the LLVM IR are difficult or even impossible to trace back to a source representation in the original code under test, which allows for the generation of many invalid mutants.

The AST, on the other hand, is close to the source code. Mutating at this level provides good traceability, which is critically important for reporting back results to the developer in the *post*-phase (lines 18 and 22 in Algorithm 1). Another advantage of mutating at the AST level is that the front-end semantic analyser can be used to ensure that the mutated code is compile-time correct, effectively eliminating invalid mutants.

2.4 | Mutating on the LLVM IR and AST level

As the AST is close to the source code, the source code from Listing 1 can be transformed into an AST without losing any information. This can be seen in Listing 2 where even the original line and column information of the statements are retained. The AST includes all the semantic details in an easily accessible manner. This, together with the front-end

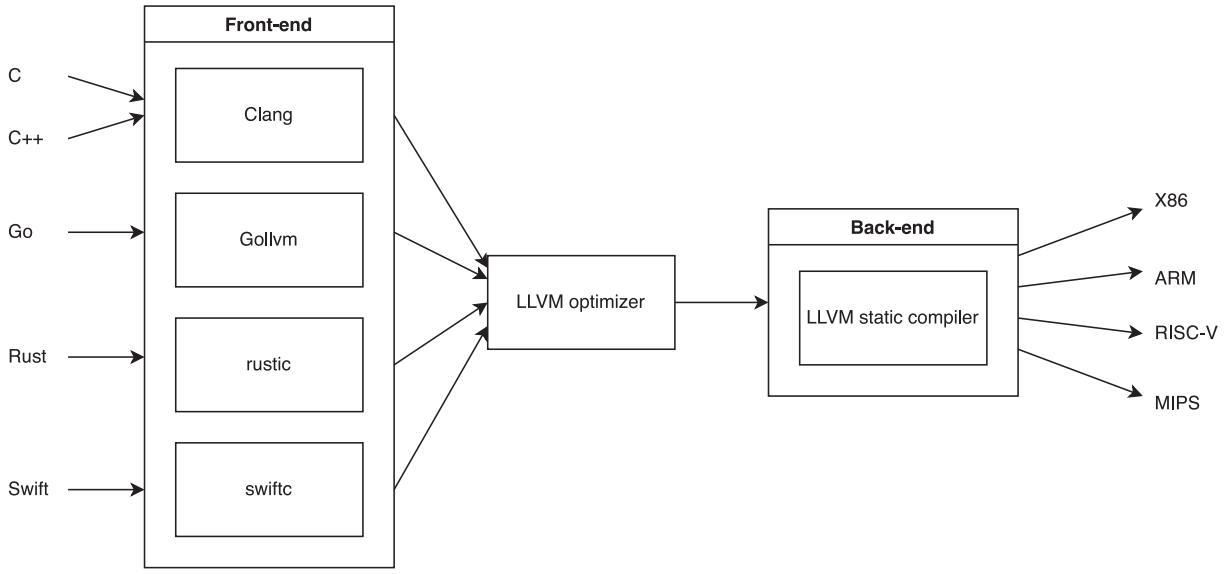

FIGURE 1 LLVM compilation.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

semantic analyser allows for more informed mutations, allowing the detection of invalid mutations such as the modulo operator of mutant 4. Mutating the multiplication, that is, the binary operator `'*'` on line 6, is achieved by changing the `'*'` to `'+,-,/'`. As the location information is preserved, the mutants are easily represented in the original source code. Tools mutating the AST will do so by either iterating through the entire AST or by using AST matchers that provide a list of all candidate nodes based on a set configuration, for example, BinaryOperator.

Listing 2: AST Example

```
1  |-FunctionDecl 0x7fac9d87b710 <main.cpp:1:1, line:3:1> line:1:7 used f 'float (float, float)'
2  | |-ParmVarDecl 0x7fac9d87b5b8 <col:9, col:15> col:15 used a 'float'
3  | |-ParmVarDecl 0x7fac9d87b638 <col:18, col:24> col:24 used b 'float'
4  | `-CompoundStmt 0x7fac9d87b8a8 <col:27, line:3:1>
5  |   `-ReturnStmt 0x7fac9d87b898 <line:2:5, col:16>
6  |     `-BinaryOperator 0x7fac9d87b878 <col:12, col:16> 'float' '*'
7  |       |-ImplicitCastExpr 0x7fac9d87b848 <col:12> 'float' <LValueToRValue>
8  |       | `-DeclRefExpr 0x7fac9d87b808 <col:12> 'float' lvalue ParmVar 0x7fac9d87b5b8 'a'
   'float'
9  |       `-ImplicitCastExpr 0x7fac9d87b860 <col:16> 'float' <LValueToRValue>
10 |         `-DeclRefExpr 0x7fac9d87b828 <col:16> 'float' lvalue ParmVar 0x7fac9d87b638 'b'
   'float'
```

In Listing 3, we show our mutation example in the LLVM IR format. Lines 2 to 9 represent '**return** a *b;' As this is closer to the machine code, less information from the original source code is preserved, for example, the variable names *a* and *b* are converted to numbers *%0* and *%1*, and low-level instructions such as the allocation (alloca) of variables are introduced.

Mutating the multiplication in the LLVM IR code means mutating the *fmul* statement to *fadd, fsub* and *fdiv*. An LLVM IR mutation testing tool usually works in two steps. It first creates a list of mutation points (specific LLVM instruction and its operands). Next it creates a new LLVM IR version for each mutant by applying the mutant to the mutation point.

Listing 3: LLVM IR Example

```
1  define float @_Z1fff(float %0, float %1) #0 {
2      %3 = alloca float, align 4
3      %4 = alloca float, align 4
4      store float %0, float* %3, align 4
5      store float %1, float* %4, align 4
6      %5 = load float, float* %3, align 4
7      %6 = load float, float* %4, align 4
8      %7 = fmul float %5, %6
9      ret float %7
10 }
```

As there are many more instructions at the LLVM IR-level, they present more opportunities for mutations. However, mutations can exist in the LLVM IR code that cannot be achieved by changing the original source code [30]. As no conversion to the original code is possible, there is no traceability and no easy feedback for the developers. Additionally, not all mutation opportunities from the source code are available at the LLVM IR-level due to optimisation in the conversion to LLVM IR. In short, LLVM IR mutations provide a different set of mutants than mutating at the AST-level.

2.5 | Existing LLVM and Clang mutation testing tools

Below we discuss the most prominent mutation testing tools that are based on Clang and/or the LLVM IR. Table 2 lists them in alphabetical order with their features and optimisations. For each of them, we briefly explain which optimisations are incorporated and refer to quantitive evidence if present.

AccMut (IR-based): AccMut [31] is an LLVM IR mutation testing tool that reduces redundant execution statements anywhere in the program by analysing the original and mutated program. It identifies the redundant statements by inspecting the (local) state of both programs. When they are identical, all following statement executions are identical and redundant until the next different statement. They have demonstrated an average speedup of 8.95× over a mutant schemata approach [31].

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Clang and LLVM IR mutation testing tools.

Tool name	Mutation optimisation	Mutation level	Mutation operators
AccMut	Mutant schemata, modulo states	LLVM IR	AOR, ROR, LCR, SDL, …
CCmutator		LLVM IR	Concurrency mutation operator
Dextool mutate	Distribution, Mutant schemata	AST	AOR, ROR, LCR, SDL, UOI
	Exclude unreached mutants via code coverage		
Mart	Trivial compiler equivalence	LLVM IR	Operator groups
MuCPP	Reduced mutants Set	AST	Class level mutants
Mull	Limit total mutants based on call-depth	LLVM IR	LLVM fragments
SRCIROR	Trivial compiler equivalence	AST	AOR, LCR, ROR, ICR
	Exclude unreached mutants via code coverage	LLVM IR	

CCmutator (IR-based): CCmutator [32] is an LLVM IR mutation testing tool specifically designed to mutate concurrency constructs.

Dextool mutate (AST-based): Dextool is an open-source framework created for testing and static analysis of (often safety-critical) code. The Dextool framework is used within industry, for example within Saab Aeronautics. One of the plugins in the framework is Dextool mutate. It was developed with a heavy emphasis on the reporting part of mutation testing in order to better understand the output of mutation testing and to gain more insight into the project under test.

Dextool mutate works by (textually) inserting mutants into the source code one at a time after analysing the abstract syntax tree (conveniently available via Clang) for points to mutate.

Dextool mutate allows users to provide scripts and special flags in order to compile and test projects with the explicit intention to scale for more and bigger (industrial) projects. Dextool mutate supports a distributed setup as the mutants are stored in a central database. Multiple nodes can then access the database and execute a subset of all mutants. During this study, we created a proof-of-concept schemata plugin for Dextool mutate to enable mutant schemata. This plugin was requirement for our experiment in order to execute the mutant schemata approach on the Saab Case as they where already utilising Dextool.

The steps in the Dextool mutate schemata plugin are identical to our standalone tool as described in this paper. The results and timings from the Saab Case were gathered using the Dextool mutate schemata plugin. As a result, the creators of Dextool created a proper implementation for mutant schemata into the tool.[1,2]

Mart (IR-based): Mart [33] is an LLVM IR mutation testing tool that currently supports 18 different operator groups (with 68 *fragments* and 816 operators). These operator groups match against the LLVM IR syntax to create the mutants. Additional operator groups can be implemented by the user to further extend its capabilities. Mart has an in-memory implementation of trivial compiler equivalence to eliminate equivalent and duplicate mutants [34].

MuCPP (AST-based): MuCPP is a Clang-based mutation testing program that generates mutants by traversing the Clang AST and storing the mutants in different branches using a version control system [35]. MuCPP implements mutations at the class level. These include mutations related to inheritance, polymorphism and dynamic binding, method overloading, exception handling, object and member replacement, and more. The study is aimed at reducing the total number of mutants that need to be executed by eliminating so-called unproductive mutants. These include equivalent mutants, invalid mutants, easy-to-kill mutants and mutants in dead code. MuCPP demonstrates that Clang can be used for implementing mutant analysis and generation at the AST level. The study lists the speedup gained from the reduced mutation set but does not list the overhead impact of the generation and analysis of the mutants using the Clang framework. It does not implement other optimisation techniques, so there are no detailed measurements of the potential reduction in compilation and execution overhead using the Clang framework, nor the overhead the implementation of such techniques using the Clang framework might cause.

Mull (IR-based): Mull is an open-source mutation testing tool[3] that modifies fragments of the LLVM intermediate representation (LLVM IR). It only needs to recompile the modified fragments in order to execute the mutants,

[1] https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate/contributors.md.
[2] https://github.com/joakim-brannstrom/dextool/blob/master/plugin/mutate/doc/design/notes/schemata.md.
[3] https://github.com/mull-project/mull.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

keeping the compilation overhead low [30]. As Mull modifies LLVM code, it is compatible with all programming languages that support compilation to LLVM IR, such as C, C++, Rust and Swift. Mull includes a *do-fewer* optimisation where you can limit which mutants are executed to only those mutants that are within a certain call-depth starting from the test case. The study reports on the total runtime for the optimised mutation tool but does not provide details for the runtime of the individual steps nor is the tool contrasted to a traditional, unoptimised approach. While the tool certainly provides a speedup for mutation testing, the lack of detailed measurements makes it difficult to estimate where the advantages lie and where overhead might occur.

SRCIROR (AST or IR-based): SRCIROR [36] is a toolset for performing mutation testing at the AST level or at the LLVM IR level. Both variants implement the *AOR, LCR, ROR, ICR* mutation operators. When mutating the AST, SRCIROR uses the AST matchers from the Clang LibTooling library to search for candidate mutation locations in the AST. It then generates a different source file for each of the generated mutants. When mutating the LLVM IR, SRCIROR creates a list of mutation opportunities (specific LLVM instruction and one of its operands) within the generated LLVM IR code of the project. SRCIROR then creates a mutated version for each of these mutation opportunities. SRCIROR allows to filter out unreachable mutants based on code coverage metrics. It also allows to filter out some equivalent mutants using trivial compiler equivalence [34].

Summary. There is a lot of evidence that optimisation techniques for mutation testing can make orders of magnitude improvements for languages outside of the C language family. Moreover, the current state-of-the-art demonstrates that mutant analysis for the C language family is possible on top of the LLVM and Clang compiler framework. However, the extent to which the various optimisation strategies allow reduced compilation and execution overheads is unknown. In particular, there exist no detailed estimates nor measurements on two of the most advanced techniques (*mutant schemata* and *split-stream mutation testing*) for the C language family.

3 | PROOF-OF-CONCEPT TOOL

In this paper, we investigate to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. For this, we built a proof-of-concept incorporating five strategies: exclude invalid mutants, exclude unreachable mutants, mutant schemata, reachable schemata and split-stream. The proof-of-concept is available under an open source license on Code Ocean together with a description on how to use it.[4] We refer the interested reader to a separate artefact paper describing the implementation details and how other researchers can build upon our work [37]. Below we provide a high-level overview of the five strategies implemented.

The goal of these five optimisation strategies is twofold: eliminate compilation overhead and reduce execution overhead. The different strategies build on each other and successively introduce more optimisations.

1. The first compilation overhead comes from the invalid mutants, which cause compilation errors hence introduce wasted compilation cycles.
2. The second compilation overhead comes from individually compiling each mutant, which is computationally expensive. A mutant schemata strategy eliminates this overhead by compiling all mutants at once, drastically reducing the compilation time.
3. The first execution overhead comes from the unreachable mutants. These mutants are never reached by the test suite, hence will always survive. We explicitly label these mutant as 'unreachable' and 'survived' to indicate that none of the existing test cases are executing them.
4. The second execution overhead stems from the observation that not all test cases reach each mutant. In order for a test case to potentially kill a mutant, that test case needs to first reach and execute the target mutant. If the test case does not reach the mutant, we know that it will never be able to kill it, hence we know its result before we execute it. We implemented a detection algorithm that avoids this redundant execution by detecting which test case reaches which mutant and thus only executes a subset of test cases for each mutant.

[4] https://codeocean.com/capsule/3514968/tree/v1.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5. The final optimisation strategy we looked into is the split-stream mutation. This strategy exploits the state-space information so that the execution of each mutant can start from the mutation point itself instead of always starting the execution at the beginning of the test suite. This essentially cuts the amount of statements that need to be executed in half.

We discuss each strategy, their differences and similarities, the potential impact on compilation and execution time and provided detailed measurements of the steps in the optimisation strategies. The general steps of the optimisation strategies are visualised in Figure 2. We explain each of the optimisations by its steps, starting at the bottom of the image to the top.

Generate mutants. (Bottom boxes in Figure 2.) The generation of the mutants is done independently of the optimisation strategies. This allows us to use the same mutants, ensuring a fair comparison of the strategies. This is represented by its inclusion in every pillar in Figure 2.

Currently, for our proof-of-concept tool, we have implemented the relational operator replacement (ROR), arithmetic operator replacement (AOR) and logical connector replacement (LCR) (see Table 1). This is a representative selection of all possible mutation operators, sufficient to demonstrate the feasibility of Clang based optimisation strategies. We discuss this limitation under Section 7.

Our proof-of-concept tool relies on the LibTooling library of Clang and its ability to iterate through all declarations, statements and expressions from the abstract syntax tree (AST). For each of these declarations, statements and expressions, the implemented mutation operators create the corresponding mutants. As an example, the AOR mutation operator when encountering the multiplication '$*$' operator residing in the binary expression '$a*b$' creates mutants where the multiplication is replaced by the '$+$', '$-$', '$/$' and '$\%$' operators, respectively. To ensure that we know where to replace the original operator with the mutated one, we store the filename in which the mutant occurs, the offset to the beginning and the end of the original operator, and the mutant itself. For each executed mutant, we store whether or

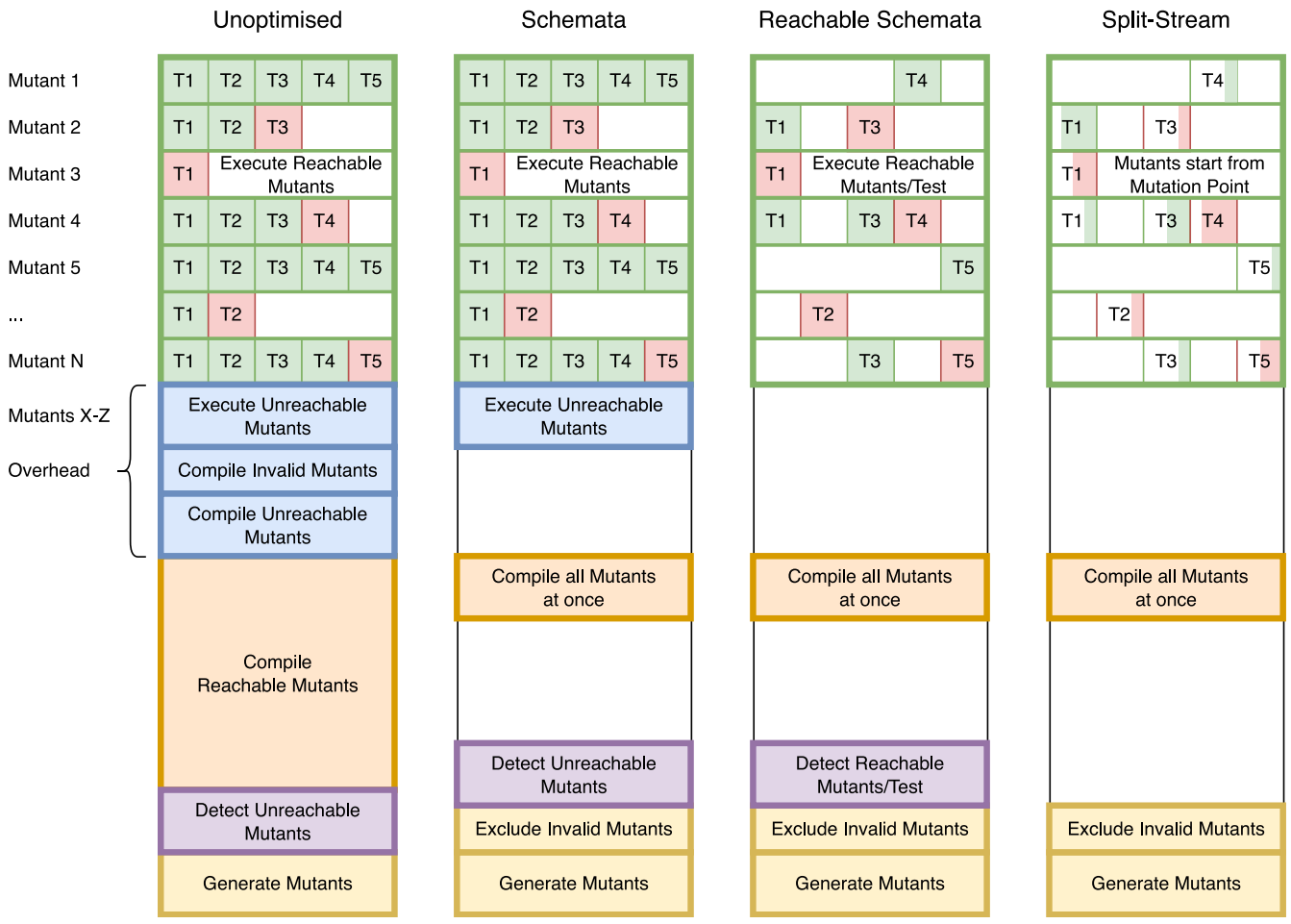

FIGURE 2 Implementation strategies with algorithm steps (first step at the bottom). Each strategy improves on the previous one by reducing compilation and/or execution overhead.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

not the mutant was reached, how long it took to execute, and whether or not the mutant was killed. As it is possible that some mutants will cause an infinite loop, the user can set a maximum execution time for each mutant. If the test suite is not able to finish executing within this time, we stop the test execution and say that the mutant *timed out*. Mutants that are timed out can be considered as killed, as they would also time out in a continuous integration test. This information can then be extracted to form a report to inform developers where and which mutants survived the test suite.

3.1 | Unoptimised mutation testing

In a traditional, unoptimised mutation testing setting each mutant is inserted, compiled and executed separately. Some mutants, however, can be located in uncovered code. Such mutants are unreachable and are unable to infect the program state; thus, they can never be killed. In a realistic scenario, one would first run a code coverage technique to determine which mutants are located in uncovered code and instantly label them as survived. As the purpose of this paper is to gain more insight into the potential speedups of our optimisation techniques, we do generate and execute them in order to gain more insight into the kind of overheads these unreachable mutants introduce.

For detailed measurements, we timed the generation of the mutants, the compilation of the mutants, and the execution of the mutants. For the mutants themselves, we make a distinction between the invalid mutants (i.e., mutants that cause compilation errors) and the valid mutants. We further divide these valid mutants into the unreachable mutants (i.e., mutants that no test case reaches), and the reachable mutants.

We implemented the unoptimised mutation testing approach to obtain a baseline for the mutation analysis and to verify the correctness of our optimisation strategies.

Detect unreachable mutants. In order to determine which mutants are completely unreachable by the test suite, we instrument the code base with a wrapper on the mutatable locations. Running the test suite with the instrumented code base then provides us with a list of mutants that are reachable by the test suite. From this, we can deduce the completely unreachable mutants. While existing code coverage tools can be modified to achieve the same results, we utilised our own built-in instrumentation tool as it does not require additional setup and maintenance and does not create additional requirements for to the build environment. By utilising our own specialised tool, we can tailor it to our needs, choose a fitting output format and avoid any overhead from features of a general code coverage tool that we do not need.

Compile mutants. After the generate mutants phase, each mutant needs to be individually inserted into the original code base and compiled before it can be executed. We measured the compilation time for each mutant and divided them into three categories: reachable, unreachable and invalid mutants. This allows us to quantify the overhead caused by the unreachable and invalid mutants.

All three can be seen in the first pillar, that is, unoptimised, in Figure 2. It is important to note that we do not clean the build environment between the different mutants. We use consecutive builds to speed up the compilation, as the compiler then only needs to re-compile the files that are impacted by the mutant.

Execute mutants. After generation and compilation, the valid mutants need to be executed. For each mutant, the entire test suite needs to be run. The test cases of the test suite are represented by T1 to T5 in Figure 2. When a test case fails, represented by the red colour, this means that the mutant is killed. When none of the test cases fail, the mutant survives, represented by the green colour. To optimise the mutation testing execution, we rely on the early-stop principle, for which the test suite execution stops when the first test case kills the mutant. In the unoptimised approach, the entire test suite is run for each unreachable mutant. We represented this with mutants X–Z in Figure 2.

3.1.1 | Expected performance

For the unoptimised approach, we estimate the performance via Formula 1. The formula consists of three phases: the generate mutants, compile mutants and execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches but its impact should be negligible. We do not include the detection of unreachable mutants as it is not part of an unoptimised approach. We only need it to distinguish between the unreachable and reachable mutants. The total compilation time is subject to the amount of reachable, unreachable and invalid mutants. From our measurements, we have seen that invalid mutants can take up to 10% of the total

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

execution time. The total test suite execution time is only subject to the amount of reachable and unreachable mutants as invalid mutants cannot be executed. Note that the compilation time of consecutive builds is lower than a clean build and that the test suite execution time varies depending on where, if at all, the mutant is killed due to the early-stop mechanism.

$$\begin{aligned} & t_{mutant\_generation} \\ + \; & t_{compilation} && *(reachable\_mutants + unreachable\_mutants + invalid\_mutants) \\ + \; & t_{test\_suite\_execution} && *(reachable\_mutants + unreachable\_mutants) \end{aligned} \tag{1}$$

3.2 | Mutant schemata

The mutant schemata strategy compiles all mutants at once, essentially eliminating the compilation overhead [38]. Previous studies with mutant schemata have shown that this optimisation approach can provide an order of magnitude improvement on a full mutation analysis. Untch et al. reported a preliminary speedup of 4.1 [38]. In a later study, Wang et al. confirmed these findings and reported a speedup between 6.46 and 14.00 on systems written in Java [31]. However, we found no specific studies investigating the speedups of mutant schemata for the C language family, hence, in this study, we provide detailed measurements. We timed the generation of the mutants, the compilation of all the mutants at once, and the execution of the mutants. We can then compare this to the unoptimised approach to gain insights into its speedups and potential overheads. For a fair and complete comparison, we again make the distinction between the unreachable and reachable mutants.

Exclude invalid mutants. As all mutants are inserted into a single compilation unit, all generated mutants need to compile. If even a single mutant causes a compilation error, the complete mutation analysis fails. This is especially challenging for statically typed languages with many interacting features and unforgiving compilers (C, C++, …). This is where the Clang compiler front-end can be used for ensuring that all injected mutants will compile. As Clang has access to all the type information from the project, our program can verify, during the generate mutants phase, that a newly created mutant is compile-time correct. For this, we rely on the semantic analyser of Clang. When traversing the AST, we modify a node and ask the semantic analyser of the compiler if the modified node is syntactically correct in its given context. We are essentially going though the different steps of a compiler (but stop before the compiler performs optimisations, and before it exports to binary code). Listing 4 shows the relevant Clang code for building a modified binary operation with its verification. Line 6 calls the semantic analyser and line 7 verifies whether it would cause compilation errors. If not it is considered invalid and labeled as such. This allows us to avoid a full compilation of the program for every mutant. To ensure that we exclude only the invalid mutants, we verified that the mutants we labelled as invalid are the same mutants that actually caused compilation failures in the unoptimised baseline. As this step is tightly integrated with the generation of the mutants, we measured this together with the generation of the mutants.

Listing 4: Exclude Invalid (Binary) Mutants using Clang Semantic analyser

```
1 clang::Sema* sema;  // Clang semantic analyser
2
3 bool makesSense(clang::BinaryOperator* binOp, clang::BinaryOperatorKind Opc) {
4     clang::Expr* lhs = getIntendedLHS(binOp->getLHS());
5     clang::Expr* rhs = getIntendedRHS(binOp->getRHS());
6     clang::ExprResult expr = sema->BuildBinOp(sema->getCurScope(), binOp->getExprLoc(), Opc,
          lhs, rhs);
7     return (!expr.isInvalid() && expr.isUsable());
8 }
```

Detect unreachable mutants. The detection of the unreachable mutants is identical to the one for the unoptimised approach.

Compile mutants. Instead of compiling each mutant separately, the mutant schemata strategy compiles all mutants at once. This means that all mutants are inserted into the code, each mutant being guarded by a conditional statement allowing the activation of individual mutants at run-time. The result is a single code base and only one compilation is needed, drastically reducing the compilation time. We visualised this in Figure 2 by using a smaller

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

'compile all mutants at once' block. A specific mutant is activated by using an external environment variable. For this, additional code, that is, the external variable, needs to be added to each file of the project (see line 1 in Listing 5). Listing 5 shows the mutant schemata version of the original '$a*b$' example. We use the ternary operator, the short-handed version of an *if* statement, to allow nesting the mutants inside the condition of *if* statements.

In the example, the '%' operator gives an '`invalid operands to binary expression`' error; therefore, we cannot compile our program unless we remove this mutant. As we exclude invalid mutants during generation, these do not show up in the mutated code base.

Listing 5: Mutant Schemata Example

```
1  extern int MNR; // prepended external variable, allowing selection of active mutant
2
3  // mutated ''return a * b;'' statement using the ternary operator:
4  float f(float a, float b) {
5      return (MNR == 1 ? a + b :
6              (MNR == 2 ? a - b :
7              (MNR == 3 ? a / b :
8              (MNR == 4 ? a % b : a * b))));
9                          ~~~~~~~Invalid operands to binary expression
10 }
```

Execute mutants. As a final step, all the mutants need to be executed. This can be done by running the executable for each valid mutant and only changing the environment variable. We again make the distinction between the reachable mutants, represented by mutants 1 to N, and the unreachable mutants represented by mutants X-Z in Figure 2.

3.2.1 | Expected performance

For the mutant schemata approach, we estimate the performance via Formula 2. The formula consists of three phases: the generate mutants, the compile mutants and execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches, but its impact should be negligible. We do not include the detection of unreachable mutants as it is not part of the schemata approach. We only need it to distinguish between the unreachable and reachable mutants. As the strategy removes the compilation overhead by only compiling the project once, instead of for each mutant, a drastic speedup is to be expected. Therefore, we no longer need to multiply the compilation time by the number of mutants. Note that the compilation time will be slightly longer, as there is more code that needs to be compiled. The execution part of the formula stays the same, as we still need to execute the test suite for each mutant. The test suite execution time for each mutant will, however, also be slightly longer as there is more code that needs to be executed due to the *if* statements of the ternary operator in order to activate the correct mutant.

$$\begin{aligned} & t_{mutant\_generation} \\ + \; & t^{schemata}_{compilation} \\ + \; & t^{schemata}_{test\_suite\_execution} \; * (unreachable\_mutants + reachable\_mutants) \end{aligned} \tag{2}$$

3.2.2 | Schemata implementation

We implemented the mutant schemata technique using the ternary operator, the short-handed version of an *if* statement. We do note that the most common case for this implementation is the worst-case scenario. Most of the times no mutant in the statement will be activated, causing an evaluation of all the if statements before the original statement can be executed. This creates a fixed overhead per mutant, though we expect this overhead to be limited. The advantages of this approach is that it simplifies and streamlines the implementation of the schemata technique. It allows nesting the mutants inside the condition of an *if* statement. This can be seen in Listing 6. The body of the if statement can also directly be mutated inside the expression. Another advantage of the ternary implementation is that it causes no local scope and can thus mutate assign statements without additional analysis.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Listing 6: Mutating *if* Statements

```
// original
if (a > b) {/*body*/}

// mutated using ternary operator
if  (MNR == 1 ? a <  b :
     (MNR == 2 ? a <= b :
     (MNR == 3 ? a == b :
     (MNR == 4 ? a >= b : a > b)))) {/*mutated body*/}
```

An alternative implementation is possible by using a switch case approach. We can nest the switch inside the condition of the if statement using statement expressions,[5] as seen in Listing 7. While statement expressions are supported by default in `gcc` and Clang, they are not supported by the C standard. We are thus strongly discouraged to use them if we want to enable mutation testing in an industrial setting where it is not always guaranteed that the used compiler will support the GNU extension. The easiest workaround is putting the statements in a function and calling said function in the condition of the if statement. For each mutated statement, a new function will then have to be created.

We can also implement the schemata technique using a global switch, which has its own set of challenges. Firstly, switch cases have a local scope. This means that when mutating an assign statement using the switch case approach, the variable will first need to be defined outside the scope of the switch case. Secondly, code explosion will occur as the body of the if statement needs to be repeated for each case, see Listing 7. The code explosion can be reduced by creating functions for the body.

We can then either mutate the body inside the default case using additional switch cases, or append the cases of the mutated body to the existing switch. The latter reduces the amount of switch cases, but would cause a code explosion.

Listing 7: Mutating *if* Statements using switch-cases

```
// original
if (a > b) {/*body*/}

// mutating using nested switch case
if (({bool r;
      switch (MNR) {
          case 1:   r = a < b;  break;
          case 2:   r = a <= b; break;
          case 3:   r = a == b; break;
          case 4:   r = a >= b; break;
          default:  r = a > b;
      }
      r;}){/*mutated body*/}
}

// mutating using global switch case
switch(MNR) {
    case 1:  if (a >  b) {/*body*/} break;
    case 2:  if (a >= b) {/*body*/} break;
    case 3:  if (a == b) {/*body*/} break;
    case 4:  if (a >= b) {/*body*/} break;
    default: if (a >  b) {/*mutated body*/} break;
}
```

3.3 | Reachable schemata

The previous strategy essentially eliminated the compilation overhead. This causes the test suite execution to become the most time-consuming part of the mutation analysis. While eliminating the completely unreachable mutants does

[5] https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

speed up the mutation testing analysis, the reachable schemata strategy aims to use a more fine-grained approach to reduce most of the execution overhead. The reachable schemata strategy reduces the test suite scope to only those test cases which reach the mutant, effectively reducing the execution overhead. We time the generation of the mutants, the compilation of all the mutants at once and the execution of a reduced test suite for each of the mutants. We can then compare this to the unoptimised and mutant schemata strategies to gain insights into its speedups and potential overheads.

Exclude invalid mutants. As the reachable schemata strategy is an optimisation based on the regular mutant schemata approach, we need to exclude the invalid mutants as well. This is done in the same way as with the mutant schemata approach.

Detect reachable mutants/test. In order to determine which mutants are reached by which test cases, we instrument the code with a wrapper on the mutatable locations we implemented ourselves. Since the Clang AST level has facilities available to enumerate the mutable locations, we relied on those to just enable traceability between a test and the operators actually executed. We store each mutatable location a test run reaches. Running the instrumented code base for each test case provides us with lists of mutants that are reachable by each test case. This results in a reduced set of test cases that need to be executed for each mutant, resulting in a speedup without information loss.

The best results are obtained by using fine-grained test selection and running each test case individually. The speedups of this technique depends on the test driver used for each project. Some test drivers might only be able to run individual modules instead of individual test cases. Speedups from module-grained test selection will be lower than from fine-grained test selection. Our projects were all compatible with fine-grained test selection on a test-by-test case basis.

Compile mutants. The compilation of the mutants is identical to the one for mutant schemata. We do note that an extra optimisation can be done by only inserting and compiling the reachable mutants in this phase. We did not yet implement this optimisation in our proof-of-concept.

Execute mutants. As a final step, all the mutants need to be executed. Instead of running the entire test suite for each mutant, we now only need to run those test cases that reach the mutated locations. In Figure 2, we see that mutant 1 is not reachable by the entire test suite; therefore, no tests are executed for the mutant. For mutant 2, the second test case is not executed as that test case does not cover mutant 2.

3.3.1 | Expected performance

For the reachable schemata approach, we estimate the performance via Formula 3. The formula consists of four phases: the generated mutants, the detect reachable mutants, the compile mutants and the execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches but its impact should be negligible. We included the detection of the reachable mutants/test as it is a part of the reachable schemata approach in order to gather the reachable mutants to the test case relationship. We estimate its time impact at a single compilation and execution of the test suite. This is immediately compensated when there is more than one unreachable mutant. The compilation part again consists of a single compilation as it is an extension of the mutant schemata approach. For the execution part, only the reachable mutants are considered, together with a decoupling factor. This factor is a percentage based on the reduction in average mutants reachable by each test case. The more coupled a project is, the less effective this strategy will be. The less coupling there is (especially in combination with mocking), the more effective this strategy will be.

From our measurements, we have seen that the average number of mutants reached per test is between 10 and 20% of all valid mutants. This implies that our technique can reduce the number of tests needed to execute per mutant between 5× and 10×. We therefore also expect a speedup between 5× and 10×.

$$\begin{aligned} & t_{mutant\_generation} \\ + & \left(t^{schemata}_{compilation} + t^{schemata}_{test\_suite\_execution}\right) \\ + & t^{schemata}_{compilation} \\ + & t^{schemata}_{test\_suite\_execution} * reachable\_mutants * decoupling\_factor \end{aligned} \tag{3}$$

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

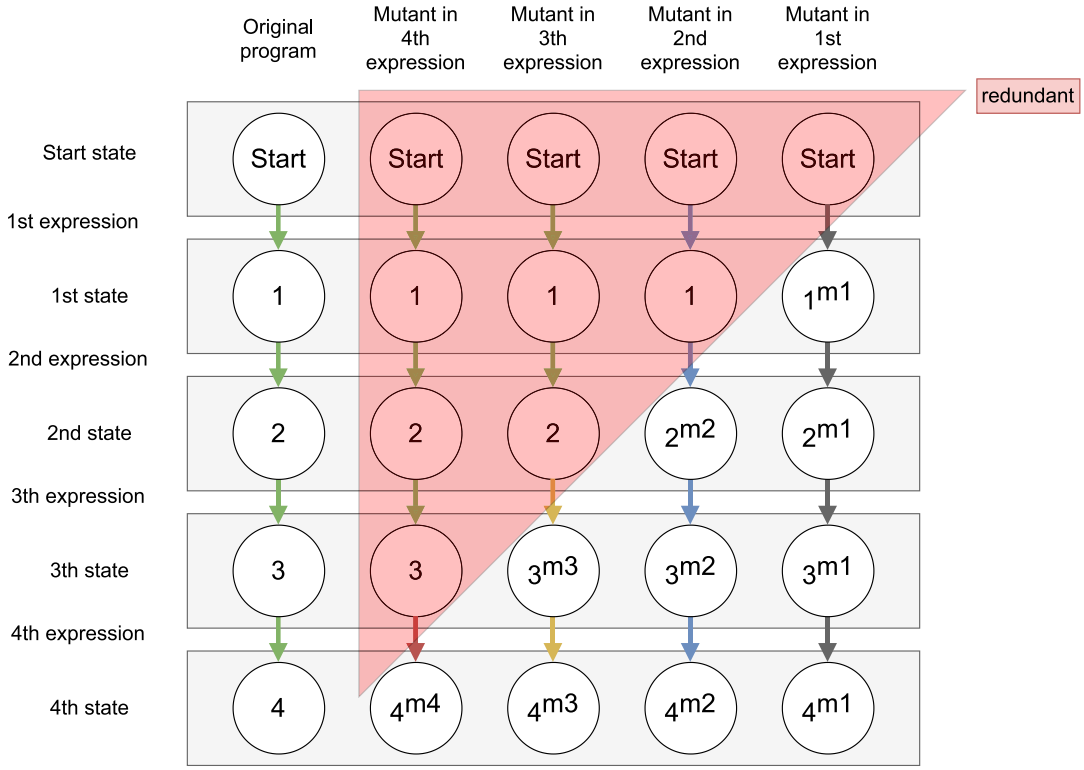

FIGURE 3 Trace of original and mutated programs.

3.4 | Split-stream mutation testing

The split-stream mutation testing strategy reduces the execution overhead of mutation testing even further. Instead of letting each mutant initiate execution from the start of the program, each mutant is started from the mutation point itself. This can be achieved by exploiting the state-space information. Previous research has shown an average speedup of 3.49× of split-stream mutation testing over mutant schemata by mutating the LLVM IR [31].

We explain the overhead and general idea of the split-stream mutation testing approach using Figure 3. Here, we visualised the execution trace of the original, unchanged program, as a long vertical trace of states, each representing the execution of a single instruction. The trace of each mutant looks identical to the original mutant up until the mutated expression. In Figure 3, we show mutant 4 with a mutation in the 4th expression, mutant 3 with a mutation in the 3rd expression. The trace only deviates from the original trace after the mutated expression. All of the states up until the mutated state are in fact redundant, as we already know them from the execution of the original program. We highlighted these redundant states and expressions in red in Figure 3.

Exclude invalid mutants. As this strategy is an optimisation from the regular mutant schemata approach, we need to exclude the invalid mutants as well. This is done identically as with the mutant schemata approach.

Compile mutants. The compilation of the mutants is identical to the one for mutant schemata. However, additional code needs to be instrumented in the code base in order to exploit the state space and start the mutants from their mutation point instead of from the start of the program.

Execute mutants. In order to exploit the state space, we start the split-stream mutation analysis only once instead of for each mutant as with mutant schemata. We explain the split-stream mutation testing process using Figure 4.

When we start the split-stream mutation analysis, no mutant is active. The first mutant the program encounters is not yet activated, so the program forks the entire program and pauses the original one. The forked program is then in the exact same state as the original program, essentially this is a duplicate of the state-space. In the forked program, we activate the encountered mutant. We let the forked program execute, only taking into account the active mutant and store its results at the end of its execution. We then continue our first program, until it encounters another mutant that

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

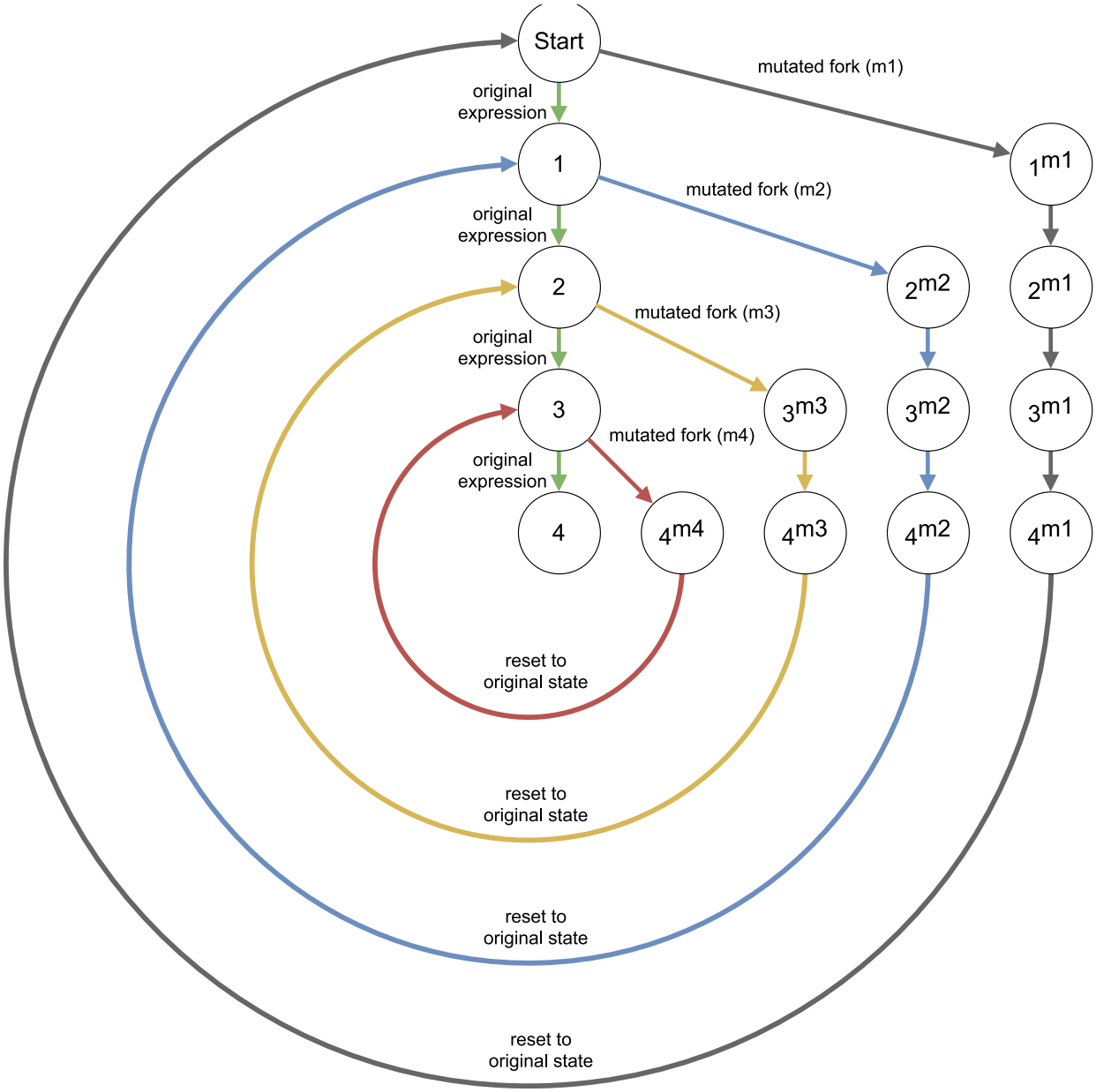

FIGURE 4 Split-stream mutation process.

is not yet activated. Here, the process is repeated, a fork is created, the mutant is executed, and its results are stored. This goes on until the original program reaches its end state.

Like with the mutant schemata strategy, our proof-of-concept tool needs to decide what needs to happen at each mutation position. Instead of only deciding whether or not a mutant needs to be active or not, it now also needs to decide *when* it needs to fork the process. For this, we instrumented additional code at each mutation point. We explain the general concept using the C++ example in Listing 8.

When we start our program with all the mutants in it, no mutant is ever be activated. The original program is only responsible for forking newly encountered mutants and storing their results.

When a mutant is encountered in the main program, for example, on line 16, then it continues to line 7, where it verifies whether or not it already encountered that mutant. If it did, it will continue the program, otherwise, it will fork the program and wait for the forked program to finish, on lines 8 and 9. The forked program will only take into account the active mutant and run till its end. When the forked program encounters a mutant, it verifies if it is the mutant that the process was forked for, if so it will execute it, otherwise it will execute the original code. The forked process will not create additional forks. The main program will then store the result (exit code) of the forked program, add the mutant to the already executed list and continue until it encounters a new mutant, on lines 9 and 10.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Listing 8: Split-Stream Mutation Testing Example

```
1  extern list<int> MNR_list;  // already activated mutants
2  extern int MNR;             // active mutant
3
4  bool split(int mnr) {
5      if (/* this is the fork */) { return mnr == MNR; }
6      else { /* this is the main program */
7          if (MNR_list.contains(mnr)) {return false;}
8          else { /* fork, and set MNR to mnr in forked process*/ }
9          /* wait for fork to complete and store results */
10         MNR_list.insert(mnr);
11     }
12     return false;
13 }
14
15 float f(float a, float b) {
16     return (split(1) ? a + b :
17             (split(2) ? a - b :
18             (split(3) ? a / b : a * b)));
19 }
```

The split-stream mutation testing approach naturally ensures that unreachable mutants are not executed. The approach will never create forks for mutants that it will not reach. By executing each of the tests separately, only those mutants reachable by the test case will be executed for that test. This is similar to the reachable schemata approach, with the added benefit that there is no separate compilation and/or test suite execution run necessary in order to extract the reachable mutants.

External dependencies. As we start the mutants from their mutation points, we now also need to ensure that all of the project dependencies, for example, files and databases, are in their correct state. If a test changes the state of an external dependency (a file, a database, …), the mutation point needs to copy the existing state at the mutation point, continue the forked process and when it returns to the mutation point restore the original state. The protocol of such external dependencies complicates matters to a large degree. For example, a file needs to be opened and closed, a database transaction needs to commit or abort. If this protocol is not followed the test will fail because the system under test in in a corrupted state. This will change the mutation score, which is something that must be avoided at all cost

We therefore built in hook functions to copy and restore the state, which can be specialised for each project. We instantiated these hook functions to automatically reset local files to the state they were in for each of the mutation points via a local `git` repository and the `system('git reset –hard HEAD 1')` command.

3.4.1 | Expected performance

For the split-stream mutation approach, we estimate the performance via Formula 4. The formula consists of three phases: the generate mutants, the compile mutants and the execute mutants phase. We included the generation of the mutants in the formula as this step is necessary and identical for all approaches but its impact should be negligible. We no longer need to detect the reachable mutants/test as it is inherent to the split-stream approach. The compilation part again consists of a single compilation, which should be slightly longer than the compilation of the schemata approach. The execution time is impacted negatively as the forking of the process will take some time. As the forking process works on a copy-on-change principle, only the memory that is changed will be copied. This ensures that the impact of the forking is kept at a minimum. The benefit of this strategy is that we no longer need to execute the redundant code. A mutant that is located at the very end of the execution will now only take very little time to execute. This is in contrast to the original mutation testing where it would have needed to start its execution from the very beginning. Mutants located in the front will only have a small improvement in their execution time. In general, one could thus assume that this strategy would contribute to an additional 2× speedup. Similar to the reachable schemata approach, executing the test suite on a test-by-test case should yield the best results. Hence, we have the same decoupling factor.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

$$
\begin{aligned}
& t_{mutant\_generation} \\
+\ & t^{split\_stream}_{compilation} \\
+\ & t^{split\_stream}_{test\_suite\_execution} * reachable\_mutants * decoupling\_factor/2
\end{aligned}
\tag{4}
$$

4 | EXPERIMENTAL SET UP

This paper presents a feasibility study, investigating to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. We measure the speedup from two perspectives (compilation time and execution time) assessing four optimisation strategies. This gives rise to the following research questions:

- *RQ1: How much speedup can we gain from mutant schemata?*
- *RQ2: How much speedup can we gain from split-stream mutation testing?*
- *RQ3: How much speedup can we gain from eliminating invalid mutants?*
- *RQ4: How much speedup can we gain from eliminating unreachable mutants?*

To answer these research questions, we measure the impact of the optimisation techniques, which delays we introduce and how much of the overhead we eliminate. We also collect qualitative evidence on difficulties we encounter when applying each of the strategies.

4.1 | Cases

To investigate the strengths and weaknesses of the Clang-based optimisation strategies, we validate our proof-of-concept tool on four open-source C++ libraries and one industrial component. These cases cover a wide diversity in size, C++ language features used, compilation times, and test execution times as shown in Table 3. To allow other researchers to reproduce our results, we refer to the latest commit id of the version of the project we used for our analysis.

In the first block of the table, we list general details about the project, number of commits, contributors, Lines Of Project Code (LOPC), Lines Of Test Code (LOTC), and the number of test cases the project has.

The second block list the compilation time of the project and the test suite execution time. Projects with a longer compilation time might benefit more from a mutant schemata strategy than projects with a longer test suite execution time.

In the third block, we list how many mutants we generated for each project. As our implementation of the mutant schemata optimisation cannot yet work with mutants in so-called const-expression and/or templates, we excluded these mutants for a fair comparison. We also list the number of mutants per line of production code (LOPC), which indicates how densely or sparsely packed the mutants are. A project with a high density of mutants might introduce delays specific to the optimisation strategy.

In the fourth block, we list the number of valid mutants and the number of invalid mutants, that is, mutants that are killed by the compiler. We also list the number of mutants that are not reachable by the current test suite. Finally, we list the average amount of test cases that actually reach a valid mutant, both in absolute number as well a relative percentage. The reachable mutant schemata strategy uses this information to its advantage to reduce the mutant execution time.

In the last block, we list the number of survived mutants and the number of killed mutants.

4.1.1 | TinyXML2

TinyXML2 (https://github.com/leethomason/tinyxml2 commit id: ff61650517cc32d524689366f977716e73d4f924) is a simple, small, efficient, C++ XML parser that can be easily integrated into other programs. It represents a small-sized project with 3542 lines of production code and 1885 lines of test code (without empty lines and comments). Even though the project is small, we generate 1038 mutants for it. We choose this project deliberately because of its short

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 3 Results: Project details.

	TinyXML2	JSON	Google test	Cpp Check	Saab case
Commits	1052	4312	3840	25,309	Confidential
Contributors	78	213	100	340	
GitHub stars	3.9k	28.4k	24.9k	3.9k	
LOPC	3542	10,955	32,630	98,171	
LOTC	1885	26,586	28,064	159,780	
Test cases	1	88	61	3745	
Compilation time	1.12 s	3 min 52.88 s	6 min 31.34 s	5 min 49.68 s	3 min 08.52 s
Test suite run time	0.11 s	14 min 10.76 s	15.01 s	17.04 s	2.63 s
Generated mutants	1038	3764	4488	61,007	Confidential
Excluded mutants (const, constexpr or templated mutants)	185	3254	1755	5591	
Considered mutants	853	510	2733	55,416	
Mutants/LOPC	0.241	0.047	0.084	0.564	0.233
Valid mutants	716	333	2498	54,643	Confidential
Invalid mutants (killed by compiler)	137	177	235	773	
Completely unreachable mutants	36	0	144	5712	
AVG tests/reachable mutant	0.95	8.46	11.91	402.96	
	94.97%	9.61%	19.53%	10.76%	
Survived mutants	331	33	1028	21,291	Confidential
Killed mutants (excl. timed out)	211	300	1419	24,639	
Timed out	138	0	51	8713	

LOPC, lines of production code; LOTC, lines of test code. LOPC (incl. include files) calculated using cloc (excl. newlines and comments; https://cloc.sourceforge.net).

compilation and execution time (1.12 s and 0.11 s, respectively), as it represents a worst-case scenario for all overhead introduced by the optimisation strategies.

4.1.2 | JSON

JSON (https://github.com/nlohmann/json commit id: 7c55510f76b8943941764e9fc7a3320eab0397a5) is a special case as the entire source code consists of a single header file. This means that any changes to that file will cause a complete rebuild of the entire project, including virtually all test files. Furthermore, all valid mutants are reachable by the test suite. This means that there we cannot expect any speedup from detecting the unreachable mutants.

The JSON project consist of many const, constexpr and templates. As our implementation of the mutant schemata optimisation cannot yet work with mutants in so-called const-expression and/or templates, we need to exclude many mutants from the JSON project.

The test suite of JSON consists of 88 tests with a runtime of 14 min. Two of the tests however cause the majority of this time, that is, test 12 with 1 min and test 80 with 11 min execution time. By limiting the number of mutants to only those mutants that are reached by these tests, the total runtime can be reduced drastically.

4.1.3 | Google test

Google test (https://github.com/google/googletest commit id: f2fb48c3b3d79a75a88a99fba6576b25d42ec528) represents a medium size project with 32,630 lines of production code and 28,064 lines of test code. It is a widely used framework for testing C++ code, so one could expect it to be fairly reliable. This is confirmed by the mutation coverage, from the 2498 valid mutants, only 144 (5.8%) are unreachable by the test suite.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.1.4 | CppCheck

CppCheck (https://cppcheck.sourceforge.io commit id: 8636dd85597acdc1560f7e0bd364c94851bec3b9) is a static analysis tool for C/C++ programs. It aims to detect bugs, undefined behaviour and dangerous coding constructs. It has the most mutants per line of production code and will thus also have the largest negative impact on the duration of its test suite execution. It is the biggest of the projects we analysed with the highest number of mutants. We would like to note that this project has many configurable macros. We ran the project without changing the defaults. Running the project with different macro configurations can lead to different results. This can even increase or decrease the number of unreachable mutants.

4.1.5 | Saab case

The Saab case (https://saabgroup.com/about-company/organization/business-areas/) represents the project from our industrial collaboration. The company develops safety-critical systems and must adhere to 100% MC/DC testing (Modified Condition/Decision Coverage, the coverage criterion adopted for the highest Design Assurance Level (DAL) in accordance to the RTCA-DO178B/C standard). This means that their project is well tested. Due to the project being classified as confidential, some information has been left out and marked *Confidential* in the coming sections. However, the information related to the speedup caused by the use of mutant schemata can be disclosed.

4.2 | Hardware and software set-up

We used the same infrastructure for the analysis of the selected open-source projects. An Intel(R) Core(TM)2 Quad Q9650 CPU, with two 4GB (Samsung M378B5273DH0-CH9) DDR3 RAM modules (for a total of 8GB) and a 250GB Western Digital (WDC WD2500AAKX-7) hard drive. The PC was running Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-29-generic x86_64). Using an SSD should drastically influence the compilation times and negatively impact the total speedups. Using more RAM and/or a faster CPU might influence the compilation time and/or execution time, this however is presumed to be marginal.

The industrial project of Saab Aeronautics ran on their build server. Some of their results could be slightly impacted by background services.

5 | RESULTS AND DISCUSSION

Before we measure the speedup induced by the different optimisation strategies implemented using the Clang front-end, we first analyse the time spent in each of the optimisation phases. The individual timing results can be found in Table 4. In essence, this timing information allows us to assess the significant terms in the performance estimation formulas for the unoptimised configuration (Formula 1; p. 12); the mutant schemata (Formula 2; p. 14 and Formula 3; p. 16) and the split-stream mutation testing (Formula 4; p. 19).

5.1 | Individual timings

Time measurement. We split the mutation analysis up in the afore mentioned phases (i.e., mutant generation, insertion, compilation and execution). We time each of these phases using the unix time command. We utilised the std: chrono function to get the system clock and measure internal execution times of the generation and validation of the mutants. Executing a complete run easily takes several days, hence we only collect one measurement per run.

Generate mutants. The first block of Table 4 confirms that the generate mutants phase is negligible. This phase is necessary and identical for all our investigated optimisation techniques and corresponds to $t_{mutant\_generation}$ in Formula 1 to 4. The four open-source cases illustrate that this phase is orders of magnitude faster than any of the following steps and less than the original compilation time of the projects as listed in Table 3.

Detect (un)reachable mutants. In the second block of Table 4, we list the detection times of the (un)reachable mutants. For the unoptimised and schemata technique, we detect which mutants are reached by the test suite. The reachable schemata approach is more fine-grained and detects which mutants are reached on a test-by-test case.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 Results: Individual Timings.

Strategy	Mutants	TinyXML2	JSON	Google test	CppCheck	Saab case
Generate mutants & validation:						
All		0.09 s	0.64 s	3.79 s	2 min 47.15 s	Confidential
Detect (un)reachable mutants:						
Unoptimised	Reachable	0.11 s	14 min 12.09 s	7 min 11.39 s	7 min 51.60 s	2.63 s
Schemata	/test suite					
Reachable	Reachable					
Schemata	/test case					
Split-stream		0 s	0 s	0 s	0 s	0 s
Compile mutants:						
Unoptimised	Reachable	7 min 17.49 s	21 h 45 min 46.62 s	40 h 21 min 45.49 s	3 days 18 h 52 min 39.52 s	Confidential
	Unreachable	22.79 s	0 s	2 h 19 min 35.25 s	10 h 36 min 20.31 s	
	Invalid	50.83 s	3 h 05 min 23.74 s	3 h 05 min 24.35 s	1 h 01 min 41.15 s	
Schemata	Valid	1.23 s	3 min 54.68 s	6 min 36.04 s	7 min 07.39 s	3 min 02.63 s
Reachable						
Schemata						
Split-stream		1.28 s	3 min 57.27 s	6 min 41.85 s	7 min 13.91 s	N.A.
Execute mutants (excluding timed out):						
Unoptimised	Reachable	1 min 05.60 s	17 h 41 min 49.05 s	9 h 49 m 36.26 s	2 days 05 h 41 min 30.47 s	Confidential
	Unreachable	4.10 s	0 s	36 min 01.25 s	1 day 03 h 02 min 43.46 s	
Schemata	Reachable	1 min 21.84 s	17 h 43 min 28.57 s	9 h 50 min 30.45 s	4 days 22 h 16 min 45.61 s	
	Unreachable	4.81 s	0 s	36 min 04.84 s	2 days 11 h 32 min 05.66 s	
Reachable	Reachable	1 m 13.54 s	1 h 01 m 29.01 s	1 h 46 min 36.16 s	9 h 27 min 30.01 s	
Schemata	/test case					
Split-stream		DNF				
Technique execution overhead:						
Schemata versus	Reachable	17.44%	0.16%	0.15%	120.29%	0.11%
Unoptimised	Unreachable	19.95%	N.A.	0.17%	120.13%	
Timed out mutants:						
Unoptimised		27.60 s	0 s	26 min 00 s	2 days 12 h 30 min 25 s	Confidential
Schemata					5 days 01 h 00 min 50 s	
Reachable						
Schemata						
Split-stream		DNF				

We estimated this phase at $\left(t^{schemata}_{compilation} + t^{schemata}_{test\_suite\_execution}\right)$, a single compilation and test suite execution in Formula 3. The four open-source cases and the industrial case illustrate that this phase is also orders of magnitude faster than any of the following steps.

As the split-stream approach inherently only executes reachable mutants, it does not have a detection delay.

Compile mutants. The third block of Table 4 confirms that the compilation phase takes up a significant amount of time. Additionally, the compilation of the invalid and unreachable mutants is considerable.

We estimated this phase for the unoptimised approach at $t_{compilation} * (reachable\_mutants + unreachable\_mutants + invalid\_mutants)$ in Formula 1.

By moving from the unoptimised approach to the schemata approach, we only need to compile once instead of for all mutants, reducing the compilation time to $t^{schemata}_{compilation}$ from Formula 2 where the multiplication is removed. This essentially removes the compilation overhead as is confirmed by the four open source cases in Table 4. We can also see that the compilation is only slightly longer than the original compilation, as seen in Table 3. This is due to the fact that

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

the injected mutants increase the code base, but this is limited to each function scope, therefore leaving the linking part of the compilation untouched. However, when we introduce more advanced mutation operators and mutate the aforementioned const expressions, we likely need to introduce different functions, causing the compilation time to increase further. Still, we can expect the compilation time to remain drastically decreased from an unoptimised traditional approach. This is also the case for the split-stream mutation testing approach, which in its turn is slightly longer compared to the mutant schemata one, as it has internal functions to regulate the activation and forking of the mutants.

Execute mutants. The fourth block of Table 4 contains the execute mutant phase excluding the timed out mutants. Here, we can see that the execute mutant phase also takes up a significant amount of time. This was expected as we estimated its duration at $t_{test\_suite\_execution} * (reachable\_mutants + unreachable\_mutants)$ for the unoptimised and schemata approach in Formula 1 and 2. The first observation we can make is that the unreachable mutants can have a minimal to large impact on the mutant test execution. The stronger a test suite is, the fewer mutants will be completely unreachable. We see no impact in the JSON project as it has no unreachable mutants. We see the most impact in the CppCheck project. Here, the test execution time of the unreachable mutants for the *unoptimised and mutant schemata* approach is half of the reachable ones. It is possible that running the project with a different configuration leads to a different number of unreachable mutants.

Our second observation is that the execution time of the reachable schemata approach is drastically shorter than that of the schemata and unoptimised approach, even when compared to only the reachable mutants. Instead of executing the entire test suite for each mutant, the reachable schemata approach only executes those test cases that reach the specified mutant. A mutant that survives will have fewer test cases executed compared to the mutant schemata and unoptimised approach. It also stands to reason that mutants that are detected are detected faster as the test cases that do not reach the mutant are not executed. We estimated its duration at $t^{schemata}_{test\_suite\_execution} * reachable\_mutants * decoupling\_factor$ in Formula 3. In Table 4, we can see that the decoupling factor varies from project to project. This factor can roughly be represented by the average amount of tests that will be executed per mutant, which we listed in Table 3. TinyXML2 has only 1 test, so there is no additional speedup to be gained from the reachable schemata approach. The JSON project has the most speedup from this approach as it only needs to execute 9.61% of the test cases per mutant, this is followed by the CppCheck project at 10.76% and the Google test project at 19.53%.

Our last observation is that we were unable to collect measurements regarding the test suite execution of split-stream mutation testing on the projects under analysis. Essentially because all projects relied on external components with special state. See Section 5.2.2 for an in-depth explanation. We, therefore, labeled its result as DNF (did not function).

Technique execution overhead. In block four, we can see that the execution time for both the unreachable and the reachable mutants is increased by using the schemata technique compared to the unoptimised technique. This is due to the additional run-time instructions that ensure the correct activation of the mutants. We listed the execution overhead introduced by the optimisation techniques in the execute mutant phase in percentages in the fifth block of Table 4. We can see that the percentual overhead between the unreachable and reachable mutants is, as expected, very close. As there are no unreachable mutants for the JSON project, no overhead can be calculated for it. We can also see that there is a limited overhead for the reachable mutants in the JSON and Google test projects of 0.16 and 0.15%. The TinyXML2 project is impacted more by 17.44%. This increase in overhead can be attributed to the increase in the number of mutants per LOPC. Where this was only 0.047 and 0.084 for JSON and Google test, it is 0.241 for TinyXML2. The more mutants there are per LOPC, the more if statements there are to verify which mutant needs to be executed. Mathematically dense programs will suffer more from the schemata implementation using the ternary operator. Switching to a switch-case implementation would reduce the impact. The CppCheck case has, with 0.565 mutants per LOPC, the highest number of mutants per LOPC. We thus also expected a higher impact on the execution time of the project. We measured an overhead of 120.29%. This overhead can be attributed to the many if statements that are introduced per statement. Changing the activation of the mutants from if statements to switch-case-based should reduce this overhead.

Timed out mutants. For many of the projects, the test time caused by timed-out mutants is fairly limited. This can be seen in the last block of Table 4. However, for the bigger, and longer running projects, this does become a significant amount of the mutation analysis time. This is especially true for the CppCheck project as the timed-out mutants take up 93% of the reachable schemata mutation analysis. We manually set a fixed timeout time for each of the projects. The time out is identical for each of the optimisation approaches except for the CppCheck project. The introduced overhead caused an additional delay for the schemata-based approaches. Here, we double the time-out time. The time-out time can be reduced specifically for the reachable mutant schemata strategy. We execute

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 5 Results: Speedups.

Mutants	TinyXML2	JSON	Google test	CppCheck	Saab case
Unoptimised:					
Considered	10 min 12.42 s	1 day 18 h 33 min 00.05 s	2 day 09 h 14 min 27.64 s	11 days 06 h 50 min 50.51 s	Confidential
Valid	9 min 17.58 s	1 day 15 h 27 min 36.31 s	2 days 05 h 33 min 02.04 s	10 days 02 h 46 min 25.90 s	
(vs. considered)	1.10×	1.08×	1.07×	1.12×	
Reachable	8 m 52.25 s	1 day 15 h 45 min 50.43 s	2 days 02 h 44 min 36.93 s	8 days 13 h 15 min 13.74 s	
(vs. valid)	1.05×	0.99×	1.06×	1.18×	
Schemata:					
Valid	1 min 50.77 s	17 h 47 min 23.89 s	10 h 59 min 15.12 s	12 days 10 h 59 min 35.81 s	Confidential
(vs. unopt. valid)	5.03×	2.22×	4.87×	0.81×	
Reachable	1 min 47.43 s	18 h 05 min 38.01 s	10 h 30 min 21.67 s	9 days 23 h 59 min 35.81 s	
(vs. unopt. reachable)	4.95×	2.20×	4.83×	0.86×	5.16×
(vs. schemata valid)	1.03×	0.98×	1.0×	1.25×	Confidential
Reachable chemata:					
Reachable/test	1 min 43.92 s	1 h 23 min 38.45 s	2 h 26 min 27.38 s	5 days 10 h 46 min 06.16 s	Confidential
(vs. unopt. considered)	5.89×	30.52×	23.45×	2.07×	
(vs. unopt. reachable)	5.12×	28.52×	20.79×	1.57×	
(vs. schemata reachable)	1.03×	12.98×	4.30×	1.83×	
(vs. valid schemata)	1.07×	12.76×	4.50×	2.29×	

each test case individually instead of the entire test suite: We should thus set an individual time out for each of the test cases. These time-outs should thus be much shorter, resulting in a reduced detection time when a mutant times out due to, for example, an infinite loop.

5.2 | Complete mutation analysis

In this section, we look at the impact of the optimisation strategies in relation to the complete mutation analysis while answering the research questions. Their timings and speedups can be found in Table 5.

5.2.1 | RQ1: How much speedup can we gain from mutant schemata?

The *schemata approach*, which virtually eliminates the compilation overhead, drastically speeds up the mutation analysis in most cases. As we have seen in Section 5.1, mutant schemata can reduce the compilation time down to almost the original, single, compilation time of the project. Here, we see speedups of up to 5.03× for the valid mutants, and up to 4.95× for the reachable mutants compared to the unoptimised approach, as listed in Table 5. JSON has a speedup of 2.22× due to a high proportion of test suite execution time versus its compilation time. The CppCheck project obtained a speedup 0.81× due to the increase in mutant execution time, attributed to the techniques implementation in combination with the high number of mutants per line of production code. In the previous section, we saw that there was a limited impact on the execution times of the mutants in projects with a low number of mutants per line of production code (less than 0.1: JSON and Google test). Projects with a high number of mutants per line of production code (0.24 and 0.56: TinyXML2 and CppCheck) suffered from increased execution overheads which limits its speedup potential. One may reduce such excessive overhead by adopting *switch* statements instead of *if* statements for mutant selection. We discuss this as one of the lessons learned (see '6. Ternary Operator and the Occasional Excessive Overhead'; p. 61).

We measured a speedup of 5.16× for the industrial Saab project for the mutant schemata approach compared to a traditional approach when excluding the completely unreachable mutants. For the other projects, we see that the speedups for this optimisation are slightly smaller than with the unreachable mutants. This is as expected as more time (compilation and execution time) is saved excluding the unreachable mutants in an unoptimised approach then in a schemata approach (execution time).

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

With the TinyXML2, Google test and the industrial Saab project, the mutant schemata technique obtains a speedup between 4.87× and 5.16×. For the other cases, the speedup was less eminent. JSON has a speedup of 2.22× due to a high proportion of test suite execution time versus its compilation time. CppCheck obtained as speedup of 0.81× due to due to the increase in mutant execution time, attributed to the techniques implementation in combination with a high number of mutants per line of production code.

The obtained speedup from a mutant schemata technique depends on two factors: (a) the proportion of the project compilation time versus its execution time; (b) the number of mutants per line of production code. The latter is implementation specific and its impact may be reduced by changing the activation of the mutants from if statements to switch-cases.

5.2.2 | RQ2: How much speedup can we gain from split-stream mutation testing?

For *split-stream mutation testing*, we expected an additional speedup of approximately 2× over mutant schemata. Unfortunately, this is only feasible for test suites consisting of true unit tests without any external dependencies. This property does not hold for the projects under investigation.

- TinyXML2 uses `iostream` to read from an input stream. At each mutation point we must copy the read pointer of the file and restore it to the position where it was. To complicate matters even further, the stream is opened and closed deep inside the library; thus, one needs intimate knowledge of the internal design.
- JSON suffered from the same issue as Tinyxml: it imports `iostream` to read from an input stream hence restoring the read pointer causes problems.
- Google test utilises its own forks to optimise its tests execution, which interferes with the forks we introduce. On top of that Google test does not report test failures via return values but uses the internal facilities for test reporting. Here, as well, one needs intimate knowledge of the internal design to ensure that the internal state is consistent when restoring.
- CppCheck is designed for a unix style pipes and filter architecture, and imports `iostream` to read from an input stream and write to the output and errorstream. Just as with TinyXML and JSON the mutation score for split stream mutation testing differed, hence we assume also for similar corrupted file pointers.

To deal with such external dependencies, we built in support to reset local files to a specific state using a local `git` repository. Also, we provided specific hook functions to reset more advanced dependencies (e.g., accessing a database). However, instantiating these hook functions requires domain specific knowledge of the program under test. Indeed, for split-stream mutation testing to work, one must for each injected mutant save the current state of the file-IO and revert it back later. This implies that one not only must interfere with the standard file IO, but sometimes must do so via other external libraries or intervene with the exception handling. Thus, it requires intimate knowledge of the system under test, its test architecture and the libraries it depends upon. During our analysis, the mutation score for the split stream mutation version always differed from the mutation score of the other optimisations. Somehow the internal state of the dependent component gets corrupted when resetting to a previous version. We lacked the domain specific knowledge about the system under test to identify and remedy the root cause for these corrupt states. As a result, the test fails when it shouldn't.

We conclude that split-stream mutation testing only pays of for test suites consisting solely on true unit tests without any external dependencies. This property does not hold for the projects under investigation. In theory, this strategy could yield a speedup of a factor 2, but we were unable to confirm this on our dataset.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5.2.3 | RQ3: How much speedup can we gain from eliminating invalid mutants?

In Table 5, we see that for the *unoptimised approach* a speedup between 1.07× to 1.12× is achieved by excluding the invalid mutants. This speedup is attained by a reduction in the compilation time as invalid mutants are mutants that cause compilation issues and can thus not be executed.

Compiler-integrated techniques like mutant schemata and split-stream for the C language family come with an important drawback: the tight integration with the compiler. Since all mutants are injected simultaneously, the resulting program must compile without any errors. Invalid mutants are not acceptable, since they prevent the compilation (and the execution) of the mutated program. This is especially challenging for statically typed languages with many interacting features and unforgiving compilers (C, C++, …). Since Clang has access to all of the project's type-information, we can programmatically ensure that any of the created mutants are statically correct. We can therefore not calculate a speedup for the invalid mutants as their exclusion is a requirement for the schemata techniques.

> Compared to an *unoptimised* approach the reduction in the compilation overhead leads to a speedup by a factor between 1.07× and 1.12×. This is not all that much, but excluding invalid mutants is a necessary prerequisite for the mutant schemata strategy discussed under RQ1.

5.2.4 | RQ4: How much speedup can we gain from eliminating unreachable mutants?

In Table 5, we see that for the *unoptimised approach* a speedup between 1.05× to 1.18× is achieved by excluding the completely unreachable mutants. This speedup stays approximately the same with the *mutant schemata* approach where the speedup is between 1.03× to 1.25× by excluding the completely unreachable mutants. (The JSON project is the exception that proves the rule—this project has no unreachable mutants.) This speedup, however, will depend on the coverage of the test suite. A test suite with low coverage and thus reaching fewer mutants will yield a higher speedup by excluding them.

However, the *reachable schemata* technique goes a step further by not only excluding completely unreachable mutants but by also excluding the test cases that do not reach the mutant on a mutant by mutant case. This gives an additional speedup between 1.83× and 12.98× over the schemata technique that excludes completely unreachable mutants. This provides a speedup between 2.29× and 12.76× over the normal schemata technique. (Here, the TinyXML2 project is the exception, as it only has a single test case.) Further speedups are possible by optimising the time-out function that, for example, detects mutants stuck in infinite loops. This can be achieved by specifying a time-out for each test case instead of a time-out for the global test suite. It would be interesting to investigate whether excluding tests that do not infect the program states (i.e., that do not even pass the *weak* mutation criterion) would further induce speedups.

> Compared to an *unoptimised* approach, excluding the completely unreachable mutants provides a speedup between 1.05× to 1.18×. If we go one step further (also excluding the test cases that do not reach the mutant on a mutant by mutant base) we achieve an additional speedup between 1.83× and 12.98×.

6 | CHALLENGES, LIMITATIONS AND LESSONS LEARNED

This paper presents a feasibility study, investigating to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Mutation testing and mutation optimisations for statically typed languages like the C family are challenging to implement as they have many interacting features and unforgiving compilers. We demonstrated via a proof-of-concept that the Clang front-end allows to alleviate some problems. Doing our feasibility study, we made ten interesting observations, which we share below as challenges, limitations and lessons learned.

1. *Tests with external dependencies hinders split-stream mutation testing*.
   In Section 5.2.2, we concluded that split-stream mutation testing only pays of for test suites consisting solely on true unit tests without any external dependencies. In the projects we used in our analysis, this property did not hold and such external dependencies are a common phenomenon in continuous integration settings. After the test suite has run, the external dependencies, like input/output files and databases, need to be reset for the next run of the Continuous Integration. The support for this is usually baked into the used build systems of the projects (e.g., `make clean`). Of the four optimisations we investigated, split-stream mutation testing suffers when the test demands that the external dependencies must be reset to a specific state. Unfortunately, the Clang compiler framework does not provide sufficient features for a precise analysis of where such external dependencies occur, hence cannot identify and revert state changes in dependent components and libraries. We provided built-in support to reset local files to a specific state using a local git repository. Yet, more advanced dependencies like running an (in-memory) database need specific commands and intimate knowledge of the program under test and its test architecture. The implementation for this will be different for each project. These problems are not insurmountable, they are similar in nature to the problems that occur with distributed mutation testing. They are however out of scope for what a Clang compiler front-end can support.
2. *Multi-threading prohibits split-stream mutation testing*.
   Traditional, unoptimised mutation testing and mutant schemata inherently support applications that utilise multiple threads as they run from start to finish. Split-stream mutation testing (as all other optimisation techniques that manipulate the test execution) suffers when dealing with multi-threaded programs, as was the case with Google test. If we 'start' the first mutant after a secondary thread is spawned, our mutant likely runs to the end of the program and wait for the secondary thread to finish. A second mutant 'started' from the same location expects the secondary thread to be there, but as it is already closed, the second mutant does not run correctly. Supporting multi-threaded programs for the split-stream approach require additional development to ensure that not only the main thread will be forked, but also the existing threads. Here, as well, this is out of scope for what the Clang compiler front-end can support.
3. *Equivalent mutants*.
   Equivalent mutants have been heavily studied in the literature as they may induce heavy overhead on test engineers aiming for 100% mutation coverage [27]. Eliminating equivalent mutants before executing the test suite is thus a promising optimisation, much like eliminating unreachable mutants. A paper by Offutt et al. illustrates how program analysis can help to identify equivalent mutants by demonstrating that they belong to an *infeasible path* [39]. The authors argue that a mutant is equivalent if the injected mutant lies on an *infeasible path*; thus (according to the RIPR model), the injected mutant can never propagate to the assert statements that reveals it. Whether the Clang AST level provides sufficient facilities to detect *infeasible paths* remains to be seen. Indeed, the infeasible path analysis will induce a much larger overhead than the other optimisation technique we considered here. However, even if the computational cost is high, it will greatly reduce the cost of subsequent manual analysis. Automatically detecting even a small fraction of equivalent mutants saves valuable developer time and that alone is reason enough to consider this in future work.
4. *Trivial compiler equivalence*.
   The most pragmatic approach for detecting equivalent mutants to date is called *trivial compiler equivalence* (TCE). The technique compares the generated (byte) code of the mutated program against the original [34]. Due to compiler optimisations the syntactic differences between the original and mutated program may disappear and then they are considered *trivially* equivalent. This allows to identify the easy cases, however, for the difficult ones, further analysis—like infeasible paths—is required.
   Implementing a scheme for trivial compiler equivalence on top of the Clang front-end in combination with the LLVM IR looks a very promising avenue for further investigation. For the moment, we did not yet implement it in our proof-of-concept since it goes well beyond the Clang front-end analysis we set out as as a goal. Nevertheless, based on our experience, we believe that we can integrate trivial compiler equivalence by relying on the different compilation phases as we did when generating and validating the mutants. We only need to execute the first steps of the compilation, including the generation of the abstract syntax tree, once. The abstract syntax tree then resides in memory, after verifying the mutant with the semantic analyser, we can then ask the compiler to generate the accompanying LLVM IR and optimise it. This approach then generates multiple binaries that we can utilise to detect trivial compiler equivalency, but with a reduced compilation cost. Combining this approach with our mutant schemata

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

approach however re-introduces some of the overhead we saved with the schemata approach as we need to generate the binaries for both the trivial compile equivalency detection and the single binary for the mutant schemata execution. Depending on the amount of equivalent mutants, it might also be better to perform this detection technique post mutation analysis by only incorporating the survived mutants. Additional research is needed to investigate the best approach to integrate trivial compiler equivalence as an optimisation strategy.

5. *Ensuring type safety*.
   A first challenge occur when we implement mutant schemata for a statically typed programming language. By design, the mutated program must be syntactically correct—no type errors should occur; thus, every mutated statement should be valid for the compiler. We cannot generate mutants like 'string − string' or 'float % int'. Classes can overload or omit operators like '+' and '−' further complicating the matter. Clang allows us to access all the statically available information of the project and to verify if a mutated statement is syntactically correct without the need to compile the complete project. In that sense, the Clang AST level provides all what is needed to implement mutant schemata and the extra analysis time is more than catered for with a reduced compilation time.
   While we have demonstrated that this works for binary expressions, we have not investigated how this needs to be done for less straightforward mutations such as access modifiers change (AMC), where the public access label is changed to private. With our experience, we believe other mutation operators can be incorporated in mutant schemata using Clang, but the analysis time for such a change will be longer compared to the analysis of a binary expressions.
6. *Ternary operator and the occasional excessive overhead.*
   In Section 3.2.2 (Listing 6), we used the ternary operator to select the active mutant. The ternary operator simplifies and streamlines the implementation of the schemata technique and it allows nesting the mutants inside the condition of an *if* statement. However, the ternary operator limits the mutations we can inject, as all mutants in the expression need to be of the same type. We cannot have a char pointer on the *iftrue* side and an unsigned int on the *iffalse* side.
   In most projects, the ternary operator induces a fixed and limited overhead cost for the technique per mutant. However, in some cases—where the number of mutants per line of production code is high (such as math-heavy projects)—the ternary implementation causes too many additional evaluations of the if statements to reach the correct mutant. The most common case where no mutant in the statement is activated is the worst-case scenario, as all the if statements need to be evaluated before the original statement can be executed. In the simplistic example in Listing 9, we can see that an '$a*b*c$' statement has six mutations. This means that for every mutant outside of that expression, six if statements need to be evaluated. By switching to a switch-case implementation like in Listing 9, only the switch needs to be evaluated and the offset for which case to jump to needs to be calculated. This would save many instructions especially when there are several instructions on a single line of code, as was the case for the CPPCheck project. Alternatively, we could add an additional check to verify if the active mutant is located on the line. If it is not, then we could immediately execute the original statement. This should alleviate a large part of the overhead.
   To reduce the overhead of the schemata strategy, we recommend using the switch-case implementation in larger projects, despite the implementation challenges that need to be overcome. We described some of these challenges and tradeoffs in Section 3.2.2, including code explosion and the scoped nature of the switch case causing the need to initialise variables outside the switch. Because of these challenges, additional research is needed to understand where and how switch cases should be used, and how to deal with its tradeoffs. It is even possible that if statements and switch-cases should be used together to achieve an optimal speedup.

Listing 9: Mutant Schemata using Switch Case

```
1  extern int MNR;      // driver for active mutant
2  float f(float a, float b, float c) {
3      switch (MNR) {
4          case 1:  return (a * b) + c;
5          case 2:  return (a * b) - c;
6          case 3:  return (a * b) / c;
7          case 4:  return a + (b * c);
8          case 5:  return a - (b * c);
9          case 6:  return a / (b * c);
10         default: return a * b * c;
11     }
12 }
```

7. *Const, constexpr and templates.*
   The driver for mutant schemata relies on information from outside the program to control the activation of the mutants by setting the MUTANT_NR variable. The MUTANT_NR variable is initialised at runtime and will thus

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

never be const. This implies that we cannot use the variable inside const and constexpr functions, as these functions are evaluated at compile time and the MUTANT_NR value cannot be known at compile time. An example of this can be seen in Listing 10. As a consequence, we cannot mutate const and constexpr in the same way as non-const functions. This includes type definitions (e.g., *using* …), template arguments, static_asserts and so on.

Listing 10: Invalid Schemata from const

```
1  const float a = 1.2;
2  const float b = 2.0;
3  // original statement
4  const float r = a * b;
5
6  // mutated statement
7  extern int MNR;      // driver for active mutant
8  const float r = (MNR == 1 ? a + b :
9                   (MNR == 2 ? a - b :
10                  (MNR == 3 ? a / b : 1.2 * 2)));
11                             ~~~~~~~Invalid schemata MNR cannot be know at compile time
```

For now, we choose not to implement support for mutations in constant expressions, type definitions, template arguments and static_asserts. We do however generate and verify these for a standalone mutation, but using them in a single compilation for mutation testing using mutant schemata and/or split stream requires additional logic. This can be implemented by creating separate values and/or functions for each mutated operation and selecting the correct one everywhere in the project where they are used (e.g., const val becomes const val_0, const val_1, const val_2, …). This will drastically increase the size of the mutated code and the compiled binary.

8. *Mutation operator support*.
   Currently, our proof-of-concept tool supports a few representative mutation operators: relational operator replacement (ROR), arithmetic operator replacement (AOR) and logical connector replacement (LCR). This is only a fraction of the commonly used mutation operators for C and C++ (listed in Table 1) and an even smaller fraction of the more than 100 mutation operators reported in the academic literature. Extending the analysis to other mutation operators remains an area for further research.
   Our proof-of-concept tool is designed to be extended for supporting other binary mutation operators and unary operators. However, other mutation operators will need more work because ensuring that these are valid mutations involves a deeper semantic analysis beyond the AST node currently mutated. Access modifier change, for instance, demands that a method or instance variable raised from private to public does not overwrite other public instances. Also, the way we exclude unreachable mutants (an initial test run counting all mutated AST nodes) will not hold anymore with mutation operators like statement deletion.
9. *Test per test case versus per module*.
   Our program extracts the reachable mutants per test case or per module in the program under test. The best results are obtained by using fine-grained test selection and running each test case individually instead of module-grained. All our projects are compatible with the fine-grained test selection. Testing per test instead of per module reduces the average number of tests per mutant. We verified this for the CPPCheck project. Here, the average tests per mutant for the fine-grained test selection is 403, while it is 812 for the module-based one. The additional 409 tests selected by the module-based approach do not cover the mutant and will never be able to kill the mutant. Given a random distribution of these tests, we can assume that twice the amount of test cases will be executed per mutant for the CPPCheck project. If all test cases take approximately the same amount of time, the reachable schemata technique per test is twice as fast as the per module. Naturally, the average tests per mutant for the fine-grained test selection or the module based one, and hence the speed difference, will depend on the project itself.
   The CPPCheck program implemented its own testrunner to enable running the tests per test instead of per module. If one use ctest, then traditionally, tests are added using the *add_test* command to a specific test module. Each test module can be run separately, but one cannot run a specific test within a test module without running the other tests from that module. By switching from the *add_test* command to *gtest_add_tests* and/or *gtest_discover_tests*, one can run each test individually. This means that each test will run slightly slower, as a new test environment will be created for each test instead of for each module. This will however be greatly offset by the reduction in the number of mutants that will be selected to run for each test.
10. *Timed out mutants overhead*.
    For the bigger projects like CppCheck, we have seen that the execution time of the optimised mutation analysis consists mostly of timed-out mutants. Here, the timed-out mutants take up 93% of the reachable schemata

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

mutation analysis. In our current approach, we detected the timeout once the total time for that mutant reached a threshold. For the CppCheck project, this means that a test that is stuck in an infinite loop would only be timed out after 45 s. As the CppCheck project has 3745 test cases, the average test time is below 0.01 s. The impact of the timed-out mutants will be drastically reduced if we stop mutants not after a global threshold but after localised thresholds based on the individual tests.

7 | THREATS TO VALIDITY

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with guidelines for empirical research (see previous studies [40, 41]), we organise them into four categories.

Construct validity: do we measure what was intended?

In essence, we want to know which parts of the mutation testing process are affected by building a mutant schemata and split-stream mutation optimisation on top of the Clang front-end, as we believed these strategies would eliminate the compilation and execution overhead of mutation testing. The mutant schemata strategy and extended reachable schemata strategy cause additional delays in some parts of the mutation testing process, but in return eliminate the need to compile every mutant individually. The real question is whether the time benefit from a single compilation outweighs the added delays. For this, we measured each part of an unoptimised mutation testing process and compared it to the optimisation strategies. Below are three threats to validity concerning the way we collected the timing measurements.

We implemented all of our mutants using the ternary (conditional) operator. This causes many additions to the code base, which impact the compilation time and test suite execution time. The execution time for the CheckCpp project was affected heavily as many ternary operators are written on frequently executed lines. In such cases, a different structure using switch-cases could have prevented such large delays (cf. Listing 9), perhaps even a hybrid approach.

While we chose a representative (but limited) set of mutant operators (i.e., ROR, AOR and LCR) and omitted mutants in const and templated expressions, we are confident that our results represent the performance benefits one can achieve using the optimisation strategies. Indeed, the compilation times for the currently supported mutants are only impacted slightly, adding less than 25% to the compilation time for our biggest project. The test suite execution times were not impacted significantly except for the CheckCpp case where it caused an additional delay of 120%, which could be improved by using *switch* statements instead of *if* statements to select the mutants.

To ensure that the results are reliable and comparable, we used the same generate mutants method for all optimisation techniques. Additionally, we verified that the mutants we label as invalid mutants using the Clang front-end are the same mutants that actually caused compilation failures in the unoptimised baseline. This ensures that the same mutants are used across the various steps of the optimisation techniques.

Internal validity: are there unknown factors that might affect the outcome of the analyses?

We expected that the more mutants there are per line of code, the more the test suite execution time would be impacted as there is simply more code to execute. However, most of our projects did not show such results, they showed no difference in execution time with or without mutants. This might be due to the architecture of modern CPUs where out-of-order execution and pipeline depths/stalls can influence the effective performance. We did not investigate this sufficiently to draw firm conclusions.

We did however, observe symptoms of this phenomenon in the case of CppCheck, for which the test execution times were heavily impacted. This was due to the number of mutants that can occur on a single line, drastically changing the execution time of that specific line. When that line is executed frequently, the performance of the entire system is affected considerably. On the other hand, the performance impact is minor if that single line is only executed once. Further investigate (using a profiler) will be needed to understand the root cause.

To minimise environment factors, we locked the frequency of the CPU to the base frequency of the CPU and kept the computer in a well-ventilated space to prevent CPU throttling. While we used an older computer (Intel (R) Core(TM)2 Quad Q9650 CPU) architecture to run our experiments, we don't think this does affects the analysis. The use of a hard drive instead of an SSD however will reduce the compilation times and likely lower the total speedups. Here, as well, the impact should be minimal, and that the benefits of the optimisation techniques would still be valid even when using an SSD.

We used the same infrastructure for the analysis of the five selected open-source projects. We used an Intel(R) Core(TM)2 Quad Q9650 CPU, with two 4GB (Samsung M378B5273DH0-CH9) DDR3 RAM modules (for a total of

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

8GB) and a 250GB Western Digital (WDC WD2500AAKX-7) hard drive. The PC was running Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-29-generic x86_64). Using an SSD would drastically influence the compilation times and negatively impact the total speedups. Using more RAM and/or a faster CPU might influence the compilation time and/or execution time, this, however, is presumed to be marginal.

External validity: to what extent is it possible to generalise the findings?

We evaluated our proof-of-concept tool on four open-source projects with different characteristics and one industrial component. These projects vary in size and in computational needs, varying from a low number of mutants per line of production code to high numbers. As such, our results from these different projects represent the performance benefits one can achieve by using the different optimisation strategies. But as always, replications with other projects are needed to confirm our findings. Our tool is also available for replication purposes on Code Ocean together with a description on how to use it.[6]

Reliability: is the result dependent on the tools?

We took great care to ensure that external tools did not impact our timing results. The internal toolchain builds on very established components from the LLVM and Clang projects. We only measured the time it took to compile the project, generate the mutants and execute them. We excluded any timings related to external tools or implementations like database access times to store the mutants. Our results thus represent the performance benefits of the optimisation strategies as implemented. On the other hand, we do see areas that can affect the performance. In some cases, where many mutants are listed on a single line, it might be faster to execute them using a *switch* statement than to use our *if* statement. In this case, the compiler can create a jump table (using consecutive indexes), that is, an array of pointers, to directly jump to the correct label. Adding more mutation operators might also impact the performance.

8 | CONCLUSION

In this paper, we investigate to which extent the Clang front-end and its state-of-the-art program analysis facilities allow to implement existing strategies for mutation optimisation within the C language family. We present a proof-of-concept tool that allows us to collect detailed measurements for each of the mutation phases, that is, generate mutants, compile mutants and execute mutants. We validate the proof-of-concept tool on four open-source C++ libraries and one industrial component, covering a wide diversity in size, C++ language features used, compilation times and test execution time.

As such, we analyse the speedup from two perspectives (compilation time and execution time) assessing four optimisation strategies (exclude invalid mutants, mutant schemata, reachable mutant schemata and split-stream mutation testing). We created a performance model to estimate the impact of these optimisation strategies: unoptimised (Formula 1; p. 12); mutant schemata (Formula 2; p. 14; Formula 3; p. 16) and split-stream (Formula 4; p. 19). The validation against the five cases allows us to assess the significant terms in the performance estimation formulas. The 'Generate Mutants' and 'Detect (Un)Reachable Mutants' are for all practical purposes negligible. The 'Compile Mutants' step however takes a significant amount of time and—quite importantly– the compilation of the invalid and unreachable mutants is considerable. The 'Execute Mutants' step is the other dominant factor, although here project specific features ((a) the average amount of tests that will be executed per mutant and (b) the number of timed out mutants) cause a lot of variation.

The rest of our analysis was driven by four research questions, addressed below.

RQ1: *How much speedup can we gain from mutant schemata?* We could virtually eliminate the compilation overhead to the point that the compilation for all mutants was only slightly longer compared to the original compilation time, that is, the time it takes to compile the project without any mutants. With the TinyXML2, Google test and the industrial Saab project, the mutant schemata technique obtains a speedup between 4.87× and 5.16×. Yet the obtained speedup depends on two factors: (a) the proportion of the project compilation time versus its execution time; (b) the number of mutants per line of production code. The latter is implementation specific and its impact can be reduced by changing the activation of the mutants from if statements to switch-cases. As a consequence of our implementation, for the other cases, the speedup was less eminent: JSON has a speedup of 2.22× due to a

[6] https://codeocean.com/capsule/3514968/tree/v1.

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

high proportion of test suite execution time versus its compilation time. CppCheck obtained as speedup of 0.81× due to the increase in mutant execution time, attributed to the techniques implementation in combination with a high number of mutants/LOPC.

RQ2: *How much speedup can we gain from split-stream mutation testing?* For *split-stream mutation testing*, we expected an additional speedup of approximately 2× over mutant schemata. Unfortunately, this is only feasible for test suites consisting of true unit tests without any external dependencies. The projects under investigation all had tests with external dependencies. We made an attempt to provide support to reset local files to a specific state using a local git repository and built in hook functions to reset more advanced dependencies like running, or even off-site, databases. Implementing these correctly requires in-depth knowledge of the system under test that we did not have. Thus, for the cases we investigated, we could not eliminate the execution overhead with the split-stream mutation testing strategy.

RQ3: *How much speedup can we gain from eliminating invalid mutants?* Compared to an *unoptimised* approach the reduction in the compilation overhead leads to a speedup by a factor between 1.07× and 1.12×. This is not all that much but it is a necessary prerequisite for the mutant schemata strategy. Since all mutants are injected simultaneously, the resulting program must compile without any errors. Invalid mutants are not acceptable, since they prevent the compilation (and the execution) of the mutated program. This is especially challenging for statically typed languages with many interacting features and unforgiving compilers (C, C++, …). Since Clang has access to all of the project's type-information, we can programmatically ensure that any of the created mutants are statically correct.

RQ4: *How much speedup can we gain from eliminating unreachable mutants?* Compared to an *unoptimised* approach, excluding the completely unreachable mutants provides a speedup between 1.05× to 1.18×. One notable exception is the JSON project that had no unreachable mutants. This speedup stays approximately the same with the *mutant schemata* approach where the speedup is between 1.03× to 1.25×. This speedup, however, depends on the actual coverage of the test suite: a test suite with low coverage (thus reaching fewer mutants) yields a higher speedup. The *reachable schemata* technique goes one step further by not only excluding completely unreachable mutants but also excluding the test cases that do not reach the mutant on a mutant by mutant base. This gives an additional speedup between 1.83× and 12.98× over the schemata technique that excludes completely unreachable mutants. Providing a speedup between 2.29× and 12.76× over the normal schemata technique. Further speedups are possible by optimising the time-out function that detects mutants stuck in infinite loops. This can be achieved by specifying a time-out for each test case instead of a time-out for the global test suite.

Overall. In summary, we successfully demonstrated the feasibility of using the Clang compiler front-end for different optimisation strategies. With the reachable schemata strategy, we virtually eliminated the compilation overhead to the point that the compilation for all mutants was only slightly longer compared to the original compilation time and we reduced the execution overhead by only executing the test cases for individual mutants that actually reach said mutants. Compared to an *unoptimised* approach we achieve a maximum speedup of 23.45× and 30.52× on the JSON and Google test projects with the *reachable schemata* strategy. Even for less ideal scenarios from the CPPCheck and TinyXML2 projects we achieve a speedup of 2.07× and 5.89×. These can be speedup further by two optimisations: Firstly, use *switch* statements for the mutant selection to reduce the technique overhead. Secondly, tailoring the time-out function, that detects mutants stuck in infinite loops, on a test by test basis instead of on the global test suite.

Finally, we report some lessons learned for deploying a mutant schemata tool using the Clang compiler framework. Most important is that we need a different, specialised approach for generating mutants in *const*, *constexpr*, *templates* and *define* macro's. These statements are evaluated at compile-time, thus obstructing the runtime selection of the mutant required by the mutant schemata technique. Secondly, the elimination of equivalent mutants (via a technique called trivial compiler equivalence) is a promising avenue for further research. This, however, extends the scope of the analysis: Besides the Clang AST level, one also needs code as generated at the LLVM IR level.

ACKNOWLEDGMENTS

This work is supported by (a) the Research Foundation Flanders (FWO) under Grant number 1SA1519N; (a) the FWO-Vlaanderen and F.R.S.-FNRS via the Excellence of Science project 30446992 SECO-ASSIST.

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interests.

DATA AVAILABILITY STATEMENT

The following supporting information is available as part of the online article:

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Availability of data	Availability statement
Data openly available in a public repository that does not issue DOIs	The data that support the findings of this study are openly available in Github:
Google test atF-ASTMut at	https://github.com/Sten-Vercammen/F-ASTMut
Data derived from public domain resources	The data that support the findings are derived from the following resources available in the public domain:
	TinyXML2 at https://github.com/leethomason/tinyxml2
	JSON at https://github.com/nlohmann/json
	Google test at https://github.com/google/googletest
	CppCheck at https://cppcheck.sourceforge.io
Data subject to third party restrictions	The data that support the findings of this study are from Saab AB. Restrictions apply to the availability of these data. The available data are inside this article with permission of Saab AB.

ORCID

Sten Vercammen https://orcid.org/0000-0002-9140-1488
Serge Demeyer https://orcid.org/0000-0002-4463-2945
Markus Borg https://orcid.org/0000-0001-7879-4371
Görel Hedin https://orcid.org/0000-0002-3003-2623

REFERENCES

1. Bass L, Weber I, Zhu L. DevOps: a software architect's perspective. Pearson Education, Limited: Sydney, 2015 (eng). OCLC: 1338835528.
2. M. R. Everything you need to know about tesla software updates. Teslarati, 2014. [on line], https://www.teslarati.com/everything-need-to-know-tesla-software-updates/ — last accessed In May 2023.
3. Jenkins J. Velocity culture, 2011. Keynote Address at the Velocity 2011 Conference.
4. Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering. 2011;37(5):649–78. https://doi.org/10.1109/TSE.2010.62
5. Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M. Mutation testing advances: an analysis and survey. In *Advances in computers* (vol. 112). Elsevier; 2019. p. 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015
6. Baker R, Habli I. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Transactions on Software Engineering. 2013;39(6):787–805. https://doi.org/10.1109/TSE.2012.56
7. Ramler R, Wetzlmaier T, Klammer C. An empirical study on the application of mutation testing for a safety-critical industrial software system. In *Proceedings of the Symposium on Applied Computing*. ACM: Marrakech Morocco; 2017. p. 1401–8. https://doi.org/10.1145/3019612.3019830
8. Petrovic G, Ivankovic M, Kurtz B, Ammann P, Just R. An industrial application of mutation testing: lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE: Vasteras; 2018. p. 47–53. https://doi.org/10.1109/ICSTW.2018.00027
9. Dupuydauby C. Mutation testing: first steps with stryker-mutator.net, 2019. https://gist.github.com/dupdob/60fefe8495c8ebe8638ffcffc98a8703 — last accessed May 2023.
10. Vercammen S, Demeyer S, Borg M, Eldh S. Speeding up mutation testing via the cloud: lessons learned for further optimisations. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM: Oulu Finland; 2018. p. 1–9. https://doi.org/10.1145/3239235.3240506
11. Usaola MP, Mateo PR. Mutation testing cost reduction techniques: a survey. IEEE Software. 2010;27(3):80–6. https://doi.org/10.1109/MS.2010.79
12. Offutt AJ, Pargas RP, Fichter SV, Khambekar PK. Mutation testing of software using a MIMD computer. In *In 1992 International Conference on Parallel Processing*. CRC Press: Boca Raton, Florida; 1992. p. II–257–266.
13. Ma W, Titcheu Chekam T, Papadakis M, Harman M. MuDelta: delta-oriented mutation testing at commit time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE: Madrid, ES; 2021. p. 897–909. https://doi.org/10.1109/ICSE43902.2021.00086
14. DeMillo RA, Krauser EW, Mathur AP. Compiler-integrated program mutation. In *[1991] Proceedings The Fifteenth Annual International Computer Software & Applications Conference*. IEEE Comput. Soc. Press: Tokyo, Japan; 1991. p. 351–6. https://doi.org/10.1109/CMPSAC.1991.170202
15. King KN, Offutt AJ. A fortran language system for mutation-based software testing. Software: Practice and Experience. 1991;21(7):685–718. https://doi.org/10.1002/spe.4380210704
16. Laplante PA, DeFranco JF. Software engineering of safety-critical systems: themes from practitioners. IEEE Transactions on Reliability. 2017;66(3):825–36. https://doi.org/10.1109/TR.2017.2731953
17. Clang: a C language family frontend for LLVM, 2023. LLVM Developer Group, https://clang.llvm.org — last accessed May 2023.
18. Myers GJ. The art of software testing. New York: John Wiley and Sons; 1979.
19. Cai X, Lyu MR. The effect of code coverage on fault detection under different testing profiles. ACM SIGSOFT Software Engineering Notes. 2005;30(4):1–7. https://doi.org/10.1145/1082983.1083288
20. Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*. ACM: Hyderabad India; 2014. p. 435–45. https://doi.org/10.1145/2568225.2568271
21. Gay G, Staats M, Whalen M, Heimdahl MPE. The risks of coverage-directed test case generation. IEEE Transactions on Software Engineering. 2015;41(8):803–19. https://doi.org/10.1109/TSE.2015.2421011

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

22. Kandl S, Chandrashekar S. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. Computing. 2015;97(3):261–79. https://doi.org/10.1007/s00607-014-0418-5
23. Parsai A, Demeyer S. Do null-type mutation operators help prevent null-type faults? In: Catania B, Královič R, Nawrocki J, Pighizzini G, editors. SOFSEM 2019: Theory and Practice of Computer Science. vol. 11376. Springer International Publishing: Cham; 2019. p. 419–34. https://doi.org/10.1007/978-3-030-10801-4_33
24. Parsai A, Demeyer S, De Busser S. C++11/14 mutation operators based on common fault patterns. In: Medina-Bulo I, Merayo MG, and Hierons R, editors. Testing software and systems. vol. 11146. Springer International Publishing: Cham; 2018. p. 102–18. https://doi.org/10.1007/978-3-319-99927-2_9
25. Li N, Offutt J. Test oracle strategies for model-based testing. IEEE Transactions on Software Engineering. 2017;43(4):372–95. https://doi.org/10.1109/TSE.2016.2597136
26. Just R. The major mutation framework: efficient and scalable mutation analysis for JAVA. In *Proceedings of the 2014 international symposium on software testing and analysis*, ISSTA 2014. Association for Computing Machinery: New York, NY, USA; 2014. p. 433–6.
27. Madeyski L, Orzeszyna W, Torkar R, Jozala M. Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Transactions on Software Engineering. 2014;40(1):23–42. https://doi.org/10.1109/TSE.2013.44
28. Offutt AJ, Untch RH. 2001. Mutation 2000: uniting the orthogonal. In Mutation testing for the new century, Wong WE (ed.), Springer US: Boston, MA; 34–44. https://doi.org/10.1007/978-1-4757-5939-6_7
29. Zhang L, Marinov D, Khurshid S. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM: Lugano Switzerland; 2013. p. 235–45. https://doi.org/10.1145/2483760.2483782
30. Denisov A, Pankevich S. Mull it over: mutation testing based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE: Vasteras; 2018. p. 25–31. https://doi.org/10.1109/ICSTW.2018.00024
31. Wang B, Xiong Y, Shi Y, Zhang L, Hao D. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM: Santa Barbara CA USA; 2017. p. 295–306. https://doi.org/10.1145/3092703.3092714
32. Kusano M, Wang C. CCmutator: a mutation generator for concurrency constructs in multithreaded C/C++ applications. In *2013 28th ieee/acm international conference on automated software engineering (ase)*. IEEE; 2013. p. 722–5. https://doi.org/10.1109/ASE.2013.6693142
33. Chekam TT, Papadakis M, Le Traon Y. Mart: a mutant generation tool for LLVM. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM: Tallinn Estonia; 2019. p. 1080–4. https://doi.org/10.1145/3338906.3341180
34. Papadakis M, Jia Y, Harman M, Le Traon Y. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE: Florence, Italy; 2015. p. 936–46. https://doi.org/10.1109/ICSE.2015.103
35. Delgado-Pérez P, Medina-Bulo I, Palomo-Lozano F, García-Domínguez A, Domínguez-Jiménez JJ. Assessment of class mutation operators for C + + with the MuCPP mutation system. Information and Software Technology. 2017;81:169–84. https://doi.org/10.1016/j.infsof.2016.07.002
36. Hariri F, Shi A. SRCIROR: a toolset for mutation testing of C source code and LLVM intermediate representation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM: Montpellier France; 2018. p. 860–3. https://doi.org/10.1145/3238147.3240482
37. Vercammen S, Demeyer S, Borg M. F-ASTMUT mutation optimisations techniques using the clang front-end. Software Impacts. 2023;16:100500. https://doi.org/10.1016/j.simpa.2023.100500
38. Untch RH, Offutt AJ, Harrold MJ. Mutation analysis using mutant schemata. ACM SIGSOFT Software Engineering Notes. 1993;18(3):139–48. https://doi.org/10.1145/174146.154265
39. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification and Reliability. 1997;7(3):165–92. https://doi.org/10.1002/(SICI)1099-1689(199709)7:3%3C165::AID-STVR143%3E3.0.CO;2-U
40. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering. 2009;14(2):131–64. https://doi.org/10.1007/s10664-008-9102-8
41. Yin RK. Case study research: design and methods (3rd edition), Applied social research methods series, vol. 5. Sage Publications: Thousand Oaks, Calif; 2003.

How to cite this article: Vercammen S, Demeyer S, Borg M, Pettersson N, Hedin G. Mutation testing optimisations using the Clang front-end. Softw Test Verif Reliab. 2023;e1865. https://doi.org/10.1002/stvr.1865

10991689, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/stvr.1865 by Universiteit Antwerpen, Wiley Online Library on [06/11/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License