

POLITECNICO DI TORINO

Master's degree in  
**Computer Engineering**

Master's degree thesis

# **Innovative Techniques for In-Field Testing of Safety-Critical Electronic Systems**



**Supervisors**

Professor Matteo Sonza Reorda  
Professor Riccardo Cantoro

**Candidate**

Mansour Sohrabian

Academic Year 2025–2026

## Abstract

Software Test Libraries (STLs) are used to test digital circuits during manufacturing and in-field diagnostics. An STL is a set of test routines that runs on the target system to stimulate the circuit and check its behavior. In many products, STLs run in short *idle windows*, for example between normal tasks, so they must finish quickly. They must also fit within tight memory limits, since the test code and its data share limited resources with the application. In practice, an STL must reach high fault coverage, but it also must run fast and fit in memory, so it can execute during short idle times without affecting normal operation.

Today, compaction is often done manually by deleting or rearranging test code and then running simulation and fault grading again. This loop is slow because simulation turnaround time can be long, so only a limited number of alternatives can be tried. It is also easy to break correctness or reduce fault coverage by mistake. Finally, when the design, toolchain, or fault model changes, the same manual steps are hard to repeat and results can be difficult to reproduce.

This thesis presents **TestCrush** [2], a Python framework that automates STL compaction. TestCrush applies a **greedy** removal strategy and keeps a change only if it reduces runtime (Test Application Time) while keeping fault coverage within the required limits. The framework integrates with an industrial simulation flow by orchestrating **Synopsys VCS** for logic simulation and **VC-Z01X** for fault simulation and grading. The result is a systematic and repeatable approach to shorten STLs with clear acceptance rules, while preserving (and in some cases improving) the achieved fault coverage. In our experiments on a specific case study, TestCrush reduced Test Application Time by up to 96% while slightly improving fault coverage by 0.35%. Furthermore, by incorporating enhancements such as preprocessing and an early-exit mechanism that skips unnecessary fault simulations, the framework significantly minimizes the required computational effort, reducing the overall CPU time for the compaction process by an average of 51% compared to previous baseline algorithms.

Future work includes developing faster compaction algorithms that reduce fault-simulation cost (e.g., by reusing fault information across iterations and mapping faults to code regions), and evaluating the approach on additional RISC-V processors and a broader set of STLs.

# Acknowledgments

I would like to express my sincere gratitude to Professor Matteo Sonza Reorda and Professor Riccardo Cantoro for their guidance, support, and valuable feedback throughout this work.

I am also grateful to my family and friends for their encouragement and patience during this journey.

# Contents

|          |                                                         |           |
|----------|---------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                     | <b>1</b>  |
| <b>2</b> | <b>Background</b>                                       | <b>5</b>  |
| 2.1      | STL usage and generation methods . . . . .              | 5         |
| 2.1.1    | How an STL is executed in practice . . . . .            | 6         |
| 2.1.2    | What an STL typically contains . . . . .                | 7         |
| 2.1.3    | How STLs are created . . . . .                          | 8         |
| 2.1.4    | What is compaction? . . . . .                           | 9         |
| 2.2      | Previous works on compaction . . . . .                  | 10        |
| 2.2.1    | Coverage-driven pool selection (coarse granularity) . . | 11        |
| 2.2.2    | Safe-fault-aware test reduction . . . . .               | 12        |
| 2.2.3    | Greedy removal of instructions and instruction blocks   | 12        |
| <b>3</b> | <b>New Techniques</b>                                   | <b>14</b> |
| 3.1      | Preprocessing . . . . .                                 | 15        |
| 3.1.1    | Motivation . . . . .                                    | 15        |
| 3.1.2    | Algorithm . . . . .                                     | 16        |
| 3.2      | A1xx: Adjacent block compaction . . . . .               | 18        |
| 3.2.1    | Motivation . . . . .                                    | 18        |
| 3.2.2    | Block structure and selection order . . . . .           | 19        |
| 3.2.3    | Acceptance policies . . . . .                           | 20        |
| 3.2.4    | Block size . . . . .                                    | 23        |
| 3.3      | A2xx: Random non-adjacent block compaction . . . . .    | 24        |
| 3.3.1    | Motivation . . . . .                                    | 24        |
| 3.3.2    | Block construction . . . . .                            | 25        |
| 3.4      | Early-exit optimisation . . . . .                       | 27        |
| 3.5      | Fault filtering . . . . .                               | 28        |
| 3.5.1    | Motivation . . . . .                                    | 28        |
| 3.5.2    | Fault-instruction reverse mapping . . . . .             | 29        |
| 3.5.3    | Per-trial fault set reduction . . . . .                 | 30        |
| 3.5.4    | Coverage adjustment and baseline refresh . . . . .      | 31        |

|          |                                                           |           |
|----------|-----------------------------------------------------------|-----------|
| <b>4</b> | <b>Experimental Setup</b>                                 | <b>34</b> |
| 4.1      | Target Design and Fault Model . . . . .                   | 34        |
| 4.1.1    | The CV32E40P processor . . . . .                          | 34        |
| 4.1.2    | Gate-level simulation . . . . .                           | 35        |
| 4.1.3    | Modules under test . . . . .                              | 36        |
| 4.1.4    | Stuck-at fault model . . . . .                            | 36        |
| 4.1.5    | Coverage metric: Observational Coverage . . . . .         | 37        |
| 4.2      | Simulation Environment . . . . .                          | 37        |
| 4.2.1    | Execution environment . . . . .                           | 37        |
| 4.2.2    | Overview of the validation pipeline . . . . .             | 38        |
| 4.2.3    | Cross-compilation . . . . .                               | 39        |
| 4.2.4    | Logic simulation with VCS . . . . .                       | 39        |
| 4.2.5    | Fault simulation with VC-Z01X . . . . .                   | 40        |
| 4.3      | TestCrush Framework . . . . .                             | 43        |
| 4.3.1    | End-to-end simulation-based compaction workflow . . . . . | 43        |
| 4.3.2    | Instruction-level compaction ( <b>A0</b> ) . . . . .      | 44        |
| 4.3.3    | Adjacent block compaction ( <b>A1xx</b> ) . . . . .       | 46        |
| 4.4      | TestCrush Configuration . . . . .                         | 46        |
| 4.4.1    | TOML configuration file . . . . .                         | 46        |
| 4.4.2    | ISA specification . . . . .                               | 47        |
| 4.4.3    | Algorithm parameters . . . . .                            | 47        |
| 4.4.4    | Preprocessing configuration . . . . .                     | 49        |
| 4.4.5    | Coverage formula and fault report . . . . .               | 49        |
| 4.4.6    | Traceability: logging and backup . . . . .                | 50        |
| 4.5      | Benchmark STLs . . . . .                                  | 50        |
| 4.5.1    | MULT STLs . . . . .                                       | 51        |
| 4.5.2    | LSU STLs . . . . .                                        | 52        |
| 4.5.3    | Diversity across the benchmark set . . . . .              | 52        |
| 4.6      | Evaluation Methodology . . . . .                          | 53        |
| 4.6.1    | Evaluation metrics . . . . .                              | 53        |
| 4.6.2    | Experimental protocol . . . . .                           | 54        |
| 4.6.3    | Algorithm comparison strategy . . . . .                   | 55        |
| 4.7      | Computational Cost . . . . .                              | 56        |
| 4.7.1    | Computational cost metrics . . . . .                      | 56        |
| 4.7.2    | Cost-reduction mechanisms . . . . .                       | 57        |
| 4.7.3    | Reporting format . . . . .                                | 58        |
| <b>5</b> | <b>Experimental Results</b>                               | <b>61</b> |
| 5.1      | Impact of Preprocessing . . . . .                         | 61        |
| 5.1.1    | Activation-Window Size Selection . . . . .                | 61        |
| 5.1.2    | Preprocessing Effectiveness across All STLs . . . . .     | 62        |
| 5.2      | Impact of the Range Policy . . . . .                      | 64        |
| 5.3      | A2xx Algorithm . . . . .                                  | 66        |
| 5.3.1    | Block Size Comparison within A2xx . . . . .               | 66        |

|          |                                     |           |
|----------|-------------------------------------|-----------|
| 5.3.2    | A2xx vs A1xx . . . . .              | 67        |
| 5.4      | Impact of Fault Filtering . . . . . | 68        |
| <b>6</b> | <b>Conclusions</b>                  | <b>72</b> |
| 6.1      | Summary . . . . .                   | 72        |
| 6.2      | Limitations . . . . .               | 74        |
| 6.3      | Future Work . . . . .               | 75        |
|          | <b>References</b>                   | <b>77</b> |

# Chapter 1

## Introduction

Safety-critical electronic systems must maintain dependable operation throughout their lifetime and therefore require effective in-field fault detection for permanent faults that may arise from aging and latent defects. Software Test Libraries (STLs) are a widely adopted approach because they can execute self-test routines on the target processor and expose fault effects through software-visible observations. However, STLs face a strict practical constraint: they must typically run during short idle windows, so the test application time (TaT) and code size must remain limited, often to the order of milliseconds.

In real projects, an STL is usually built by combining multiple test fragments targeting different modules and generated using different methods (manual tests, random tests, ATPG-assisted patterns converted to instructions, or formal techniques). When these fragments are merged, redundancy is common, meaning that many faults may be detected multiple times by different parts. This redundancy increases TaT without necessarily improving fault coverage, creating a clear need for STL compaction: reducing execution time (and ideally code size) while preserving the required fault coverage (FC).

The key difficulty is that deciding whether a fragment is redundant cannot be done reliably by inspection; it must be validated using the project's verification flow, typically requiring repeated logic simulation and functional fault simulation (e.g., with VC-Z01X) to measure TaT and FC after each candidate modification. These simulations are computationally expensive, so compaction methods that require a large number of experiments can become impractical, especially for large STLs.

Therefore, the problem addressed in this thesis is: how to compact an existing assembly STL in a way that reduces TaT while keeping FC within acceptable bounds, using a workflow that is computationally feasible and

integrates with an industrial VC-Z01X-based simulation environment.

This thesis is motivated by the need to make STL compaction practical, controllable, and effective under the constraints above. First, because fault simulation is typically the dominant cost, compaction must avoid unnecessary evaluations. For this reason, we apply an early-exit optimization: if a candidate modification already increases TaT in logic simulation, fault simulation is skipped and the change is immediately rejected, reducing the number of expensive runs.

Second, a compaction flow should support explicit trade-offs rather than relying on a single rigid acceptance rule. In some scenarios, allowing a small, bounded decrease in coverage can yield a significant TaT reduction; this motivates policies such as Range, alongside strict-improvement policies such as Maximize, to match different engineering constraints.

Third, the search strategy should not be limited to adjacent instruction blocks only. Block-based compaction that considers only contiguous segments can overlook beneficial removal combinations. This motivates exploring alternative candidate grouping strategies, including the A2xx random-block approach investigated in this work. Together with trace-based preprocessing to reduce the candidate search space, these motivations aim to make STL compaction both more efficient and more effective in realistic VC-Z01X-based flows.

Several works have addressed the problem of compaction of functional test programs and software test libraries. For example, Touati et al. [18] proposed a coverage-driven, pool-based strategy that selects a reduced subset of test programs to maximize coverage while minimizing redundancy; however, because it operates mainly at a coarse granularity (whole programs or large fragments), it provides limited visibility into instruction-level contributions and microarchitectural fault activation/propagation, and it may still require substantial fault-simulation effort to evaluate many candidates. Related work has also explored optimization opportunities tied to *safe faults*: Narang et al. [8] report that introducing signal-combination constraints to identify safe faults in an OpenRISC design can improve the achieved fault coverage, and they further show that removing a code segment that mainly contributes to covering safe faults can reduce the test program size on top of existing compaction techniques. Also the TestCrush framework<sup>1</sup> [2] provides a simulation-driven baseline for assembly STL compaction, including greedy algorithms such as A0 (single-instruction removal) and A1xx (adjacent fixed-size block removal) under a Maximize acceptance policy; the

---

<sup>1</sup><https://github.com/cad-polito-it/testcrush>

baseline algorithms are introduced conceptually in Chapter 2 and described in full in Chapter 4. Nevertheless, the baseline TestCrush flow is limited by strict acceptance criteria with limited support for controlled TaT-coverage trade-offs, candidate grouping strategies largely restricted to adjacent blocks, and runtime characterization based primarily on wall-clock time. Building on this baseline, this thesis introduces a Range policy, an early-exit mechanism to skip unnecessary fault simulation when TaT worsens, a new A2xx random-block compaction mode, a fault filtering mechanism to reduce per-trial fault simulation cost, and CPU-time measurement to strengthen both the effectiveness and the practical efficiency of the compaction process. These additions extend TestCrush toward a more practical and flexible compaction tool: Range supports explicit engineering trade-offs, early-exit and fault filtering reduce simulation cost, A2xx broadens the search strategy, and CPU-time measurement improves performance characterization. All contributions are developed and evaluated on assembly-level STLs targeting the CV32E40P RISC-V RV32IMC processor under the stuck-at fault (SAF) model, using Synopsys VC-Z01X for fault simulation.

The effectiveness of the proposed extensions is assessed experimentally by comparing the baseline TestCrush modes (A0 and A1xx with Maximize) against the contributions introduced in this thesis, namely the Range policy, the early-exit optimization that skips fault simulation when TaT worsens, the fault filtering mechanism that reduces the per-trial fault set, the A2xx random-block algorithm, and CPU-time measurement. The evaluation considers both outcome quality and computational effort. Outcome quality is primarily measured through the test application time (TaT) obtained from logic simulation and the fault coverage (FC) computed from the VC-Z01X fault report, while computational effort is characterized by the measured runtime (CPU time).

The analysis focuses on the trade-offs between reducing TaT and preserving FC under different acceptance policies. In particular, Range enables controlled compromises by allowing a bounded coverage decrease in exchange for TaT improvements, whereas Maximize favors strictly non-degrading coverage behavior. The results also highlight how the choice of candidate grouping strategy affects the search process: compared to adjacent block removal (A1xx), A2xx explores non-adjacent instruction groupings and can lead to different compaction trajectories. Finally, the early-exit optimization reduces the overall experimental cost by avoiding fault simulation in cases where logic simulation already indicates that a candidate is not beneficial. Detailed quantitative results, comparisons, and discussion are provided in Chapter 5.

The remainder of this thesis is organized as follows. Chapter 2 (Background) introduces Software Test Libraries (STLs), explains how they are

generated in practice, and reviews related compaction approaches from the literature. It also introduces the baseline compaction algorithms A0 and A1xx conceptually, providing the context needed to understand the extensions proposed in Chapter 3.

Chapter 3 (New Techniques) presents the contributions of this work. It describes trace-based preprocessing to reduce the candidate search space, the A1xx block-level compaction strategy with its acceptance policies (Maximize and Range), the A2xx random non-adjacent block algorithm, the early-exit optimisation that skips fault simulation when TaT already worsens, and the fault filtering mechanism that reduces per-trial fault simulation cost by excluding faults unaffected by the current candidate block.

Chapter 4 (Experimental Setup) describes the evaluation environment and methodology. It covers the target processor and fault model, the simulation infrastructure, the full TestCrush framework architecture and baseline algorithm implementations, the configuration parameters used in the experiments, the benchmark STLs, and the evaluation metrics.

Chapter 5 (Experimental Results) reports and analyzes the results in a stepwise manner. We first quantify the impact of preprocessing by comparing A1xx with and without preprocessing. After establishing preprocessing effectiveness, we compare A1xx + preprocessing under Maximize versus Range to evaluate the benefit of the Range policy. Finally, we compare A2xx + preprocessing + Range against A1xx + preprocessing + Range to assess the effect of replacing adjacent blocking with random blocking.

Chapter 6 (Conclusion) summarizes the main findings, discusses limitations, and outlines directions for future work. References are provided at the end.

## Chapter 2

# Background

### 2.1 STL usage and generation methods

An STL is a collection of software routines executed on the system processor to test the underlying hardware. STLs are the software product of the Software-Based Self-Test (SBST) method [17]: SBST is the general approach of using software programs running on the device itself to perform self-test, while an STL is the concrete artifact that results from applying this method to a specific target processor. The key idea is simple: instead of using external test equipment, the device runs a set of dedicated programs that stimulate the circuit under test (CUT) and check whether the observed behavior matches the expected one.

A major advantage of this approach is that it allows testing the system at-speed, since the test programs operate at the same frequency as normal application programs, without requiring costly, high-speed external testers. Furthermore, it implicitly tests both the individual modules composing a system (such as the processor, bus, memories, caches, and peripherals) as well as their interconnections. If a fault affects the hardware, its effect must propagate to a software-visible observation allowing the test to detect it. This software-visible observation can be achieved in several ways, such as monitoring output signals via a low-frequency tester, utilizing a programmable Multiple Input Signature Register (MISR) on the processor bus, or by writing the STL to simply store mismatching results (like a wrong register value or incorrect memory word) directly in memory for later verification.

STLs are widely used in safety-related products—such as automotive applications governed by the ISO26262 standard—because many permanent faults can appear after deployment due to aging, wear-out, or latent manufacturing defects. In such contexts, the purpose of an STL is not only manufacturing screening, but also in-field self-test: periodically checking that the hardware

still behaves correctly throughout its operational lifetime. For these safety-critical applications, standards specify exact target fault coverages that must be achieved, making minimal invasiveness on the application environment highly desirable.

In practical deployments, STLs are typically executed during short idle windows to prevent interference with the main application. This operational environment imposes strict practical constraints: the TaT must be extremely short (often on the order of milliseconds), and both the code and data must fit within tight memory limits shared with the application. To remain non-invasive while testing the system, STLs are built from multiple specialized test fragments targeting specific hardware modules and must include mechanisms for context saving, memory isolation, and deterministic completion. Therefore, designing an STL involves a constant trade-off between achieving the required FC and respecting these strict time, memory, and power constraints.

### 2.1.1 How an STL is executed in practice

The previous paragraphs already motivated *why* in-field STLs favor short runs in *idle windows*: the main application keeps real-time guarantees, while the self-test must stay within a tight TaT and memory budget. This subsection sketches *how* such a run is typically structured in practice, from scheduling to result handling.

**When the STL runs.** In most real systems, the self-test does not run all the time. It starts only when some rule says so. Typical examples are: after power-on or reset, on a fixed timer while the product is running, when the system is mostly idle, or when the operating system runs a planned diagnostic task.

Each time the self-test starts, it gets a *fixed* time allowance. Designers must assume that allowance is limited. If the whole STL is too long for one run, the work can be split over several runs. In that case, the integration must still define clearly what counts as a complete test, and startup/shutdown of each run must behave in a predictable way so that fault coverage can be reported without ambiguity.

**What happens during one invocation (high level).** Figure 2.1 shows a typical order of steps for one self-test run. *Prepare* sets up a test-safe configuration (for example interrupt and cache handling, and saving application context when required). *Test* executes the STL *fragments* that apply

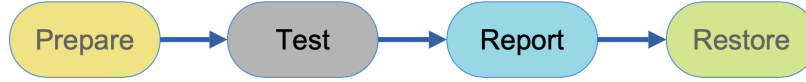


Figure 2.1: Typical phase order for one in-field self-test invocation.

stimulus and checks. *Report* writes consolidated outcomes—pass/fail flags, signatures, or small records—to agreed memory or registers. *Restore* returns the platform to normal operation and hands control back to the application. Details depend on the SoC and software architecture, but this four-step flow is a common template.

**Constraints that show up at execution time.** Even when the STL is logically correct, deployment constraints shape what “acceptable” means. The run must *finish* within the allocated window (bounding worst-case duration is often required for certification-style arguments). Memory for code and scratch data must remain within partitions agreed with the application. Watchdogs and safety monitors must remain in a valid state: long uninterrupted test loops or unexpected bus patterns may require coordination with system-level policies. Observability also matters: in-field tests may not rely on a laboratory debugger, so faults must become *software-visible* outcomes (for example sticky status bits or records that a supervisor can read later), consistent with the observation paths discussed above.

Because coverage, code size, and execution time pull in different directions, meeting these operational requirements while preserving the target FC is inherently a trade-off. Compaction, as discussed later, is one of the main levers for shortening TaT and footprint once a candidate STL exists.

### 2.1.2 What an STL typically contains

Although the exact structure depends on the processor and the fault model, industrial STLs are usually assembled from many *test fragments*. In practice, a fragment is a coherent piece of code with a focused goal (for example, stressing the integer pipeline or running a memory march test). The split is partly organizational—developers can reuse, replace, or tune fragments independently—and partly diagnostic: coverage is often reported and debugged per block or per routine. The following categories appear frequently:

- **Processor and pipeline exercises:** instruction sequences that target decoding and execution of representative operations, register-file

behavior, basic ALU/logic and shift paths, and selected pipeline or hazard situations.

- **Memories and address paths:** RAM/ROM-oriented algorithms (including march-style tests), checks that exercise address decoders and parts of the data path, and tests aimed at buses or alignment/endianness corner cases when they matter for the product.
- **Interconnect, shared resources, and peripherals:** transactions that stress on-chip buses, bridges, or controllers, plus targeted exercises for peripherals that are in the safety scope (for example a CAN controller used in an automotive ECU).
- **Caches, interrupts, and timing (when needed):** some platforms add extra routines for cache behavior, for how interrupts are handled, or for rare timing cases. Project documents do not always group these under their own heading; they still show up as separate small programs when one test needs to touch several parts of the chip at once (not only the core, not only memory, not only a peripheral).
- **Control, observation, and reporting:** three practical jobs. *Start:* put the system in the right mode for testing (and set clocks if the platform needs that). *Observe:* turn the raw test behavior into a small result—often a checksum-like signature or a pass/fail flag. *Report:* write that result to a known place (RAM or a hardware register) so other software can read it after the test finishes.

In real projects, these blocks are combined in different orders, but the end result is always the same idea: the STL is a long program made by chaining *many different* small tests one after another. Different fragments often exercise overlapping parts of the hardware, so the *same* fault can be detected more than once. Once fault coverage is already high enough, that overlap is a main reason why the library can often be shortened—the topic of compaction in later chapters.

In summary, an STL is both a test artifact (a program that must run correctly on the processor) and a diagnostic instrument (a sequence designed to expose hardware faults). Its practical value depends on achieving sufficient coverage while respecting tight time and resource budgets, which directly motivates the compaction techniques studied in the rest of this thesis.

### 2.1.3 How STLs are created

In industrial projects, an STL is rarely produced by a single method. Instead, it is assembled from fragments generated using a mix of approaches, for example:

- **Manual tests:** handcrafted routines based on expert knowledge of the processor and the platform, possibly resorting to specific test algorithms (e.g., march algorithms for memory, or structured walking-ones patterns for register files).
- **Random or constrained-random tests:** instruction sequences generated to explore diverse states and behaviors.
- **ATPG-assisted patterns converted to software:** hardware test patterns (or guided sequences) translated into instructions to target specific faults.
- **Formal or model-based techniques:** sequences derived from an abstract model to ensure specific properties or activation conditions.

To achieve the high FC required for safety-critical systems, industrial STLs are typically assembled from a heterogeneous mix of test fragments, including manual routines, ATPG-assisted patterns, and random or constrained-random instruction sequences. When traditional development methods reach a coverage plateau, using new seeds to generate additional random vectors is a common strategy to explore diverse processor states and detect remaining faults. However, this approach inherently introduces redundant test effects, as these new sequences often cover faults already detected by existing fragments. This redundancy leads to an inflated TaT and increased code size, which can prevent the STL from fitting within the short idle windows and strict memory limits required for in-field operation. Consequently, automated compaction becomes a critical step in the development flow; it allows for the removal of these overlapping test effects, ensuring the STL remains efficient and practical without sacrificing the hard-won fault coverage.

In other words, although heterogeneity is beneficial for reaching coverage targets, it also increases the likelihood that the merged STL will contain redundant test effects, meaning that the same faults are detected multiple times. As a result, a large STL may violate TaT constraints without providing proportional gains in fault coverage, thereby motivating the compaction problem addressed in this thesis. The benchmark STLs used for evaluation in this thesis are described in Chapter 4 (Section 4.5).

#### 2.1.4 What is compaction?

Following the terminology used in the literature on functional test programs [4], *compaction* means *reducing* the *size* and/or the *execution time* of a test while controlling its quality (here: fault coverage under a chosen fault model). The two objectives are related but not identical: shorter programs often run faster, but duration is not always proportional to program length.

**Dynamic versus static compaction.** Compaction techniques are often grouped into two families [4]. *Dynamic* methods act *during* test generation: they try to produce compact tests from the outset. *Static* methods act in a *separate phase after* generation, by editing an *existing* test. The two approaches are not mutually exclusive: for example, Neophytou and Michael [9] combine static and dynamic techniques in test-set generation when many bits are left unspecified. Separately, static compaction can still remove redundancy *after* dynamic generation or dynamic compaction has been applied [4].

**Why static compaction matters for STLs.** In practice, test sets are frequently *reused*: they come from validation flows, legacy bring-up, or heterogeneous fragment assembly, and only minor edits may be possible. When a test program already exists, purely dynamic approaches are not applicable. Static compaction then becomes the natural way to shorten an STL without restarting the whole generation flow [4].

**Compaction of programs versus raw bit sequences.** Classical sequence compaction operates on streams of stimuli. A *test program* must remain a valid program: it runs on a processor, uses memory, and may contain branches and loops. Therefore, compaction must work at the *assembly* (or *source*) level [4]. In particular: (i) programs interact with memories and the system environment, so the scenario is more structured than a bare stimulus stream; (ii) control flow means that TaT is not necessarily proportional to code size—removing an instruction can change paths and even *increase* duration; (iii) removing instructions can break termination (exceptions, hangs), so each candidate must be validated by execution-aware checks, not only by a coverage metric [4].

**Role in this thesis.** Section 2.2 reviews related compaction ideas from the literature; Section 4.3 then introduces the TestCrush framework [2] and the baseline compaction algorithms it provides.

## 2.2 Previous works on compaction

Many papers study *static compaction* of binary test sequences for synchronous sequential circuits. The setup is simple: you start from a fixed order of test vectors and try to make the sequence shorter while keeping the same fault coverage. Researchers showed early on that some vectors can be removed one by one without losing any detected fault. They also showed that test generation—even with dynamic compaction—can still leave unnecessary vectors in the sequence. So it remains useful to compact the sequence *after* generation [12, 5, 13, 6, 1, 7, 14, 15, 16].

The simplest idea is *vector omission*: delete the vector at clock cycle  $u$ . That choice is local, but it can change what happens from cycle  $u$  onward. In particular, a fault that was detected at cycle  $u$  or later in the original sequence might no longer be detected. For that reason, omission algorithms use fault simulation (or very similar steps) to check whether a removal is allowed. Some methods also try to remove longer contiguous blocks in one go, once they know that a single omission works, in order to save repeated checks [12, 5, 6, 1].

A different family of methods uses *restoration*. First, remove a large block of vectors. Then put vectors back, one by one or in small groups, until fault coverage matches the original sequence again. Handling one fault at a time during this repair phase can be cheaper than re-checking the whole sequence after every change. Related techniques combine overlapping restoration, pruning of whole segments, and repair steps guided by single faults [13, 1, 7]. Other work goes beyond plain omission or restoration: it changes bits inside vectors, changes the order of candidate subsequences, or randomizes which vectors are omitted first, in order to shorten the sequence further [14, 15, 16]. When the input is a *set* of functional subsequences rather than one long stream, compaction can also merge or rearrange those pieces—including two-dimensional organizations of the material [10, 11].

This thesis focuses on *functional test programs* and *software test libraries* for processors. Here the inputs are not scan vectors but instructions and branches, and the quantity we most often want to reduce is TaT, not the length of a bitstream. The problem is related to classical binary-sequence compaction, but it is not the same: Section 2.1.4 already noted that a shorter program does not always run faster, and for in-field test, wall-clock time is often what matters [4]. Earlier work on compacting test programs stressed reducing program size [3]; later work targets TaT directly while keeping fault coverage [4]. The next three subsections review typical approaches in that setting: pool-based selection, safe-fault-aware reduction, and greedy removal of instructions or small instruction blocks.

### 2.2.1 Coverage-driven pool selection (coarse granularity)

Touati et al. [18] study compaction of functional test programs using a *pool-based* idea. The method starts from a set of candidate tests (a pool). It then selects a smaller subset that still reaches the target fault coverage, while removing redundant tests.

This approach is useful when a project already has many alternative tests to choose from. However, the selection often happens at a coarse level (whole

programs or large fragments). For this reason, it is harder to see exactly which instructions help detect faults. In addition, checking many candidates can still require a lot of fault simulation.

### 2.2.2 Safe-fault-aware test reduction

Narang et al. [8] study microprocessor self-test optimization based on *safe faults*. They show that introducing signal-combination constraints to identify safe faults can improve achieved fault coverage in an OpenRISC case study, and they further demonstrate that removing a code segment that mainly contributes to covering safe faults can reduce the test program size on top of existing compaction techniques. While this line of work is not a direct replacement for STL compaction, it highlights an important practical observation: not all detected faults contribute equally to safety goals, and exploiting such structure can create additional opportunities to shorten tests without harming the coverage that matters most.

### 2.2.3 Greedy removal of instructions and instruction blocks

A complementary and widely used family of methods compacts a single functional test program (or an STL) by greedily removing instructions or small blocks of instructions, accepting a removal only if it improves runtime while keeping fault coverage non-decreasing (or within an explicit bound). Gaudesi et al. [4] introduce techniques to reduce the execution time of functional test programs using such greedy strategies. In the STL context, the open-source **TestCrush** framework [2] provides a simulation-driven baseline that implements greedy modes such as **A0** (single-instruction removal) and **A1xx** (adjacent fixed-size block removal), integrated with industrial logic and fault simulation flows.

Operationally, greedy removal maps to an iterative *propose–validate–accept/reject* loop: each candidate deletion is checked with compilation, logic simulation, and fault simulation before it is committed. This thesis builds on the TestCrush baseline and extends it with additional policies and search strategies, as discussed in Chapter 3. The full TestCrush architecture, validation workflow, and configuration are described in Chapter 4.

#### Baseline algorithms: **A0** and **A1xx**

The two baseline algorithms provided by TestCrush are **A0** and **A1xx**.

**A0** removes one assembly instruction at a time. Starting from a golden evaluation that establishes the baseline  $(TaT_0, FC_0)$ , the algorithm iterates over all candidate instructions, removes each one in turn, and validates the result through compilation, logic simulation, and fault simulation. A removal

is accepted under the Maximize policy — TaT must not increase and FC must not decrease — and rejected otherwise, restoring the instruction before moving on. A0 sets the single-instruction granularity baseline against which block-level methods are compared.

A1xx extends A0 by grouping  $k$  consecutive removable instructions into a fixed-size block and attempting to remove the entire block in one trial. If the full block fails the acceptance test, a fallback procedure restores instructions one at a time — under a configurable Forward, Backward, or Random policy — and re-evaluates the shrinking sub-block until either a sub-block passes or the entire block is restored. A1xx processes blocks from the bottom of the STL toward the top, exploiting the typical structure of fault-targeting STLs where initialisation and setup code at the top is more critical than cleanup code near the end.

## Chapter 3

# New Techniques

The compaction algorithms developed in this thesis improve upon the A0 baseline along four distinct but complementary dimensions: the size of the search space the algorithm operates on, the granularity of each removal attempt, the criterion used to decide whether a removal is acceptable, and the amount of simulation work each trial requires. No single modification addresses all of these at once; instead, each technique targets one dimension independently, and together they form a coherent compaction framework that is more efficient and more effective than the baseline.

The first dimension concerns the search space. A0 begins with the full set of instructions in the STL as removal candidates. Many of these instructions are known, from the initial fault simulation results, to lie within the activation window of at least one detected fault. Attempting to remove them is guaranteed to degrade coverage and result in a Restore verdict. Preprocessing, described in Section 3.1, eliminates these instructions from the candidate pool before the compaction loop begins, so that every trial the algorithm runs has at least a plausible chance of being accepted.

The second and third dimensions concern what gets tried on each trial and how the outcome is judged. Rather than removing one instruction at a time as A0 does, the block-level algorithms introduced in Sections 3.2 and 3.3 attempt to remove a block of  $k$  instructions as a unit. This offers two concrete advantages. First, consider a group of adjacent instructions that together form a self-contained fault test: one sets up the operands, one triggers the operation under test, one observes the result. Removing any single instruction from the group breaks the test and causes a coverage drop, so A0 rejects each individual removal and leaves the whole group in the STL. Removing the entire group in one trial, however, may be perfectly safe if the same fault is still covered by another sequence elsewhere in the STL — an opportunity A0 can never discover because it never tries the group as a unit.

Second, even for instructions that are individually removable, removing  $k$  of them as a single block requires only one compile-lsim-fsim cycle instead of  $k$  separate trials. Since fault simulation is the dominant cost per trial, this reduction in trial count translates directly into lower total CPU time. Experiments in Chapter 5 confirm that block-level compaction consistently achieves lower total CPU time than A0 for the same compaction outcome. Two block selection strategies are proposed — A1xx uses adjacent blocks drawn sequentially from the candidate list, while A2xx assembles blocks by sampling instructions randomly without adjacency constraints. Both strategies share the same family of acceptance policies, described in Section 3.2.3, which govern the TaT and OC constraints a removal must satisfy: the Maximize policy requires that OC does not decrease at all, while the Range policy permits a small controlled drop bounded by a configurable tolerance  $\Delta$ .

The fourth dimension concerns per-trial simulation cost. Even with a pruned search space and block-level candidates, each trial that reaches the fault simulation stage incurs a campaign that can last several minutes. Two mechanisms address this. The early-exit optimisation, described in Section 3.4, detects trials that logic simulation has already condemned — those where TaT worsens before fsim is even launched — and skips fault simulation entirely for them. Fault filtering, described in Section 3.5, goes further: rather than skipping fsim, it limits the fault set that VC-Z01X evaluates on each trial to only those faults that the current candidate block can plausibly affect, reconstructing the full-set OC analytically from a pre-computed baseline. Together, these two mechanisms reduce the simulation budget consumed per accepted removal without changing the compaction decisions the algorithm makes.

## 3.1 Preprocessing

### 3.1.1 Motivation

At the start of a compaction run, the candidate list is the full set of instructions in the STL. From the algorithm’s perspective, every instruction is a potential removal target until proven otherwise. In practice, however, a significant fraction of these instructions cannot be removed without degrading fault coverage. These are instructions that lie within the *fault activation window* of at least one detected fault: they execute immediately before or at the moment a fault effect is produced, and their presence is what enables the fault to propagate to a software-visible output. Removing such an instruction breaks the activation sequence, causing a previously detected fault to go undetected and producing a coverage drop that guarantees a Restore verdict.

The problem with leaving these instructions in the candidate pool is not

simply that they will be rejected — it is that every trial spent on them wastes simulation resources that could have been directed at genuinely removable candidates. Each such trial requires at minimum a cross-compilation and a logic simulation, and often a full fault simulation campaign as well, before the Restore decision is reached. In a compaction run that may execute hundreds of trials, the cumulative cost of these predictable failures is significant.

The information needed to identify protected instructions is already available before the compaction loop begins. The initial fault simulation, run once to establish the baseline TaT and OC, produces a report that records for each detected fault the simulation time and program counter at the moment of activation. By cross-referencing these activation points with the processor execution trace, it is possible to determine which instructions were executing at each activation event and in the cycles immediately preceding it. Any instruction that appears in one or more of these windows is a protected instruction: the compaction loop cannot safely remove it. Preprocessing performs this identification once, before any trial is run, and removes all protected instructions from the candidate list. The result is a reduced search space containing only instructions the algorithm has a realistic chance of removing, at zero additional simulation cost.

### 3.1.2 Algorithm

The preprocessing algorithm operates in three sequential steps on the golden fault simulation report and the processor execution trace, and produces a pruned candidate list as its output. Algorithm 1 gives the pseudocode; the steps are described below.

**Step 1 — Collect fault activation attributes.** The golden fault simulation report associates each detected fault with the simulation time and program counter value at the moment the fault effect was activated. Only detected faults contribute to this set: undetected faults never produce an activation event and therefore have no activation point to record. Duplicate  $(t, PC)$  pairs, which arise when multiple faults share the same activation moment, are collected once.

**Step 2 — Reconstruct activation windows from the trace.** A single activation point is not sufficient to identify all the instructions that contribute to fault detection. In a pipelined processor, multiple pipeline stages are in flight simultaneously, meaning that the instruction at the activation program counter is not the only one involved: instructions from the preceding cycles have already entered the pipeline and are simultaneously shaping the datapath state that allows the fault to propagate. To capture these contributors, the algorithm looks back *history* entries in the execution

---

**Algorithm 1** Preprocessing

---

**Require:**  $F_{\text{det}}$ : detected faults from the golden fsm report

**Require:** *Trace*: processor execution trace of the original STL

**Require:** *Candidates*: full instruction list of the STL (modified in place)

**Ensure:** *Candidates*: pruned list with protected instructions removed

```
1:  $A \leftarrow \emptyset$  ▷ set of unique activation attribute pairs  $(t, PC)$ 
2: for each fault  $f \in F_{\text{det}}$  do
3:   if  $f$  has activation attributes  $(t, PC)$  then
4:      $A \leftarrow A \cup \{(t, PC)\}$ 
5:   end if
6: end for
7:  $W \leftarrow \emptyset$  ▷ set of program counter values in activation windows
8: for each  $(t, PC) \in A$  do
9:    $window \leftarrow \text{QUERYTRACE}(Trace, t, PC, history)$ 
10:   $W \leftarrow W \cup window$ 
11: end for
12: for each  $pc \in W$  do
13:    $(file, lineno) \leftarrow \text{RESOLVETOSOURCE}(pc)$ 
14:   remove all entries with  $(file, lineno)$  from Candidates
15: end for
```

---

trace from each activation point, collecting the program counter values of the instructions that were executing in the cycles leading up to activation. The window therefore covers the activation instruction and a fixed number of its predecessors in the execution order. All program counter values across all windows are gathered into a single flat set  $W$ . Figure 3.1 illustrates this windowing concept.

**Step 3 — Map program counters to source lines and prune.** The compaction loop operates at the level of assembly source lines, not binary addresses. Each program counter value in  $W$  is therefore resolved to the corresponding source file and line number using the compiled ELF binary. Any candidate instruction whose source location matches a resolved entry is removed from the candidate list. This step modifies the candidate list in place, so that when the compaction loop begins it operates on the already-pruned set directly. Instructions that do not appear in any activation window are left in the list and remain eligible for removal.

The size of the search space reduction achieved by preprocessing depends on the STL and the target module. STLs with high fault coverage have many detected faults and correspondingly large activation window sets, leading to a larger fraction of instructions being pruned. The practical effect is measured and reported in Chapter 5 alongside the compaction results for each benchmark STL.

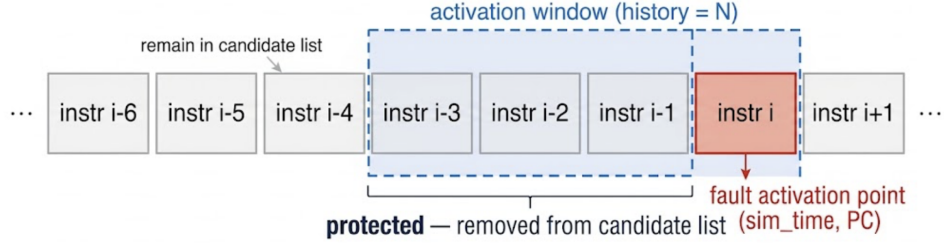


Figure 3.1: Activation window in the execution trace. The instruction at the fault activation point (`instr i`) and the  $N$  preceding instructions form the window; all instructions within the window are marked as protected and removed from the candidate list before the compaction loop begins.

## 3.2 A1xx: Adjacent block compaction

### 3.2.1 Motivation

As established in the opening of this chapter, removing instructions in blocks rather than one at a time addresses two limitations of A0: it can reach compaction opportunities that single-instruction removal cannot discover, and it reduces the number of fault simulation campaigns needed to achieve the same total removal count. The question of *how* to form and order those blocks is a design decision in itself. A1xx makes two specific choices: blocks are formed from *adjacent* instructions, and the blocks are processed from the *bottom to the top* of the STL.

The adjacency choice is motivated by systematic and exhaustive coverage. By sliding a fixed-size window across the candidate list, A1xx guarantees that every contiguous group of  $k$  instructions is attempted at least once during a full sweep. No contiguous group is ever skipped. This is important because an opportunity that requires removing a specific set of adjacent instructions together — because each one alone fails the acceptance test but the group as a whole passes — will always be discovered, regardless of where in the STL it appears.

The direction of traversal — bottom to top — is motivated by the typical structure of fault-targeting STLs. The beginning of an STL almost always contains initialisation and setup instructions: register file initialisation, stack pointer configuration, mode-switch sequences. These instructions are structural prerequisites; every fault test that follows depends on the processor state they establish. Attempting to remove them early in the run

is likely to break that state, producing a coverage collapse and a Restore verdict on the very first trials. The tail of the STL, by contrast, more often contains cleanup code, final result checks, or padding that is loosely coupled to individual fault detections and therefore more likely to be removable. By starting at the bottom and working upward, the algorithm harvests the easiest removals first and reaches the protected setup region only after the low-hanging fruit has been collected. The bottom-to-top traversal also makes the process deterministic and reproducible: given the same STL and configuration, two runs of A1xx always produce the same sequence of trials and the same final result — a property that matters in a safety-critical context where the compaction process may need to be audited.

### 3.2.2 Block structure and selection order

**Partitioning the candidate list into blocks.** After preprocessing has produced the pruned candidate list, A1xx divides it into a sequence of non-overlapping consecutive blocks, each containing exactly  $k$  instructions. The parameter  $k$  is the *segment dimension* and is set in the algorithm configuration. The candidate list is an ordered sequence of instructions as they appear in the STL. The first block is formed from the first  $k$  instructions in that sequence, the second block from the next  $k$ , and so on until all candidates are assigned. If the number of candidates is not a multiple of  $k$ , the last block holds the remaining instructions and contains fewer than  $k$  members. This partitioning is performed once before the compaction loop begins and is not updated as instructions are removed during the run.

**Block ordering.** Once all blocks have been formed, the entire block list is reversed so that the algorithm begins with the block at the *bottom* of the STL and works upward toward the beginning. As discussed in Section 3.2, this ordering targets the most likely redundant instructions first, preserving the fault-activating sequences near the top of the program for as long as possible.

**Block removal and fallback policies.** Within each block, the algorithm follows a two-phase procedure. In the first phase, all  $k$  instructions in the block are removed from the STL and staged in a candidate list. The full block is then evaluated: compile, logic simulation, and fault simulation are run on the STL with all  $k$  instructions absent. If the full block passes the acceptance test, all  $k$  instructions are permanently removed and the algorithm moves to the next block.

If the full-block removal fails, the algorithm enters the second phase: a fallback that restores one instruction at a time and re-evaluates the smaller remaining sub-block after each restoration. This continues until either a

sub-block is accepted or every instruction has been restored and the block contributes zero removals.

The key design parameter is *which* instruction gets restored on each failed attempt, and this is controlled by a configurable *block fallback policy*. Three variants are supported (Figure 3.2):

- **Policy F (Forward):** instructions are staged from the bottom of the block upward, so the fallback restores the topmost instruction first. The sub-block that is attempted shrinks from its top on each failure. For a block  $[A, B, C]$  (A topmost, C bottommost), the trial sequence is:  $\{A, B, C\} \rightarrow \text{restore } A \rightarrow \{B, C\} \rightarrow \text{restore } B \rightarrow \{C\} \rightarrow \text{restore } C$ .
- **Policy B (Backward):** instructions are staged from the top of the block downward, so the fallback restores the bottommost instruction first. The sub-block shrinks from its bottom on each failure. For the same block:  $\{A, B, C\} \rightarrow \text{restore } C \rightarrow \{A, B\} \rightarrow \text{restore } B \rightarrow \{A\} \rightarrow \text{restore } A$ .
- **Policy R (Random):** instructions are staged in a randomly shuffled order, so the fallback restoration sequence is also random. This sacrifices determinism in favour of avoiding any systematic directional bias in the fallback.

All three policies attempt the full block first and degrade gracefully through smaller sub-blocks; they differ only in which residual instructions remain in the candidate set after each failed attempt. Figure 3.2 illustrates the trial sequences for Policy F and Policy B side by side for a three-instruction block.

### 3.2.3 Acceptance policies

After each trial, the algorithm compares the STL’s new statistics against the previous ones and decides whether to accept the removal (Proceed) or undo it (Restore). The acceptance test applies two conditions simultaneously: TaT must not increase ( $\Delta T_{aT} \leq 0$ ), and OC must not fall below a configurable threshold ( $\Delta FC \geq -\Delta$ , where  $\Delta = 0$  for Maximize). Both conditions must hold for a Proceed verdict; if either fails, the trial is rejected. Figure 3.3 visualises the resulting accept and reject regions in the  $(\Delta T_{aT}, \Delta FC)$  plane for both policies. The same acceptance logic is shared by A1xx and A2xx; only the block construction strategy differs between them.

Two acceptance policies govern the coverage threshold and are the subject of this section. A third policy — Threshold — exists in the implementation but is not used in the experiments of this thesis and is not discussed further.

|                   | Policy F (Forward)                       | Policy B (Backward)                      |
|-------------------|------------------------------------------|------------------------------------------|
| <b>Trial 1</b>    | <div>A</div> <div>B</div> <div>C</div>   | <div>A</div> <div>B</div> <div>C</div>   |
| <b>Trial 2</b>    | <div>A ↵</div> <div>B</div> <div>C</div> | <div>A</div> <div>B</div> <div>C ↵</div> |
| <b>Trial 3</b>    | <div>A</div> <div>B ↵</div> <div>C</div> | <div>A</div> <div>B ↵</div> <div>C</div> |
| <b>No removal</b> | <div>A</div> <div>B</div> <div>C ↵</div> | <div>A ↵</div> <div>B</div> <div>C</div> |

Figure 3.2: Fallback trial sequences for Policy F (Forward) and Policy B (Backward) applied to a block  $[A, B, C]$  where  $A$  is the topmost instruction. Grey: instruction currently removed from the STL. Orange with  $\leftarrow$ : instruction being restored at this step. White: instruction already restored and present in the STL. Each trial is only reached if the previous one failed the acceptance test.

**Maximize policy.** Under the Maximize policy the coverage threshold for each trial is set equal to the coverage of the most recently accepted STL state. Formally, if  $cov_{\text{prev}}$  denotes the coverage after the last accepted removal, a new trial is accepted only if  $cov_{\text{new}} \geq cov_{\text{prev}}$ . When a trial is accepted,  $cov_{\text{prev}}$  is updated to  $cov_{\text{new}}$  before the next trial begins. This means the coverage bar tracks the current STL state and never allows coverage to drop from one accepted removal to the next. In practice the Maximize policy may produce zero coverage change (the STL coverage remains at or above the initial level throughout the run) but it limits the compaction it can achieve: any block whose removal would cause even a marginal coverage drop is always rejected, regardless of how small that drop is.

**Range policy.** The Range policy relaxes the per-trial constraint by permitting coverage to decrease by up to a configurable tolerance  $\Delta$  per step, subject to a hard global floor. At the start of the run the floor is set to  $cov_{\text{init}} - \Delta$ , where  $cov_{\text{init}}$  is the coverage of the original STL. A trial is then accepted if:

$$cov_{\text{new}} \geq \max(cov_{\text{prev}} - \Delta, cov_{\text{init}} - \Delta)$$

The max ensures that even if the per-step limit  $cov_{\text{prev}} - \Delta$  would permit a very low value, the global floor  $cov_{\text{init}} - \Delta$  is never violated. Because coverage can only decrease during compaction, the floor is in practice always the binding constraint: OC is guaranteed never to fall more than  $\Delta$  below the original STL’s coverage, regardless of how many individual steps each reduce coverage slightly. For example, with  $\Delta = 0.01$  (one percentage point), the compacted STL is guaranteed to retain at least  $cov_{\text{init}} - 1\%$  Observational Coverage.

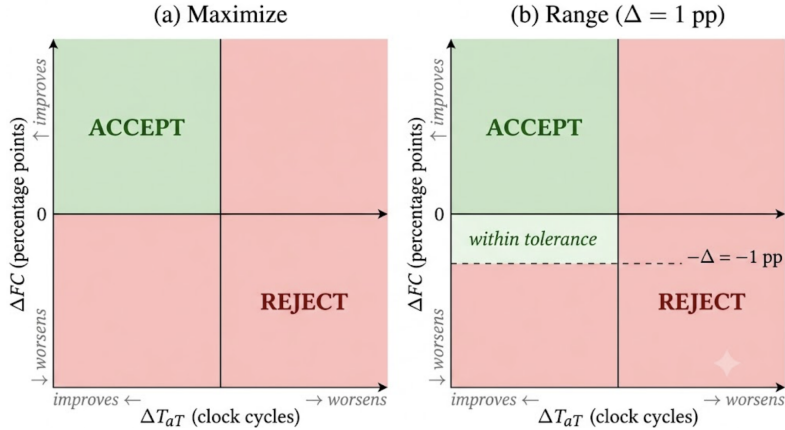


Figure 3.3: Accept/reject regions in the  $(\Delta T_{aT}, \Delta FC)$  plane. (a) Maximize: both metrics must not worsen. (b) Range ( $\Delta = 1$  pp): a coverage drop up to  $-\Delta$  is tolerated provided TaT does not increase.

**Trade-off between the two policies.** The Maximize policy offers a strict guarantee: the compacted STL is never worse than the original in OC terms. The Range policy offers a weaker but still bounded guarantee — OC degrades by at most  $\Delta$  — while giving the algorithm more freedom to accept removals it would otherwise reject. In practice this additional freedom can translate into greater instruction removal, particularly for blocks that cover faults already detected by other sequences but with a marginal overlap that causes a tiny OC drop. Which policy is preferable depends on the application: Maximize is appropriate when any coverage regression is unacceptable, while Range is appropriate when a small, controlled degradation is permissible in exchange for a more compact STL. Both policies are evaluated in Chapter 5, where each block selection strategy is paired with each acceptance policy.

### 3.2.4 Block size

The segment dimension  $k$  controls how many instructions form a single block and is the primary lever for tuning the aggressiveness of each removal attempt. Its value determines a fundamental trade-off between per-trial cost and per-trial success rate.

A larger  $k$  reduces the number of blocks in the candidate list and therefore the number of trials needed to sweep the entire STL once. Since each trial requires at minimum a cross-compilation and a logic simulation, and often a full fault simulation, fewer trials translate directly into lower total CPU time per sweep. At the same time, a larger block is more likely to contain at least one instruction that the acceptance test cannot tolerate losing — whether because it lies in a fault activation window that preprocessing missed, or because it is the sole contributor to a specific fault detection. A single such instruction is sufficient to cause the full-block trial to fail, forcing Policy F (or B or R) to fall back through smaller sub-blocks until either a safe sub-block is found or the entire block is restored. In the worst case, every trial for that block fails and the CPU cost of all its sub-block attempts is wasted.

A smaller  $k$  has the opposite profile: more blocks, more trials, higher CPU cost per sweep, but each block is a smaller and therefore less risky target. In the limit  $k = 1$ , A1xx degenerates to single-instruction removal and becomes behaviourally equivalent to A0, with the only difference being the bottom-to-top traversal order.

The choice of  $k$  therefore involves a judgment about the density of critical instructions in the STL after preprocessing. If preprocessing removes most of the protected instructions and leaves a candidate list that is largely redundant, a larger  $k$  is efficient: most full-block trials will pass and the algorithm makes rapid progress. If the candidate list still contains many

instructions that are individually load-bearing, a smaller  $k$  reduces the fallback overhead. A value of  $k = 3$  represents a practical balance: it groups enough instructions to realise the CPU time advantage of block removal, while keeping each block small enough that the fallback overhead remains modest. The specific values used in the experiments of this thesis are reported in Chapter 4.

### 3.3 A2xx: Random non-adjacent block compaction

#### 3.3.1 Motivation

A1xx’s systematic bottom-to-top sweep has a clear strength: every contiguous group of  $k$  candidates is attempted exactly once, so no adjacency opportunity is ever missed. However, this deterministic structure also means that the algorithm is sensitive to how redundant instructions are distributed in the STL. If removable instructions are not concentrated at the bottom but instead scattered throughout the program, A1xx encounters many consecutive Restore verdicts before it reaches them, and the fixed traversal order means every run produces the same sequence of failures. There is no mechanism by which A1xx can bypass a difficult region and try a different part of the STL instead.

A2xx addresses this by abandoning both the adjacency constraint and the directional ordering. Rather than forming blocks from consecutive instructions, A2xx assembles each block from instructions drawn at random from the candidate list. Rather than sweeping bottom-to-top, it processes blocks in the random order they were formed. The result is a search strategy with no systematic directional bias: every run explores a different ordering of the candidate space, and instructions that are far apart in the source can appear in the same block.

The non-adjacency property opens a class of removal opportunities that A1xx cannot reach. Consider three consecutive candidates  $[X, Z, Y]$  where  $X$  and  $Y$  are individually removable but  $Z$  is critical — its removal causes a coverage drop. Because A1xx forms consecutive blocks, all three end up in the same block. Policy F’s fallback stages  $Y$  first, then  $Z$ , then  $X$ ; on the inevitable failure of the full-block trial it restores  $X$ , tries  $\{Z, Y\}$  (fails), restores  $Z$ , tries  $\{Y\}$  alone (passes), and stops — leaving  $X$  permanently in the STL. Policy B produces the mirror result: only  $X$  is removed and  $Y$  is missed. Since A1xx partitions candidates once and never revisits a block, the missed instruction is lost for the entire run. A2xx, after shuffling, may form a block  $[X, Y]$  without  $Z$ , removing both in a single accepted trial. The advantage is not a synergy between  $X$  and  $Y$  — each is individually safe to remove — but the ability to group them away from the critical instruction

that A1xx’s fixed adjacency forces into every nearby window.

The trade-off is that A2xx gives up A1xx’s exhaustive guarantee. Because block membership is random, there is no assurance that every possible  $k$ -subset of candidates will be attempted. A given run may completely miss certain groupings that would have passed the acceptance test. Across different runs with different random seeds, the compaction results may also differ. A2xx is therefore best understood not as a replacement for A1xx but as a complementary strategy: where A1xx provides systematic coverage of all contiguous windows, A2xx provides stochastic exploration of non-adjacent combinations.

### 3.3.2 Block construction

A2xx constructs its blocks before the compaction loop begins, in a single randomisation step applied to the pruned candidate list. The procedure is as follows.

First, the candidates are grouped by assembly file. Each block is constrained to contain instructions from a single file, because the STL may consist of multiple source files and an instruction’s line number is meaningful only within its own file. This constraint ensures that when the algorithm removes or restores instructions within a block, it modifies a single file consistently.

Within each assembly file, the candidate list for that file is shuffled uniformly at random using a single call to `random.shuffle`. The shuffled list is then partitioned into consecutive non-overlapping slices of size  $k$ , exactly as in A1xx. Because the list was shuffled before slicing, the instructions within each slice are not necessarily adjacent in the original source; they are  $k$  instructions drawn from arbitrary positions in the candidate pool of that file. The last slice may contain fewer than  $k$  instructions if the file’s candidate count is not a multiple of  $k$ .

This construction means that two runs of A2xx on the same STL with different random seeds will produce different block compositions and a different block processing order, and may therefore produce different compaction results. The acceptance logic and fallback policies (Section 3.2.3) are identical to those used by A1xx, so the only difference between A1xx and A2xx lies in how blocks are formed and in what order they are visited. Figure 3.4 illustrates this difference side by side.

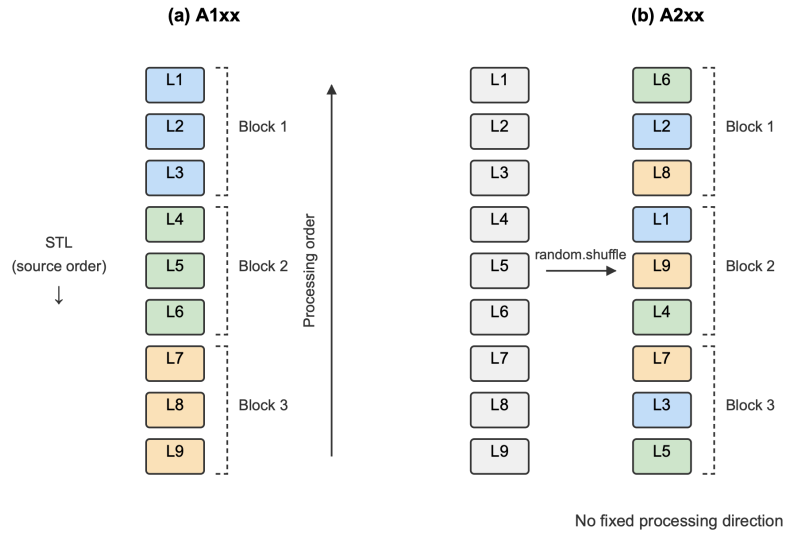


Figure 3.4: Block formation comparison for A1xx and A2xx on a nine-instruction candidate list (L1–L9 in source order). (a) A1xx partitions the list into adjacent consecutive blocks and processes them bottom-to-top. (b) A2xx applies `random.shuffle` first; the same slicing then produces blocks whose members are non-adjacent in the source. Block processing follows the shuffled order with no fixed direction.

### 3.4 Early-exit optimisation

Two independent mechanisms reduce wasted computation by terminating work early — one at the level of a single trial, the other at the level of the entire compaction run.

**TaT-conditioned fault simulation skip.** Each trial follows a fixed pipeline: cross-compile, logic simulate, fault simulate, evaluate. Of these four steps, fault simulation is by far the most expensive — a single fsim campaign on the CV32E40P STLs can take several minutes. The key observation is that the acceptance test requires *both* TaT and OC to satisfy their respective constraints. If TaT has already worsened after logic simulation — that is, the new TaT exceeds the previous accepted value — the trial is guaranteed to fail the acceptance test regardless of what fault simulation returns. Running fsim in this case is pure wasted cost: it cannot change the verdict from Restore to Proceed.

The early-exit optimisation exploits this by checking TaT immediately after logic simulation completes. If  $T_{aT, \text{new}} > T_{aT, \text{prev}}$ , the trial is immediately recorded as Restore and fault simulation is skipped entirely. The STL is restored to its previous state and the algorithm moves to the next sub-block or block. This decision is logged in the statistics CSV as `fsim_ok = SKIP`, making it possible to measure how often the optimisation fires across a run. Over a full compaction campaign spanning hundreds of trials, the accumulated fsim time saved can be substantial, particularly for STLs where many candidate blocks cause TaT regressions.

**Consecutive restore threshold.** The second mechanism operates at a coarser granularity: it monitors the number of consecutive Restore verdicts across successive trials and terminates the entire compaction run if that count reaches a configurable threshold  $N$ . After every Restore the counter is incremented; after every Proceed it is reset to zero. When the counter reaches  $N$ , the algorithm logs a warning and exits the compaction loop, leaving the STL in its current (partially compacted) state.

The motivation is pragmatic. If the algorithm has failed to accept any removal across  $N$  consecutive trials, it is likely that the remaining candidate blocks contain few or no removable instructions — either because the search space is genuinely exhausted or because the current block ordering consistently places critical instructions in every block. Continuing the run in this state produces a long sequence of costly failures without improving the result. Terminating early recovers the remaining CPU budget.

The trade-off is that a finite threshold may cause the algorithm to stop

before reaching removable instructions that appear later in the block sequence. A small  $N$  reduces CPU time aggressively but risks premature termination; a large  $N$  tolerates longer dry spells at the cost of additional trials. Setting  $N = 0$  disables the mechanism entirely, allowing the algorithm to run until all blocks have been visited. The effect of different threshold values on compaction quality and runtime is evaluated experimentally in Chapter 5.

## 3.5 Fault filtering

### 3.5.1 Motivation

The TaT-conditioned early exit described in Section 3.4 eliminates fault simulation for trials that logic simulation has already condemned. However, it has no effect on trials that pass the TaT check — and those trials, which are the ones that matter, each incur a full fault simulation campaign. On the CV32E40P STLs used in this thesis the fault list contains thousands of stuck-at faults; simulating all of them on a modified STL typically takes several minutes per trial. Since each compaction run may execute hundreds of such trials, fault simulation is the overwhelmingly dominant cost of the entire compaction process.

The key observation that motivates fault filtering is that removing a candidate block at a given position in the STL can only affect faults activated by instructions at *that position or later*. Faults whose entire activation window lies in instructions appearing *before* the candidate block are determined by program execution that is untouched by the removal — those earlier instructions still run, in the same order, producing the same processor state — so, under the locality assumption used by this heuristic, their detection status is not expected to change. Similarly, faults with any mapping to a different assembly file are assumed to be unaffected by modifications to the current file. Both assumptions do not hold universally: loops, control-flow interactions, or shared register state across files can invalidate them. They are treated as heuristically valid for the straight-line test fragments typical of the benchmark STLs. For a candidate block of three instructions at a specific position, both of these categories together account for the large majority of the total fault list. Simulating them on every trial is therefore redundant: their contribution to the OC metric is fixed and already known from the most recent full fault simulation.

Fault filtering exploits this locality by constructing, for each trial, a reduced fault list that excludes two categories of detected faults. The first category contains faults that are *exclusively earlier*: every source line associated with the fault in the current assembly file appears strictly before

the first instruction of the candidate block — meaning the fault has no activation point at or after the block’s position and cannot be affected by its removal. The second category contains faults that have any mapping to a different assembly file, whose activation is assumed to be independent of the current file. Any detected fault that has at least one activation point at or after the candidate block’s position in the current file is kept in the simulation, since its status could change if the block is removed. Undetected faults are always kept regardless of their position, as a safety measure to avoid masking any potential coverage change. The statuses of the excluded faults are taken from a pre-computed baseline established by the most recent full fault simulation. The reduced list is then passed to VC-Z01X, which simulates only the remaining faults. After the reduced simulation completes, the full Observational Coverage is reconstructed analytically by combining the simulation counts from the reduced set with the baseline status counts for the excluded faults. From the algorithm’s perspective the acceptance decision is made on the same OC metric as before; the difference is that it is computed from a much smaller simulation, running in a fraction of the time.

The information needed to perform this filtering — specifically, which faults are activated by which source lines — is already produced as a by-product of the preprocessing step. Preprocessing generates a mapping from assembly source locations to the list of detected faults whose activation window includes that line, saved as `mapping_reverse.json`. Fault filtering reads this mapping at runtime to identify, for each trial, which faults are associated only with earlier instructions or with other STL files, and excludes those faults from the fault list passed to VC-Z01X. No additional simulation or analysis is required beyond what preprocessing already provides.

### 3.5.2 Fault-instruction reverse mapping

The data structure that makes fault filtering possible is `mapping_reverse.json`, produced alongside the preprocessing step and loaded once at the start of the compaction run. Each entry in the file is keyed by an assembly source location — a filename and a zero-based line number — and records the list of detected faults whose activation window includes that line. For each fault the entry stores the port name and stuck-at type, which together identify it unambiguously in the VC-Z01X fault list.

The file is constructed by `fault_instruction_mapper.py` using the same three inputs as preprocessing: the golden fault simulation report, the processor execution trace, and the compiled ELF binary. For each detected fault, the mapper applies the same activation-window procedure described in Section 3.1: it reads the fault’s activation attributes (simulation timestamp and program counter), locates the corresponding entry in the trace database,

and retrieves the surrounding *history* entries that represent the instructions simultaneously in the processor pipeline at that moment. Each of those trace entries carries a binary program counter value, which is translated to an assembly source file and line number via the ELF debug symbols. The fault is then recorded under every source line that belongs to its window. Because this analysis operates on the same golden fault simulation that preprocessing already required, producing the mapping adds no additional simulation cost.

When the compaction run initialises, the JSON file is loaded and used to build two in-memory lookup tables: one that maps each instruction to the faults associated with it, and one that maps each fault to the instructions associated with it. Both directions are derived from the same data in a single pass.

The two structures serve distinct roles during fault filtering. `fault_to_instructions` is the primary lookup for the exclusion decision. For each candidate block, two categories of detected faults are excluded and removed from the fault list passed to VC-Z01X: faults whose every mapped position in the current assembly file lies strictly before the candidate block, and faults that have any mapping to a different assembly file. Both categories are excluded under the heuristic locality assumption — their statuses are taken from the pre-computed baseline rather than re-simulated. All remaining detected faults — those with at least one mapped position at or after the candidate block in the current file — are kept in the simulation. `instruction_to_faults` supports the complementary direction: it enables a fast look-up of which faults are associated with a specific instruction, used during baseline refresh when fault statuses are updated after a full fault simulation.

### 3.5.3 Per-trial fault set reduction

Fault set reduction runs once per trial, immediately after logic simulation has confirmed that TaT does not worsen and before fault simulation is launched. At this point the candidate block — the set of instructions staged for removal — is known, and its lowest line number in the current assembly file is used as the reference position. This is the boundary: any detected fault whose every mapped position in the current file lies strictly before this line, or that have any mapping to a different assembly file, is a candidate for exclusion.

The algorithm calls `_get_excludable_faults()` with the candidate block's assembly file identifier and reference line number. This function uses the `fault_to_instructions` lookup built in Section 3.5.2 to identify the two excludable categories and returns the list of detected faults to be removed from the simulation. Undetected faults are never included in this list, as their exclusion could mask a coverage loss that the trial would otherwise detect. A safety guard is also applied: if the returned list would cover the

entire fault set, the exclusion is cancelled for that trial and the full fault list is simulated as normal.

The excluded faults are applied to VC-Z01X through the fault generation configuration file (`gen_saf`). Before each fault simulation, the algorithm injects an `Exclude` block into this file listing the ports and stuck-at types of all faults to be skipped. VC-Z01X reads this block and omits the listed faults from the simulation entirely, so the campaign runs only over the reduced fault set. After simulation completes, the raw coverage returned by VC-Z01X reflects only the non-excluded faults; the contribution of the excluded faults is reintroduced analytically in the coverage adjustment step described in Section 3.5.4.

### 3.5.4 Coverage adjustment and baseline refresh

After the reduced fault simulation completes, the raw coverage returned by VC-Z01X reflects only the non-excluded faults. To recover the full Observational Coverage, the algorithm combines the status counts from the reduced simulation with the baseline status counts of the excluded faults. Concretely, both sets are tallied into the five standard groups — DD, DN, DA, NA, SU — and the OC formula is applied to the union:

$$\text{OC} = \frac{(DD + DN)_{\text{sim}} + (DD + DN)_{\text{excl}}}{(NA + DA + DN + DD + SU)_{\text{sim}} + (NA + DA + DN + DD + SU)_{\text{excl}}}$$

where the subscript `sim` denotes counts from the current reduced simulation and `excl` denotes counts taken from the pre-computed baseline for the excluded faults. The result approximates what a full fault simulation would have produced, provided the baseline statuses of the excluded faults are still accurate. This condition holds under the locality assumptions of the heuristic: excluded faults — those mapped exclusively to earlier instructions, or those with any mapping to a different assembly file — are *assumed* to be unaffected by the removal of the candidate block, so their detection status is not expected to change. This is not guaranteed in the presence of loops, control-flow interactions, or shared state across files.

The baseline is established once at the start of the compaction run from the golden fault simulation. As the run progresses and instructions are permanently removed, the STL changes, and fault statuses that were stable at the start can shift: a fault previously detected by a sequence that was later removed may become undetected. If the baseline is not refreshed, the status counts used for excluded faults will become stale and the reconstructed OC will diverge from the true value. To prevent this drift, the algorithm periodically runs a full, unfiltered fault simulation — with no `Exclude` block

in `gen_saf` — and uses its results to rebuild the baseline fault statuses. This refresh is triggered every  $R$  accepted fault simulations, where  $R$  is a configurable interval (`full_fsim_refresh_interval`). The cost of these periodic full simulations is amortised across the trials between refreshes, so the overall saving from fault filtering remains significant even when accounting for them.

Figure 3.5 summarises the complete per-trial fault filtering flow, from the candidate block identification through fault set reduction, reduced simulation, coverage reconstruction, and periodic baseline refresh.

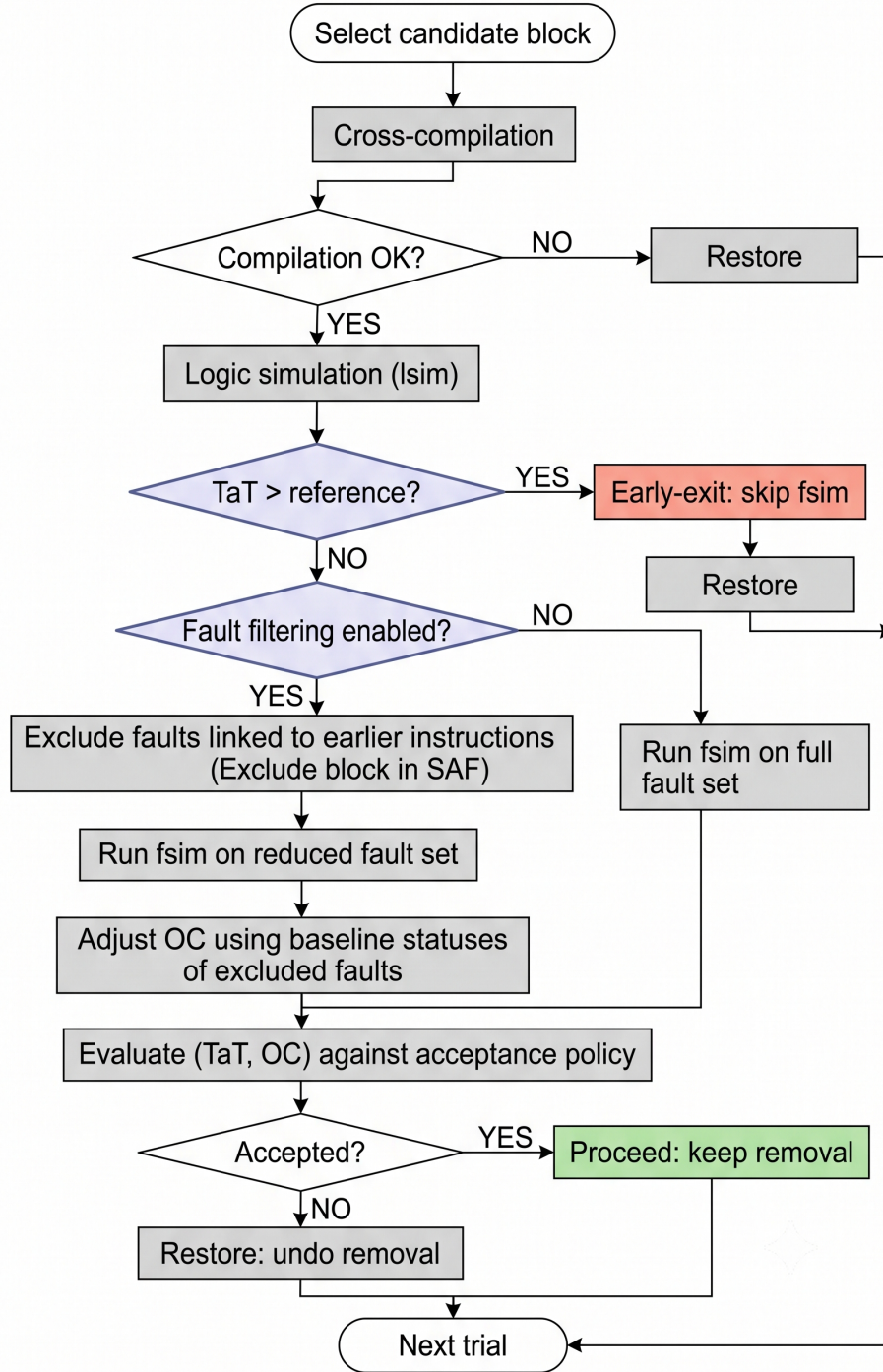


Figure 3.5: Per-trial fault filtering flow. For each trial that passes the TaT check, the reduced fault list is constructed and injected into the fault generation configuration before VC-Z01X is launched. After simulation the full OC is reconstructed analytically. Every  $R$  accepted simulations a full fsm refreshes the baseline.

## Chapter 4

# Experimental Setup

### 4.1 Target Design and Fault Model

#### 4.1.1 The CV32E40P processor

The circuit under test in this thesis is the **CV32E40P**, an open-source, in-order 32-bit processor implementing the RISC-V RV32IMC instruction set architecture. It was originally developed as RI5CY within the PULP (Parallel Ultra-Low-Power) project at ETH Zurich and the University of Bologna, and is now maintained by the OpenHW Group. The processor has been adopted in a range of embedded and ultra-low-power applications, and its open-source nature makes it a standard reference design in academic research on processor testing and verification.

The pipeline of the CV32E40P consists of four stages: instruction fetch (IF), instruction decode (ID), execute (EX), and a write-back/memory access stage. Figure 4.1 illustrates this structure. The IF stage fetches and latches instructions from program memory. The ID stage decodes each instruction and reads the required operands from the 32-entry integer register file. The EX stage contains three parallel functional units — the ALU, the multiplier (MULT), and the load/store unit (LSU) — whose outputs are selected by a multiplexer before being forwarded to the WB/Memory Access stage. Load and store instructions additionally interact with an external data memory through the LSU. Finally, the WB stage writes the computed result back to the register file, completing the instruction. The processor supports the base integer instruction set (I), the standard integer multiplication and division extension (M), and the compressed instruction extension (C), which reduces code size by encoding common operations in 16-bit words.

From a safety perspective, the CV32E40P is representative of the class of

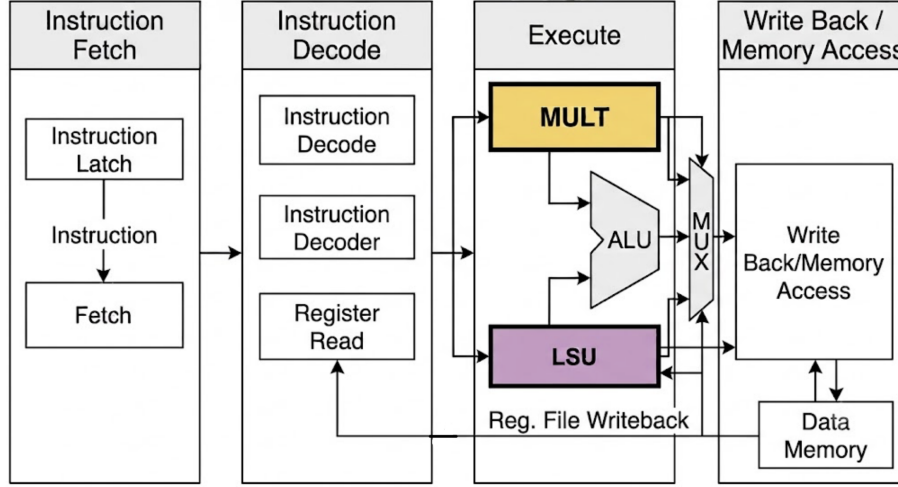


Figure 4.1: Pipeline structure of the CV32E40P processor. The four stages are instruction fetch (IF), instruction decode (ID), execute (EX), and write-back/memory access (WB). The two modules targeted by fault injection in this thesis — MULT (yellow) and LSU (purple) — are highlighted inside the EX stage.

embedded processor cores that appear in automotive and industrial control systems, where in-field reliability is a certification requirement. The combination of a non-trivial datapath (with multiplication, memory accesses, and pipeline interlocks) and a manageable size makes it a well-suited target for evaluating STL compaction: the fault universe is large enough to make compaction meaningful, but small enough that iterative fault simulation within a compaction loop remains computationally feasible.

#### 4.1.2 Gate-level simulation

All experiments in this thesis are performed at the *gate level*, meaning that the circuit model used in simulation is a technology-mapped netlist obtained by synthesizing the CV32E40P RTL description. Gate-level simulation is more representative of the physical device than RTL simulation for two reasons. First, it captures the actual logic cells, their interconnections, and the exact signal names as they will appear in silicon, which allows fault injection to target real physical nodes rather than abstract RTL variables. Second, stuck-at faults, the fault model used here, are defined on the ports of primitive logic gates; they can only be properly applied and simulated on a gate-level netlist. The netlist is compiled once and reused across all compaction iterations: the STL firmware changes with every candidate removal, but the hardware model remains fixed.

### 4.1.3 Modules under test

Rather than targeting the entire processor, each STL in this thesis is scoped to one of two specific internal modules. This targeted approach is standard practice in industrial in-field testing: a large processor has many sub-units, and it is more efficient to develop and maintain separate STLs for each module than to produce one monolithic test that covers everything at once.

The two modules used in this work are:

- **Multiplier (MULT):** implemented in `cv32e40p_mult.sv` as a sub-word multiply-and-accumulate (MAC) unit. It handles the RV32M multiplication instructions (`MUL`, `MULH`, `MULHSU`, `MULHU`) as well as dot-product and accumulate operations. The combinational and sequential logic inside this module implements multi-cycle multiplication using partial products, making it a relatively dense datapath that is non-trivial to test completely. Three of the six benchmark STLs target this module (STL1, STL2, STL6).
- **Load/Store Unit (LSU):** implemented in `cv32e40p_load_store_unit.sv`. It manages all data memory accesses generated by load and store instructions, handling byte, halfword, and word accesses, as well as misalignment correction and OBI (Open Bus Interface) protocol handshaking. The LSU sits on the critical path between the EX stage and the data memory interface, so faults here can corrupt any data value written to or read from memory. The remaining three benchmark STLs target this module (STL3, STL4, STL5).

Restricting fault injection to one module at a time means that the fault simulation scope—and therefore the fault universe on which FC is computed—is different for MULT-targeting STLs and LSU-targeting STLs. This is reflected in the varying baseline FC values shown in Table 4.2: the two sets are not directly comparable in absolute terms, and results are always discussed within each group.

### 4.1.4 Stuck-at fault model

The fault model used throughout this thesis is the SAF model, the most widely used fault model in digital circuit testing. A stuck-at-0 (SA0) fault on a signal forces that signal to remain permanently at logic 0 regardless of what the circuit would drive in the fault-free case; a stuck-at-1 (SA1) fault does the same at logic 1. Together, SA0 and SA1 faults on every port of every gate in the target module constitute the fault universe for a given experiment.

In the context of in-field testing, stuck-at faults are used to model permanent defects that can arise from aging, electromigration, or latent manufacturing

imperfections that only manifest after deployment. Although more advanced fault models exist (such as transition delay faults), the SAF model remains the baseline in functional safety standards and is the one supported by the STLs benchmarked here.

Fault injection is specified in VC-Z01X using a Synopsys Fault Specification File (`.sff`). For each experiment, the `.sff` file instructs the fault campaign compiler to generate SA0 and SA1 faults on all PORT-level signals in the chosen module hierarchy. For the MULT module, the fault scope is `cv32e40p_top.core_i.ex_stage_i.mult_i.**`; for the LSU, it is `cv32e40p_top.core_i.load_store_unit_i.**`. The `**` wildcard covers all signals at any depth within the module, so every internal net and output port is included.

#### 4.1.5 Coverage metric: Observational Coverage

The quality of an STL after each compaction step is measured using a coverage formula computed directly from the VC-Z01X fault simulation report. VC-Z01X assigns each fault a two-character status code after simulation, encoding both an *observation* dimension (whether the fault effect reached a software-visible output) and a *diagnostic* dimension (whether the effect was unambiguously attributed to that specific fault). The status codes are organised into five groups, summarised in Table 4.1.

This thesis uses *Observational Coverage* (OC) as the primary FC metric, defined as:

$$OC = \frac{DD + DN}{NA + DA + DN + DD + SU} \quad (4.1)$$

The numerator counts all faults that produced an observable effect on any software-visible signal, regardless of whether the effect was uniquely diagnosed. The denominator includes the full fault universe plus the safe-unobserved faults, penalizing tests that leave a significant fraction of faults unobserved. OC is stricter than Diagnostic Coverage (which only counts DD faults in the numerator) and is better suited to in-field testing, where the goal is to *detect* that something is wrong rather than to pinpoint the exact faulty node. All FC values reported in Chapter 5 are Observational Coverage values computed from Equation (4.1).

## 4.2 Simulation Environment

### 4.2.1 Execution environment

The experiments were executed on the SSH server `bello.darknet.local`, running Rocky Linux 8.10 (Green Obsidian) with Linux kernel `4.18.0-553.124.1.el8_10.x86_64`.

Table 4.1: VC-Z01X fault status groups and their role in the Observational Coverage formula.

| Code | Full name               | Meaning                                                                                        | Role in OC    |
|------|-------------------------|------------------------------------------------------------------------------------------------|---------------|
| DD   | Dangerous Diagnosed     | Effect observed and uniquely attributed to this fault.                                         | Num. + denom. |
| DN   | Dangerous Not Diagnosed | Effect observed but not uniquely attributed (multiple faults produce the same output pattern). | Num. + denom. |
| DA   | Dangerous Assumed       | Treated as dangerous by assumption; no direct simulation evidence of observation.              | Denom. only   |
| NA   | Not Applied             | Fault was never activated during simulation.                                                   | Denom. only   |
| SU   | Safe Unobserved         | Fault was activated but its effect did not propagate to any observable output.                 | Denom. only   |

The server is equipped with two Intel(R) Xeon(R) Gold 6238R CPUs operating at 2.20 GHz. Each socket provides 28 physical cores with two hardware threads per core, for a total of 56 physical cores and 112 logical CPUs. The simulation flow used Synopsys VCS U-2023.03-SP2\_Full64 for gate-level logic simulation. The CV32E40P RTL was synthesized using Synopsys Design Compiler (DC Ultra) version U-2022.12, targeting the NanGate Open Cell Library (typical CCS corner, `NangateOpenCellLibrary_typical_ccs_scan.db`). Since all experiments were carried out on this server, CPU time figures reported in Chapter 5 are directly comparable across configurations.

#### 4.2.2 Overview of the validation pipeline

Every compaction candidate goes through the same three-step validation chain before a verdict is issued: (1) the modified STL is cross-compiled into firmware, (2) the firmware is executed on the gate-level processor model

under Synopsys VCS to measure TaT and confirm correct termination, and (3) if and only if logic simulation succeeds and TaT does not worsen, VC-Z01X fault simulation is run to recompute FC. A candidate is accepted only when all three steps succeed and the resulting (TaT, FC) pair satisfies the configured acceptance policy. This section describes each step in detail.

### 4.2.3 Cross-compilation

The STL is written in RISC-V assembly and must be compiled into a binary that the simulated processor can execute. Cross-compilation uses the **riscv32-corev-elf-gcc** toolchain, which targets the CORE-V extended RISC-V ABI used by the CV32E40P. The architecture flag passed to the compiler is:

```
-march=rv32imc_zicsr_xcvalu_xcvbi_xcvbitmanip_xcvhwlp_xcvmac_xcvmem_xcvsimd
```

This flag enables the base integer, multiplication, compressed, and control-status-register extensions (RV32IMC + Zicsr) together with the CORE-V custom extensions (**xcv\***) that the CV32E40P hardware supports. The optimization level is **-Os** (optimize for size), and **-nostdlib** prevents the standard C library from being linked, keeping the binary self-contained. After compilation, the resulting ELF file is converted to a Verilog-readable Intel hex format using:

```
riscv32-corev-elf-objcopy -O verilog sbst.elf sbst.hex
```

The hex file is what the testbench loads into the simulated RAM at the start of each simulation run. The entire build is driven by a **make** system: TestCrush invokes **make -C sbst clean** followed by **make -C sbst all**, which cleans any previous build artifacts and rebuilds the firmware from the current STL sources. This ensures that each candidate is evaluated from a clean state, with no risk of stale object files masking changes. If the **make** invocation produces any output on stderr (other than warnings), the candidate is immediately rejected as a compilation failure and fault simulation is not attempted.

### 4.2.4 Logic simulation with VCS

Once the firmware hex file is produced, logic simulation is launched via:

```
make -C ${root_dir} vcs/sim/gate/shell
```

This target invokes the pre-compiled VCS simulation binary (**simv**) in gate-level mode, passing the firmware path as a plusarg:

```
./simv +firmware=sbst/sbst.hex
```

The testbench reads the hex file into the processor's instruction and data RAM using `$readmemh`, then releases reset and allows the processor to begin fetching instructions. From that point, the simulation runs autonomously.

**TaT measurement.** The testbench contains a free-running clock-cycle counter (`cycle_cnt_tat`) that increments on every rising clock edge from reset deassertion onward. When the simulation reaches its `final` block, the testbench prints:

```
[TESTBENCH] time: test application time = N clock cycles (M
ns)
```

TestCrush captures  $N$  from this line using the regular expression `test application time = ([0-9]+)`, with the integer match in capture group 1. The value is in *clock cycles*, making it independent of the simulation time resolution or any particular clock frequency assumption.

**Termination detection.** The testbench monitors a pair of signals (`exit_valid`, `exit_value`) driven by a *tohost* mechanism: the STL writes a specific value to a designated memory address upon completion, which the testbench decodes into an exit code. On each rising clock edge, the testbench checks these signals. When `exit_valid` is asserted and `exit_value` equals zero, the testbench prints `EXIT SUCCESS` and calls `$finish`. Any non-zero exit value produces `EXIT FAILURE` and also terminates simulation. TestCrush detects correct termination by searching stdout for the pattern `EXIT\sSUCCESS`; if this pattern is absent, the candidate is rejected regardless of any other output.

**Timeout.** To prevent a pathological STL modification (for example, one that removes a return instruction and causes an infinite loop) from stalling the compaction run indefinitely, each logic simulation invocation is subject to a per-run wall-clock timeout. The configured values are 120 seconds for A1xx and A2xx runs and 240 seconds for A0 runs. If the timeout expires before `$finish` is reached, TestCrush kills the process group and records the trial as a logic simulation failure.

#### 4.2.5 Fault simulation with VC-Z01X

Fault simulation is the most computationally intensive step in the compaction loop. It is performed using **Synopsys VC-Z01X**, a parallel fault simulator that injects each stuck-at fault one at a time into the gate-level netlist and re-simulates the firmware to determine whether the fault effect propagates

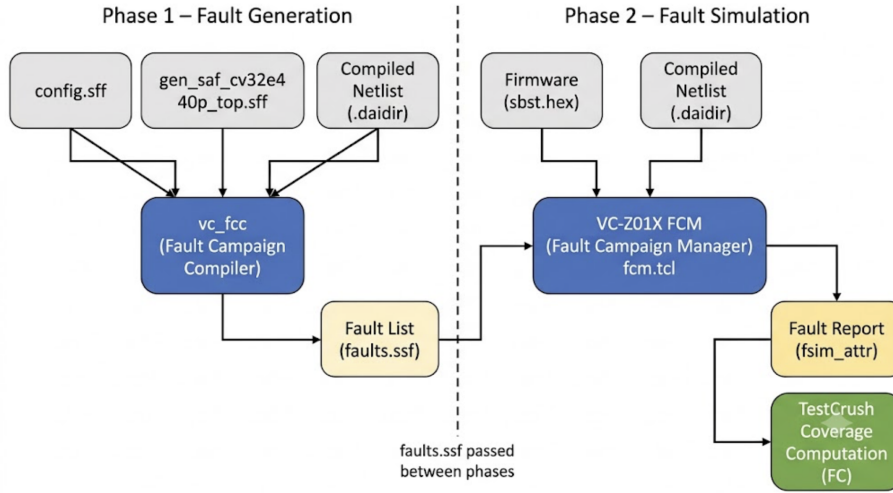


Figure 4.2: Two-phase VC-Z01X fault simulation flow. Phase 1 (`vcs/fgen/saf`) uses `vc_fcc` to compile the fault list from the `.sff` specification files and the compiled netlist, producing `faults.ssf`. Phase 2 (`vcs/fsim/gate/shell`) runs the fault campaign using the VC-Z01X FCM with the firmware and the fault list as inputs, writing per-fault status results to `fsim_attr`. TestCrush then parses `fsim_attr` to compute FC.

to an observable output. VC-Z01X is invoked in two sequential phases, both orchestrated by `make` targets. Figure 4.2 shows the overall flow.

**Phase 1 — Fault list generation (`vcs/fgen/saf`).** The first phase compiles the fault list from the fault specification files using the VC-Z01X Fault Campaign Compiler (`vc_fcc`):

```
vc_fcc -full64 -daidir simv.daidir -sff
      config.sff,gen_saf_cv32e40p_top.sff \
-report cv32e40p_top_saf.rpt -campaign cv32e40p
      -overwrite -collapse off
```

The `.daidir` argument points to the compiled gate-level database produced by VCS, and the two `.sff` files supply the status definitions and the fault scope (Section 4.1). The compiler resolves which physical ports exist in the netlist, generates SA0 and SA1 entries for each one, and writes them to a fault report file. The `-collapse off` flag disables fault collapsing, meaning logically equivalent faults are not merged; every individual port fault is listed and simulated separately. The resulting report is also copied to `faults.ssf`, which VC-Z01X uses as its starting campaign in Phase 2.

**Phase 2 — Fault simulation campaign (vcs/fsim/gate/shell).** The second phase runs the actual fault simulation campaign using the VC-Z01X Fault Campaign Manager (FCM), driven by a TCL script (`fcm.tcl`):

```
create_testcases -name {"test1"} -exec simv \  
    -args "./simv +firmware=sbst/sbst.hex" \  
    -fsim_args "-fsim=fault+dictionary"  
fsim -verbose  
report -campaign cv32e40p -report fsim_attr \  
    -overwrite -showallattributes
```

The `create_testcases` command registers the same VCS simulation binary used in logic simulation, but now VC-Z01X takes control of the execution: for each fault in the campaign, it injects the fault into the gate-level netlist, launches the simulation with the STL firmware, and observes whether the fault effect propagates. The `-fsim=fault+dictionary` flag enables both fault simulation and diagnostic dictionary generation. After the campaign completes, `report` writes the per-fault status results to the file `fsim_attr` with the `-showallattributes` flag, which ensures that timing information and diagnostic attributes are included for every fault entry. This file is what TestCrush parses to compute FC.

**Timeout and error filtering.** Each fault simulation invocation is guarded by a per-run wall-clock timeout. The timeout values used in this thesis — 400 seconds for A1xx and A2xx runs and 900 seconds for A0 runs — are not universal constants; they were chosen empirically based on the typical fault simulation duration observed on the specific server used for the experiments. On a faster or slower machine, or with a different netlist size, the appropriate values would differ.

The choice follows a practical principle: the timeout should be set generously above the expected worst-case simulation time, so that legitimate (but slow) runs are not killed, while still being low enough to catch pathological cases — such as a candidate removal that corrupts the program counter and causes an infinite loop — without stalling the compaction run indefinitely. A wall-clock timeout is used rather than a CPU-time limit because VC-Z01X spawns multiple processes and its total CPU consumption is harder to bound reliably; wall-clock time provides a simpler and more robust termination guarantee.

VC-Z01X emits the informational message `Info: Connected to started server` on stderr at startup; this message is explicitly whitelisted in the TestCrush configuration so that it does not trigger a false simulation error. Any other stderr output causes the trial to be treated as a fault simulation failure and the candidate is rejected.

**Cost relative to logic simulation.** Fault simulation is significantly more expensive than logic simulation: a single logic simulation run takes a few seconds, while a full fault simulation campaign on the same firmware can take several minutes, because the hardware is re-simulated once per fault. This asymmetry motivates the *fault-filtering* to reduce number of simulated faults and *early-exit* optimization discussed in Chapter 3: if logic simulation already shows that a candidate worsens TaT, fault simulation is skipped entirely for that trial, avoiding the dominant cost of the compaction loop.

### 4.3 TestCrush Framework

TestCrush is an open-source Python framework whose role is to automate *simulation-driven compaction* of assembly STLs. Rather than attempting to prove redundancy analytically, TestCrush follows an engineering workflow that repeatedly proposes candidate removals, rebuilds the STL, and validates each candidate using the same logic and fault simulation infrastructure already adopted in the project. In this sense, TestCrush acts as an *orchestration layer*: it drives external tools and collects the metrics needed to decide whether a modification should be kept or rolled back.

The scope of TestCrush is intentionally bounded. It does not replace industrial simulators, nor does it synthesize new STL content from scratch; instead, it assumes that a verification environment exists (typically Synopsys VCS for logic simulation and VC-Z01X for fault simulation) and that the user can express the required build and simulation steps as shell commands. TestCrush also does not redefine fault models or coverage semantics: FC is computed from the Z01X fault report using a user-selected coverage formula, while TaT is extracted from the logic simulation output.

Figure 4.3 shows the high-level architecture: TestCrush reads and edits the STL assembly sources; cross-compilation turns them into firmware; VCS logic simulation and VC-Z01X fault simulation then supply the TaT trace and fault report that TestCrush uses to accept or reject each candidate edit.

Figure 4.4 refines this into the *iterative* compaction loop executed for every candidate change. The following subsection walks that loop in the order cross-compile → logic simulation → fault simulation → policy decision.

#### 4.3.1 End-to-end simulation-based compaction workflow

TestCrush implements STL compaction as an iterative, simulation-in-the-loop process. The starting point is a *golden* STL whose assembly sources are provided as input together with a list identifying which lines correspond to removable instructions. The workflow then follows a fixed validation chain for every candidate modification.

First, the STL is *cross-compiled*, producing the firmware image used in simulation. If compilation fails, the candidate change cannot be a valid compaction step; TestCrush therefore rejects the modification and restores the STL to its previous state. If compilation succeeds, a *logic simulation* step is executed under VCS. This step has two roles: it checks that the modified program still terminates correctly under the testbench assumptions, and it extracts the TaT from the simulator output. Logic simulation therefore acts as the functional gatekeeper for candidate removals.

Next, *fault simulation* is executed under VC-Z01X. This step evaluates the fault-detection quality of the modified STL by producing an updated fault simulation report from which FC is computed for a user-selected coverage formula. Fault simulation is typically the computationally dominant part of the loop, and it provides the metric that must remain under control during compaction.

Finally, TestCrush compares the newly measured (TaT, FC) pair against the reference statistics according to the configured acceptance policy. The modification is accepted if both conditions are met; otherwise it is rolled back by restoring the removed instruction(s). The process repeats until the candidate space is exhausted or no further acceptable removals are found, producing a compacted STL together with logged statistics of the attempted trials.

#### 4.3.2 Instruction-level compaction (A0)

Algorithm A0 compacts an STL at the granularity of *single assembly instructions*. During initialization, TestCrush parses each configured STL source file using the ISA keyword list: only lines classified as valid instructions become removable candidates, while other assembly text is not directly targeted for removal. All candidates are collected into an ordered list  $\mathcal{C} = \{(i, \ell)\}$ , where  $i$  identifies the source file and  $\ell$  denotes a specific removable instruction.

Before the main compaction loop begins, the ordered list of candidate instructions is shuffled. The shuffle is applied 100 times (`times_to_shuffle` = 100), each time re-randomising the traversal order. This prevents the compaction result from being biased by the original source-file ordering and gives the search a better chance of finding removable instructions that would otherwise be skipped early due to sequential dependencies. The value of 100 is the default used across all algorithms.

Before the iterative search begins, A0 performs a *golden* evaluation on the unmodified STL: cross-compilation, VCS logic simulation to obtain the baseline TaT, and VC-Z01X fault simulation to compute the baseline FC.

---

**Algorithm 2** A0 instruction-level compaction (Maximize)

---

**Require:** STL assembly sources  $\mathcal{S}$ , ISA keyword list, build/sim commands

**Ensure:** Compacted STL (in-place edits) + logged trials

```
1:  $(TaT_{\text{ref}}, FC_{\text{ref}}) \leftarrow \text{GOLDENRUN}(\mathcal{S}) \triangleright \text{cross-compile} \rightarrow \text{lsim} \rightarrow \text{fsim} \rightarrow$   
   parse FC  
2:  $\mathcal{C} \leftarrow \text{COLLECTINSTRUCTIONCANDIDATES}(\mathcal{S})$   
3: optionally shuffle  $\mathcal{C}$  multiple times  
4: while  $\mathcal{C} \neq \emptyset$  do  
5:    $(i, \ell) \leftarrow \text{POPHEAD}(\mathcal{C})$   
6:   remove  $\ell$  from source  $i$   
7:   if not  $\text{CROSSCOMPILE}()$  then  
8:      $\text{RESTORE}(i)$ ; continue  
9:   end if  
10:  if not  $\text{LOGICSIM}()$  then  $\triangleright$  success + TaT extraction  
11:     $\text{RESTORE}(i)$ ; continue  
12:  end if  
13:   $TaT^* \leftarrow \text{MEASUREDTAT}()$   
14:  if not  $\text{FAULTSIM}()$  then  
15:     $\text{RESTORE}(i)$ ; continue  
16:  end if  
17:   $FC^* \leftarrow \text{COMPUTEFCFROMREPORT}()$   
18:  if  $TaT^* \leq TaT_{\text{ref}}$  and  $FC^* \geq FC_{\text{ref}}$  then  
19:     $(TaT_{\text{ref}}, FC_{\text{ref}}) \leftarrow (TaT^*, FC^*)$   
20:  else  
21:     $\text{RESTORE}(i)$   
22:  end if  
23: end while
```

---

The resulting pair  $(TaT_0, FC_0)$  initializes the reference statistics used to judge subsequent trials.

The main compaction loop processes candidates until the list is exhausted. At each iteration, A0 selects the next candidate instruction, removes it in-place from the corresponding assembly source, and validates the modification through cross-compilation, logic simulation, and fault simulation. If any step fails, the trial is rejected and the instruction is restored. When both simulations succeed, A0 measures the new TaT and recomputes FC from the updated fault report. Trials are accepted or rejected according to the *Maximize* policy: accept if and only if  $TaT_{\text{new}} \leq TaT_{\text{ref}}$  and  $FC_{\text{new}} \geq FC_{\text{ref}}$ . Algorithm 2 gives the pseudocode.

### 4.3.3 Adjacent block compaction (A1xx)

Algorithm A1xx extends A0 by changing the *candidate granularity*: instead of removing one instruction at a time, A1xx removes a *block* of adjacent instructions of fixed size  $k$  from each assembly file. The resulting ordered list of blocks  $\mathcal{B} = \{(i, B)\}$ , where  $i$  identifies the assembly file and  $B = [\ell_1, \dots, \ell_m]$  is a block of adjacent removable instructions, constitutes the search space.

The main loop iterates over blocks in *reverse* order (bottom to top). For each block, A1xx removes all  $k$  instructions and validates the modified STL through the same simulation pipeline as A0. If compilation or either simulation fails, the block is restored. When both simulations succeed, A1xx applies the *Maximize* acceptance rule; if the block passes, the reference statistics are updated. If the full block fails, a fallback restores instructions one at a time under the configured removal policy (F: forward, B: backward, R: random) and re-evaluates the shrinking sub-block after each restoration.

The notation **A1XY** denotes A1xx with segment size  $k = X$  and removal policy  $Y \in \{F, B, R\}$ . The forward policy and a segment size of 3 were found in prior work [2] to offer the best balance between compaction quality and simulation cost, and are used as the primary baseline configuration (**A1F3**) in this thesis. Details of the logging and backup mechanisms are described in Section 4.4.

## 4.4 TestCrush Configuration

### 4.4.1 TOML configuration file

Every TestCrush run is driven entirely by a single TOML configuration file. This file collects all project-specific settings paths, build commands, simulation commands, algorithm parameters, and preprocessing options in one place, so the core compaction algorithms require no modification when moving between projects or machines. The file is divided into named sections, each controlling a different aspect of the compaction flow: `[user_defines]` for path variables, `[a1xx_behaviour]` (or `[a2xx_behaviour]`) for algorithm parameters, `[isa]` for the ISA keyword file, `[assembly_sources]` for the STL files under test, `[cross_compilation]`, `[vcs_logic_simulation]`, `[zoix_fault_simulation]` for the three build and simulation steps, `[fault_report]` for the coverage formula, and `[preprocessing]` for the trace-based candidate filtering.

To avoid hard-coding absolute paths throughout the file, TestCrush supports a placeholder substitution mechanism: any value defined under `[user_defines]` can be referenced anywhere in the file by wrapping its key in percent signs.

For example, defining `root_dir = "../cv32e40p"` in `[user_defines]` and then writing `%root_dir%/run/vc-z01x/fsim_attr` in the `[fault_report]` section causes TestCrush to substitute the full path before use. This keeps the configuration concise and makes it trivial to relocate the project directory without editing every individual path.

#### 4.4.2 ISA specification

TestCrush must distinguish which lines of an assembly source file are executable instructions (candidates for removal) and which are non-instruction text (labels, assembler directives, blank lines, comments) that must be preserved. This distinction is made using an *ISA keyword file*: a plain-text file listing every valid instruction mnemonic, one per line, with comment lines starting with `#`.

For this thesis, the ISA file is `riscv.isa`, which defines 171 mnemonics organised into the following groups:

- **Pseudo-instructions**: assembler-level aliases such as `li`, `la`, `mv`, `ret`, `nop`, `call`, and the branch pseudo-ops (`beqz`, `bnez`, `bgt`, `ble`, etc.).
- **RV32I** (base integer): all standard 32-bit RISC-V instructions including arithmetic, logical, shift, load, store, and branch instructions.
- **RV32M** (multiplication/division): `mul`, `mulh`, `mulhsu`, `mulhu`, `div`, `divu`, `rem`, `remu`.
- **RV32C** (compressed): the 16-bit compressed encodings of common instructions.
- **Zicsr / Zifencei** extensions: control-status register instructions and `fence.i`.
- **Floating-point extensions** (RV32F, RV32FC, RV32D, RV32DC): single and double-precision instructions included for completeness, though the STLs benchmarked here do not use them.

At startup, TestCrush reads this file, builds a mnemonic set, and then scans each assembly source line by line. A line is classified as a removable instruction candidate if and only if its first token matches a mnemonic in this set; all other lines are left untouched throughout the compaction process.

#### 4.4.3 Algorithm parameters

Each algorithm is configured through its own behaviour section in the TOML file. The parameters used in the experiments of this thesis are as follows.

**Segment dimension and removal policy (A1xx and A2xx).** Both A1xx and A2xx use a segment dimension of  $k = 3$ , meaning each block processed in the compaction loop contains three instructions. The removal policy for both is F (forward), meaning that within each block the instructions are removed starting from the last one in the block and working toward the first. As discussed in Chapter 2, the forward policy and segment size of 3 were found to offer the best balance between compaction quality and simulation cost. These settings are therefore held constant across all A1xx and A2xx experiments to isolate the effect of the other variables being compared.

**Acceptance policy and coverage tolerance.** The compaction policy controls which (TaT, FC) outcomes are accepted. Two policies are used in the experiments (see Figure 3.3 in Section 3.2.3 for the corresponding accept/reject regions):

- **Maximize:** accepts a candidate removal only if the new TaT is no worse than the reference TaT *and* the new FC is no worse than the reference FC. Both metrics must hold simultaneously; no coverage degradation is permitted.
- **Range:** accepts a candidate removal if the new TaT is no worse than the reference TaT *and* the new FC does not fall below a global floor defined as  $FC_{\text{initial}} - \Delta$ , where  $\Delta$  is the `allowed_drop_coverage_prt` parameter. In the experiments,  $\Delta = 0.01$ , corresponding to a maximum permitted absolute drop of 1 percentage point from the initial baseline coverage. This tolerance allows the compaction to make incremental progress even when a candidate removal causes a marginal coverage decrease, as long as the cumulative drop never exceeds 1 pp.

The Range policy with  $\Delta = 0.01$  is used for A1xx and A2xx experiments. Maximize is used as the strict baseline for comparison. For A0, different Threshold policies are used, for example for  $\Delta = 0.03$  coverage may drop by up to 3 pp from the initial value; A0 is applied only to evaluate single-instruction granularity baselines where a looser tolerance is needed to observe non-trivial compaction.

**Fault simulation refresh interval.** In the A1xx configuration, the parameter `full_fsim_refresh_interval = 20` instructs TestCrush to run a complete, unfiltered fault simulation every 20 accepted removals, even if fault filtering is active. This periodic refresh recomputes the baseline FC from scratch to prevent the reference statistics from drifting due to accumulated approximations from fault filtering. Setting this to 0 would disable the refresh; the value of 20 was chosen as a practical balance between accuracy and simulation cost.

#### 4.4.4 Preprocessing configuration

The preprocessing step, described algorithmically in Chapter 3, is enabled for all experiments in this thesis (`enabled = true`). Preprocessing identifies instructions occurring within fault-activation windows and excludes them from the removable candidate set because they are considered likely contributors to fault detection. The configuration parameters that control it are:

- **processor\_name** = `'CV32E40P'`: selects the trace parser. TestCrush uses a Lark grammar specific to the CV32E40P trace format (`trace_cv32e40p.lark`) to parse the processor execution trace into a structured SQLite database.
- **processor\_trace**: path to the execution trace file (`trace.log`) produced by running the golden STL under VCS. This trace records, for every executed instruction, the simulation time, the program counter, the decoded instruction, and the register/memory state. It is generated once before preprocessing begins and is not regenerated during the compaction loop.
- **elf\_file**: path to the ELF binary (`sbst.elf`) produced by cross-compilation. TestCrush uses this file together with the `addr2line` utility (via the `pyelftools` library) to map program-counter addresses found in the trace back to specific source file lines. This mapping is what allows the preprocessor to associate a fault's activation window with a concrete assembly line number in the STL sources.
- **zoix\_to\_trace** = `{ 'PC_ID': 'PC', 'sim_time': 'Time' }`: a dictionary that maps VC-Z01X fault attribute names to the corresponding column names in the trace database. VC-Z01X records each fault's activation time as `sim_time` and the program counter at activation as `PC_ID`; these are mapped to the `Time` and `PC` columns of the trace table so the preprocessor can query which instructions were executing during the fault's activation window.

Beyond pruning the candidate list, the preprocessing step also produces a mapping from assembly source locations to the detected faults whose activation window includes each line, saved as `mapping_reverse.json`. This mapping is the input consumed by the optional fault filtering mechanism (described in Chapter 3 and Section 4.7.2), which uses it at runtime to identify which faults can be safely excluded from each per-trial fault simulation campaign.

#### 4.4.5 Coverage formula and fault report

The fault report parsed by TestCrush after each fault simulation run is the `fsim_attr` file written by VC-Z01X (Section 4.2). The coverage formula

applied to this report is set to `'Observational Coverage'`, which corresponds to the formula defined in Equation (4.1) and described in Section 4.1. TestCrush reads the formula string directly from the `Coverage` section of the fault report and evaluates it by tallying the fault counts per status group, ensuring that the FC value computed during compaction matches exactly what would be reported by a standalone VC-Z01X run.

#### 4.4.6 Traceability: logging and backup

Before any in-place editing of the assembly sources begins, TestCrush archives a complete copy of the original STL files as a timestamped zip archive. This ensures that the unmodified baseline can always be recovered if needed, regardless of how many iterations the compaction loop executes.

During the compaction loop, every trial — whether accepted or rejected — is recorded in a timestamped CSV log file. Each row captures: the assembly source file, the instruction or block removed, whether compilation succeeded, whether logic simulation succeeded, the measured TaT, whether fault simulation succeeded, the computed FC, and the final verdict (accepted or rejected). The filename encodes the algorithm name, policy, and segment dimension, making it straightforward to identify which run produced which log when multiple experiments are run in sequence. These logs are the primary data source for the experimental results presented in Chapter 5.

### 4.5 Benchmark STLs

The six STLs used in this thesis were collected from a real verification environment targeting the CV32E40P processor. They were not designed with compaction in mind; they are production-quality test programs written using a variety of generation strategies, as reported in Table 4.3. This makes them realistic benchmarks: the compaction algorithms must work with whatever redundancy happens to be present in each program, rather than with synthetic inputs tuned to demonstrate a particular outcome.

The STLs are divided between the two target modules introduced in Section 4.1. STL1, STL2, and STL6 target the multiplier unit (MULT); STL3, STL4, and STL5 target the load/store unit (LSU). Each STL is evaluated against the fault list derived from its respective module’s SFF scope file, so the fault coverage figures reported in Table 4.2 are module-specific and cannot be compared across the two groups directly.

Table 4.2: Baseline characteristics of the benchmark STLs.

| STL | Target module | #Instr | Duration (CCs) | FC (%) |
|-----|---------------|--------|----------------|--------|
| 1   | MULT          | 638    | 45,487         | 91.14  |
| 2   | MULT          | 646    | 11,460         | 95.09  |
| 3   | LSU           | 760    | 36,177         | 71.51  |
| 4   | LSU           | 110    | 1,650          | 69.89  |
| 5   | LSU           | 132    | 2,333          | 78.13  |
| 6   | MULT          | 48     | 3,131          | 54.78  |

Table 4.3: Generation methods of the benchmark STLs.

| STL  | Generation method                    |
|------|--------------------------------------|
| STL1 | ATPG                                 |
| STL2 | Random vectors                       |
| STL3 | Manual, with some ATPG patterns      |
| STL4 | Manual, with some ATPG patterns      |
| STL5 | Random vectors + corner cases        |
| STL6 | Random vectors + complementary pairs |

#### 4.5.1 MULT STLs

STL1 and STL2 are the two primary MULT test programs, and despite having nearly identical instruction counts (638 and 646 respectively) they differ substantially in runtime and coverage. STL1 was generated by ATPG: the tool automatically synthesizes instruction sequences that activate and propagate specific stuck-at faults, targeting the multiplier logic directly. The resulting program runs for 45,487 clock cycles and achieves 91.14% FC. STL2, by contrast, was built from randomly generated input vectors. Although the random approach lacks the fault-directed precision of ATPG, it achieves higher coverage (95.09%) in considerably less time (11,460 CCs), suggesting that the MULT unit’s fault space is broad enough that random stimuli cover it effectively without the overhead of fault-directed computation. The roughly  $4\times$  difference in runtime between STL1 and STL2 — despite similar instruction counts — also implies that ATPG-generated sequences tend to include instructions with longer execution latencies, such as multi-cycle multiply operations with data-dependent timing.

STL6 is a much more compact program: 48 instructions and 3,131 clock cycles, generated using random vectors augmented with complementary pairs. A complementary pair consists of a test instruction followed by the same instruction applied to the bitwise complement of its inputs; this strategy ensures that both SA0 and SA1 faults on each signal are activated within the

same pair. Despite this targeted coverage strategy, STL6 reaches only 54.78% FC, the lowest in the benchmark set. Its small size limits the diversity of fault activation scenarios, leaving a substantial fraction of the MULT fault list undetected. STL6 is included precisely because it represents a challenging case for compaction: a small program with modest coverage, where any instruction removed risks a meaningful FC drop.

#### 4.5.2 LSU STLs

The three LSU programs span a wider range of sizes and strategies. STL3 is the largest program in the benchmark set, with 760 instructions and 36,177 clock cycles. It was written manually by a domain expert with some ATPG patterns incorporated to fill coverage gaps, and achieves 71.51% FC. The manual origin of STL3 is consistent with its relatively large size: hand-written tests tend to be less compact than automatically generated ones, as authors often include exploratory or defensive sequences that a formal tool would prune.

STL4 and STL5 are both compact programs in the same general style — manual authorship with targeted additions — but they differ in their complementary approach. STL4, at 110 instructions and 1,650 clock cycles, is the shortest test in the entire benchmark set. It achieves 69.89% FC, a respectable result for a program of this size, and its very low TaT makes each compaction trial cheap to simulate. STL5 adds randomly generated corner-case patterns on top of a manually written base, resulting in 132 instructions and 2,333 CCs. Despite being only 20% longer than STL4, STL5 achieves noticeably higher coverage (78.13%), suggesting that the corner-case additions activate fault scenarios that the manually written core of STL4 misses.

#### 4.5.3 Diversity across the benchmark set

Taken together, the six STLs provide a benchmark set with substantial diversity across all three dimensions relevant to compaction:

- **Program size:** from 48 instructions (STL6) to 760 (STL3), a  $16\times$  range. Larger programs offer more candidate instructions and thus more compaction potential, but also impose longer simulation times per trial.
- **Test application time:** from 1,650 clock cycles (STL4) to 45,487 (STL1), a  $28\times$  range. TaT is directly coupled to per-trial simulation cost, so the benchmark set exercises the compaction loop under both fast and slow simulation regimes.
- **Fault coverage:** from 54.78% (STL6) to 95.09% (STL2), a spread of over 40 percentage points. High-coverage programs leave little room

for FC degradation, making the acceptance policy stricter in practice; low-coverage programs offer more tolerance but also suggest that some faults may be structurally difficult to detect regardless of test length.

This diversity is intentional: it allows the experimental results in Chapter 5 to assess whether compaction performance is consistent across different program characteristics, or whether algorithm effectiveness depends on properties such as program size, baseline coverage, or the generation strategy used to create the original STL.

## 4.6 Evaluation Methodology

### 4.6.1 Evaluation metrics

Three quantitative metrics are used to characterise the outcome of each compaction run. They are computed by comparing the state of the STL at the end of the compaction loop against its state at the start.

**Test application time reduction.** The primary metric is the reduction in TaT, expressed as a percentage of the original:

$$\text{TaT reduction} = \frac{\text{TaT}_{\text{initial}} - \text{TaT}_{\text{final}}}{\text{TaT}_{\text{initial}}} \times 100\% \quad (4.2)$$

TaT reduction is the direct measure of what compaction is trying to achieve: a shorter test means the processor spends less time in self-test mode and is available for its operational task sooner. By construction of the acceptance condition (new TaT must not exceed the reference TaT), the compaction loop can only accept removals that keep TaT the same or reduce it, so TaT reduction is always non-negative. A higher value is better.

**Change in fault coverage.** The secondary metric is the change in FC between the original and compacted STL, reported in percentage points (pp):

$$\Delta\text{FC} = \text{FC}_{\text{final}} - \text{FC}_{\text{initial}} \quad (4.3)$$

A value of  $\Delta\text{FC} = 0$  means the compacted STL detects exactly the same set of faults as the original; a negative value means some faults were lost. The acceptance policies described in Section 4.4.3 bound how negative  $\Delta\text{FC}$  can become: the Maximize policy enforces  $\Delta\text{FC} \geq 0$  at every step, while the Range policy allows  $\Delta\text{FC}$  to drift down to at most  $-\Delta = -1$  pp below the initial value over the entire compaction run. Together with TaT reduction,  $\Delta\text{FC}$  defines the cost-benefit tradeoff of a compaction result: a good result achieves high TaT reduction at a small (ideally zero) FC cost.

**Instruction count reduction.** A structural metric complementary to TaT reduction is the fraction of instructions removed from the STL source:

$$\text{Instr reduction} = \frac{N_{\text{initial}} - N_{\text{final}}}{N_{\text{initial}}} \times 100\% \quad (4.4)$$

Instruction count reduction is not identical to TaT reduction because instructions have different execution costs: removing a multi-cycle instruction (such as a 32-bit multiply) reduces TaT more than removing a single-cycle compressed instruction. This metric is reported alongside TaT reduction to characterise the structural degree of compaction independently of instruction timing, and to help identify cases where many cheap instructions are removed versus few expensive ones.

#### 4.6.2 Experimental protocol

Each experiment applies one compaction algorithm to one STL and records the resulting compacted program. The protocol is the same for every combination:

1. The original, unmodified STL assembly source is used as the starting point. The initial TaT and FC values — the baseline from which all relative metrics are computed — are those reported in Table 4.2, obtained by running a full logic and fault simulation on the unmodified STL before the compaction loop begins.
2. Preprocessing is run once before the compaction loop begins (Section 4.4.4). The initial fault simulation results are cross-referenced with the execution trace: for each detected fault, a window of instructions executing around the fault’s activation point is identified by querying the trace database using the fault’s recorded simulation time and program counter. The instructions in these windows are then *removed from the candidate list*, protecting them from the compaction loop because they are known to contribute to fault detection. Only the remaining instructions — those that do not appear in any fault activation window — are passed to the algorithm as removable candidates. This step runs once per STL and its output is reused for all algorithms applied to the same STL.
3. The selected compaction algorithm is then applied with the parameters described in Section 4.4.3. The algorithm terminates when it has exhausted all candidates without finding any further acceptable removals. The compacted STL produced at termination is the final output.
4. The final TaT and FC are those recorded in the last accepted trial within the compaction loop, as logged in the CSV trace file (Section 4.4).

These values are verified by a standalone logic and fault simulation run on the compacted STL after the loop completes.

### 4.6.3 Algorithm comparison strategy

The experimental comparison is structured around two axes: algorithm granularity and acceptance policy. Table 4.4 summarises the five configurations evaluated in this thesis.

Table 4.4: Algorithm configurations evaluated in the experiments.

| Configuration | Algorithm | Policy    | Tolerance ( $\Delta$ ) | Block size          |
|---------------|-----------|-----------|------------------------|---------------------|
| A0            | A0        | Threshold | 3 pp                   | 1 instruction       |
| A1xx-Max      | A1xx      | Maximize  | 0 pp                   | 3 instr. (adjacent) |
| A1xx-Rng      | A1xx      | Range     | 1 pp                   | 3 instr. (adjacent) |
| A2xx-Max      | A2xx      | Maximize  | 0 pp                   | 3 instr. (random)   |
| A2xx-Rng      | A2xx      | Range     | 1 pp                   | 3 instr. (random)   |

**Granularity: A0 as baseline.** A0, which removes one instruction at a time using the Threshold policy ( $\Delta = 3\%$ ), is used as the reference baseline in all comparisons. It represents the simplest compaction strategy and sets a lower bound on the compaction quality achievable when no block structure is exploited. Any multi-instruction algorithm (A1xx or A2xx) that outperforms A0 in TaT reduction at equal or lower FC cost demonstrates that block-level compaction adds value beyond what single-instruction removal can achieve.

**Policy: Maximize vs Range.** For both A1xx and A2xx, experiments are run with two acceptance policies: Maximize (no FC degradation permitted at any step) and Range ( $\Delta\text{FC} \geq -1$  pp over the entire run). Comparing the two policies on the same STL isolates the effect of coverage tolerance: any additional TaT reduction achieved by Range relative to Maximize comes from accepting removals that would have been rejected by the stricter policy. The FC cost of this additional compaction is quantified by  $\Delta\text{FC}$ .

**Block structure: A1xx vs A2xx.** A1xx processes candidates in a fixed sequential order (with shuffling applied to the segment list as a whole), while A2xx draws random non-adjacent blocks. Comparing the two on the same STL and policy isolates the effect of block selection strategy: do random non-adjacent blocks find more compaction opportunities than adjacent sequential ones, or does the adjacency structure of A1xx lead to more coherent removals that pass the acceptance test more reliably?

**Reading results.** For a given STL, a result from algorithm *A* is said to *dominate* algorithm *B* if *A* achieves at least as much TaT reduction and at least as high a final FC. When results are not dominated — for example, algorithm *A* reduces TaT more but *B* preserves FC better — the preferred choice depends on the application’s tolerance for coverage degradation. The results in Chapter 5 are reported in full (both TaT reduction and  $\Delta\text{FC}$ ) for each combination, so that the reader can apply whichever preference is appropriate for their use case.

Compaction quality alone, however, is not sufficient to fully characterise an algorithm. The computational cost of reaching the compacted STL — measured as the number of logic and fault simulation calls executed during the run — is an equally important dimension of comparison. An algorithm that achieves the same TaT reduction and FC as a competitor but requires fewer fault simulation campaigns is strictly preferable, since fault simulation is the bottleneck step in the compaction loop. The cost metrics defined in Section 4.7 are therefore reported alongside the quality metrics in Chapter 5, and comparisons between algorithms account for both dimensions simultaneously.

## 4.7 Computational Cost

### 4.7.1 Computational cost metrics

The computational cost of a compaction run is dominated by the number of simulations it invokes. Each compaction trial goes through up to three simulation steps in sequence: cross-compilation, logic simulation, and fault simulation. Cross-compilation is fast (a few seconds at most) and contributes negligibly to total runtime. Logic simulation is significantly more expensive: it runs the full gate-level processor model with the modified STL firmware and typically takes tens of seconds per trial. Fault simulation is the most expensive step by a wide margin: it runs one simulation per fault in the target module, and for a module with hundreds of faults, a single fault simulation campaign can take several minutes.

The cost metric used in this thesis is **total CPU time**, measured for all algorithm configurations using a `cpu+` timer that wraps the entire compaction run in `main.py`. This mode accumulates the CPU time of the Python orchestration process together with the CPU time of all child processes — VCS and VC-Z01X — via `RUSAGE_CHILDREN`, capturing the true computational load of the run rather than just Python overhead or wall-clock elapsed time. The total CPU time is printed to the console at the end of each run. For the A1xx configuration, CPU time is also recorded at a finer granularity: an additional `cpu+` timer inside the compaction loop measures and logs the CPU time of each individual candidate block trial, making it possible to see

the per-trial cost distribution in addition to the total.

### 4.7.2 Cost-reduction mechanisms

Two mechanisms reduce the number of full fault simulation calls executed during a compaction run, and their effect is directly observable in the CSV logs.

**Early-exit optimisation.** After each logic simulation, the measured TaT is compared against the current reference TaT before fault simulation is launched. If the new TaT is strictly greater than the reference — meaning the removal already worsened the test — the trial is immediately rejected and fault simulation is skipped entirely. This early-exit check (visible as `fsim_ok = SKIP` entries in the CSV log) is particularly effective on STLs where removing instructions frequently increases TaT, such as when removing a short-circuit instruction causes the processor to execute a longer alternative path. The fraction of trials that hit the early-exit condition is reported in Chapter 5 for each configuration, as it indicates how often the `lsim-only` check was sufficient to make a reject decision without the cost of a full fault simulation.

**Fault filtering.** TestCrush supports an optional fault filtering mechanism, controlled by the `fault_filtering` flag in the configuration file. When enabled, fault simulation still runs on every trial but operates on a reduced fault set rather than the full fault universe. Before launching fault simulation for a given candidate block, the algorithm identifies faults that are exclusively associated with instructions appearing earlier in the STL than the current candidate, as well as faults from other assembly files. These faults are written into an `Exclude` clause in the SAF specification file; VC-Z01X then skips them during the campaign, reducing per-trial simulation workload in proportion to how many faults can be safely excluded.

Because the excluded faults are absent from the `fsim` report, their contribution to coverage is recovered using their *baseline* status counts from the most recent full fault simulation. The full OC formula is then applied to the union of the current `fsim` results (included faults) and the baseline statuses (excluded faults). The rationale is that faults exclusively activated by earlier instructions cannot be affected by the removal of the current candidate, making their baseline status a valid substitute. To prevent this baseline from drifting as the STL shrinks, a complete unfiltered fault simulation is triggered unconditionally every `full_fsim_refresh_interval` accepted removals (set to 20 in the A1xx configurations used here). The full design and motivation are discussed in Chapter 3.

### 4.7.3 Reporting format

For each algorithm-STL configuration, Chapter 5 reports the following cost quantities alongside the compaction quality metrics:

- **Total trials:** the total number of candidate removal attempts made by the algorithm before termination, derived from the row count of the CSV log.
- **Lsim calls:** the number of logic simulations executed, counted as trials where compilation succeeded (i.e., `lsim_ok` is not empty in the CSV).
- **Fsim calls:** the number of fault simulation campaigns executed, counted as trials where `fsim_ok` is neither `SKIP` nor empty. When fault filtering is enabled, each `fsim` call operates on a reduced fault set rather than the full fault universe.
- **Fsim skip rate:** the fraction of trials where fault simulation was avoided entirely via early-exit (`fsim_ok = SKIP`). A higher skip rate indicates that the early-exit optimisation was effective for that STL.
- **Total CPU time:** the total `cpu+` time of the compaction run as printed to the run log by the timer in `main.py`, covering Python orchestration and all child simulation processes. For A1xx, per-trial CPU time is also available from the internal timer and cited where relevant.

The first four quantities are derived from the per-trial CSV log and are comparable across all configurations. CPU time is recorded from the run log and, since all experiments were executed on the same server, is directly comparable across configurations within this thesis.

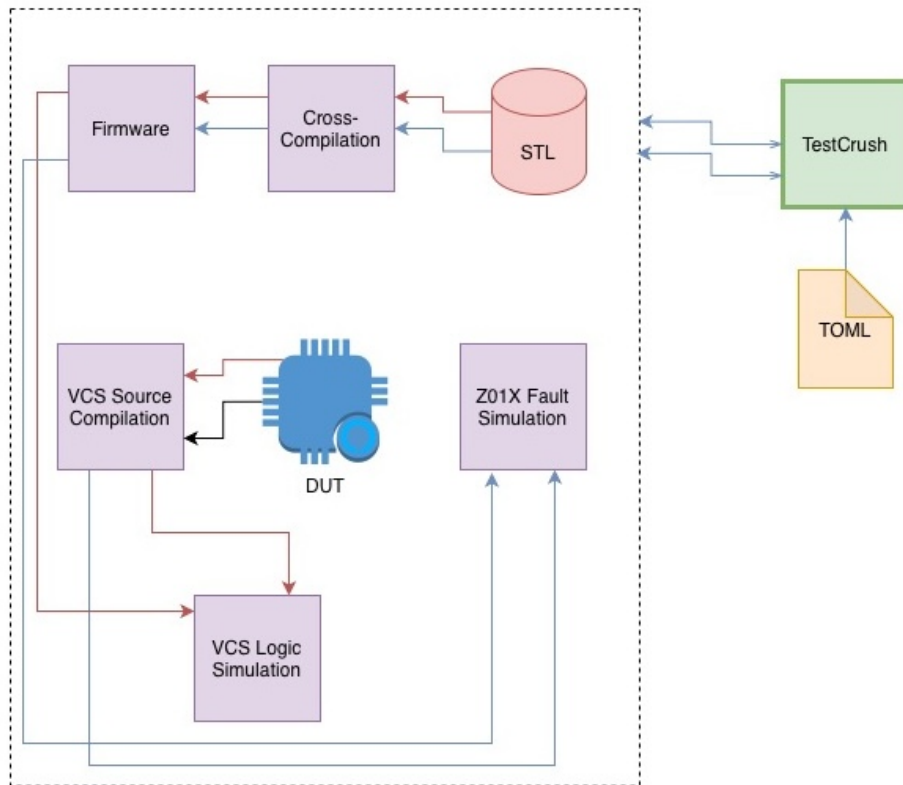


Figure 4.3: High-level architecture of TestCrush with TOML configuration, STL sources, cross-compilation to firmware, DUT compilation in VCS, and coupled logic (VCS) and fault (VC-Z01X) simulation (adapted from the TestCrush project materials [2]).

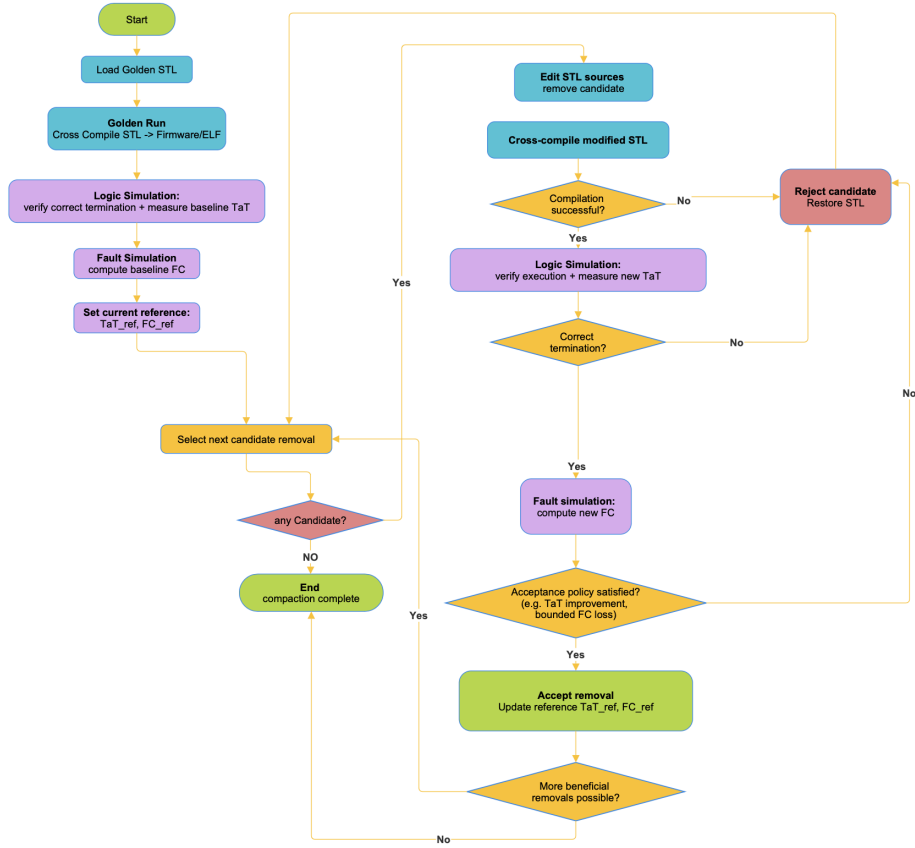


Figure 4.4: Iterative compaction loop in TestCrush: cross-compilation to firmware, VCS logic simulation (TaT and termination), VC-Z01X fault simulation (fault report and FC), and policy-based accept/reject—consistent with the stack view in Fig. 4.3.

## Chapter 5

# Experimental Results

This chapter reports the results of the experiments described in Chapter 4. The presentation follows a stepwise structure.

Section 5.1 studies the impact of trace-based preprocessing, starting with the selection of the activation-window size and then evaluating the effect of preprocessing across all six benchmark STLs. Section 5.2 compares the Maximize and R1 acceptance policies applied to A1F3 with P4 preprocessing, evaluating the trade-off between TAT reduction and fault coverage. Section 5.3 evaluates the A2xx random non-adjacent block algorithm under the same default settings (PP + R1), comparing block sizes F2 and F3 and then comparing A2xx against A1xx. Section 5.4 isolates the effect of fault filtering by comparing PP+R1 without fault filtering against the same configuration with fault filtering enabled.

### 5.1 Impact of Preprocessing

Trace-based preprocessing (Section 3.1) reduces the candidate search space by retaining only the instructions that fall within the activation window of at least one fault. Before comparing compacted results with and without preprocessing, we first determine a suitable activation-window size, since this parameter controls how aggressively the candidate set is pruned.

#### 5.1.1 Activation-Window Size Selection

To select the activation-window size, we ran A1F3 with preprocessing on STL1 and STL2 for window sizes ranging from 2 to 6. Table 5.1 reports CPU time, TAT reduction, and FC difference for each configuration.

CPU time decreases monotonically as the window grows. Preprocessing identifies the instructions that execute just before each fault activates

Table 5.1: Impact of the activation-window size on A1F3 compaction (STL1 and STL2).

| Window | STL1      |              |               | STL2      |              |               |
|--------|-----------|--------------|---------------|-----------|--------------|---------------|
|        | CPU (min) | TAT Red. (%) | FC Diff. (pp) | CPU (min) | TAT Red. (%) | FC Diff. (pp) |
| 2      | 325       | 83.71        | +5.98         | 400       | 8.8          | 0             |
| 3      | 321       | 77.72        | +4.86         | 396       | 8.8          | 0             |
| 4      | 318       | 79.09        | +4.65         | 392       | 21.0         | 0             |
| 5      | 314       | 81.91        | +5.17         | 389       | 17.0         | 0             |
| 6      | 310       | 81.91        | +5.17         | 385       | 8.8          | 0             |

and excludes them from the compaction candidate set, because removing them would directly destroy fault coverage. A larger window classifies more instructions as fault-activating and therefore excludes more from the candidate set; with fewer candidate blocks to evaluate per run, total CPU time falls.

The TAT reduction does not follow a monotonic trend. For STL1, TAT reduction is high across all window sizes (77–84%), indicating that preprocessing works well for this STL regardless of how aggressively candidates are pruned. For STL2, the picture is more sensitive: at windows 2 and 3, too few instructions are excluded, leaving many candidates that appear removable but violate coverage when actually removed; the algorithm wastes trials on these blocks and achieves only 8.8% TAT reduction. At window 4, enough additional instructions are excluded that this problem disappears, and TAT reduction rises to 21% (the best result for STL2 across all tested windows). At windows 5 and 6, the exclusion goes too far: instructions that genuinely participate in productive removal combinations are also excluded, and TAT reduction for STL2 falls again to 17% and 8.8%. FC difference is negligible for STL2 across all window sizes and remains in a similar range for STL1.

Based on these observations, a window size of 4 (hereafter  $P_4$ ) is adopted for all subsequent experiments. Window 4 achieves the best TAT result for STL2 while remaining competitive for STL1, and its CPU time is lower than windows 2 and 3 without the quality loss introduced by windows 5 and 6.

### 5.1.2 Preprocessing Effectiveness across All STLs

Table 5.2 compares A1F3 without preprocessing against A1F3 with  $P_4$  across all six benchmark STLs. The three reported metrics are CPU time, TAT reduction, and FC difference relative to the initial STL.

**CPU time.** Preprocessing reduces CPU time for five of the six STLs. The reduction is most pronounced for the smaller STLs: STL5 and STL6 each lose roughly 43% of their runtime (80→46 min and 35→20 min respectively), while STL4 drops by 25% (72→54 min). The two largest STLs, STL2 and STL3,

Table 5.2: Impact of P4 preprocessing on A1F3 compaction across all STLs.

| STL | Without Preprocessing |              |               | With P4   |              |               |
|-----|-----------------------|--------------|---------------|-----------|--------------|---------------|
|     | CPU (min)             | TAT Red. (%) | FC Diff. (pp) | CPU (min) | TAT Red. (%) | FC Diff. (pp) |
| 1   | 274                   | 58.00        | +5.25         | 318       | 79.09        | +4.65         |
| 2   | 556                   | 67.73        | +0.61         | 542       | 44.82        | +0.34         |
| 3   | 630                   | 40.16        | 0             | 616       | 40.00        | 0             |
| 4   | 72                    | 33.70        | +4.62         | 54        | 26.97        | +11.26        |
| 5   | 80                    | 20.53        | +0.53         | 46        | 19.46        | +0.54         |
| 6   | 35                    | 8.85         | 0             | 20        | 21.11        | 0             |

show only modest reductions (2–3%), suggesting that when the STL is large and fault activation is broadly distributed, preprocessing prunes a smaller fraction of the total candidate set. STL1 is the only outlier: preprocessing increases CPU time by 44 min (+16%). This occurs because preprocessing changes which instructions are kept as candidates; in STL1, the pruned search space happens to follow a more difficult compaction trajectory that requires more trials to converge, even though it ultimately achieves better TAT reduction.

**TAT reduction.** The effect of preprocessing on compaction quality is mixed. STL1 improves substantially: TAT reduction increases from 58.0% to 75.7% (+17.7 pp), a result that also explains the higher CPU cost, since the algorithm explores more aggressively before converging. STL6 shows a similarly large improvement (8.85%→21.11%, +12.3 pp), demonstrating that preprocessing can guide the search toward removal combinations that the unguided run misses entirely. Conversely, STL2 degrades noticeably (67.73%→44.82%, –22.9 pp) and STL4 degrades moderately (33.70%→26.97%, –6.7 pp). In these cases, preprocessing prunes candidate blocks that the unguided algorithm would have removed, restricting the search to a subset that leads to a lower TAT reduction. STL3 and STL5 are essentially neutral ( $\Delta \leq 1$  pp), indicating that for these STLs the pruned and unpruned candidate sets lead to comparable compaction outcomes.

**FC difference.** FC difference remains small in all configurations, both with and without preprocessing. All non-zero entries are positive, indicating that the compacted STL slightly improves fault coverage relative to the original in those cases—a side effect of removing instructions that introduce signal-observation noise rather than contributing positively to fault detection. The largest change is for STL4 with P4, where FC difference reaches +11.26 pp; this remains acceptable within the Maximize policy, which requires that coverage does not decrease.

**Summary.** Preprocessing with P4 is generally effective at reducing compaction cost, particularly for smaller STLs where the search space reduction

is proportionally large. Its effect on compaction quality, however, depends on the STL structure: for some STLs (STL1, STL6) it improves TAT reduction by guiding the search toward better candidates, while for others (STL2, STL4) it restricts the search in a way that limits the achievable reduction. FC impact is negligible in all cases. Given the mixed but overall positive profile—cost reduction in five out of six STLs, quality improvement in two, no FC degradation—P4 preprocessing is retained as the default configuration in all remaining experiments.

## 5.2 Impact of the Range Policy

Having established P4 as the default preprocessing configuration, we now evaluate the effect of replacing the Maximize (M) acceptance policy with the Range policy (Section 3.2.3). Under Maximize, a candidate block is accepted only if the resulting fault coverage does not decrease at all. Under the Range variant R1, a decrease of up to 1 pp in fault coverage is tolerated, in exchange for potentially larger TAT reductions. Both configurations use A1F3 with P4 preprocessing. Table 5.3 reports CPU time, TAT reduction, and FC difference for each STL under both policies.

Table 5.3: Impact of the Maximize (M) and R1 acceptance policies on A1F3 + P4 compaction.

| STL | M (Maximize) |              |               | R1 (Range, $\Delta FC \leq 1$ pp) |              |               |
|-----|--------------|--------------|---------------|-----------------------------------|--------------|---------------|
|     | CPU (min)    | TAT Red. (%) | FC Diff. (pp) | CPU (min)                         | TAT Red. (%) | FC Diff. (pp) |
| 1   | 318          | 79.09        | +4.65         | 281                               | 90.22        | +1.98         |
| 2   | 392          | 44.82        | +0.34         | 549                               | 75.27        | −1.00         |
| 3   | 612          | 40.00        | 0             | 392                               | 91.67        | −0.46         |
| 4   | 54           | 26.97        | +11.26        | 34                                | 38.30        | +9.69         |
| 5   | 47           | 19.46        | +0.54         | 40                                | 24.70        | +0.25         |
| 6   | 18           | 21.11        | 0             | 19                                | 21.11        | 0             |

**TAT reduction.** R1 improves TAT reduction for every STL. The gains are particularly large for STL2 and STL3: TAT reduction rises from 44.82% to 75.27% (+30.5 pp) and from 40.0% to 91.67% (+51.7 pp) respectively. These are STLs where Maximize was blocking a large number of candidate blocks due to small coverage fluctuations that fell just below zero; relaxing the acceptance bound to −1 pp unlocks many of those blocks and allows the algorithm to make substantial progress. STL1 and STL4 also benefit noticeably (+11.1 pp and +11.3 pp), while STL5 improves moderately (+5.2 pp). STL6 is unchanged, suggesting that the blocks removable under Maximize are already sufficient to exhaust the compaction potential for that STL regardless of the acceptance policy.

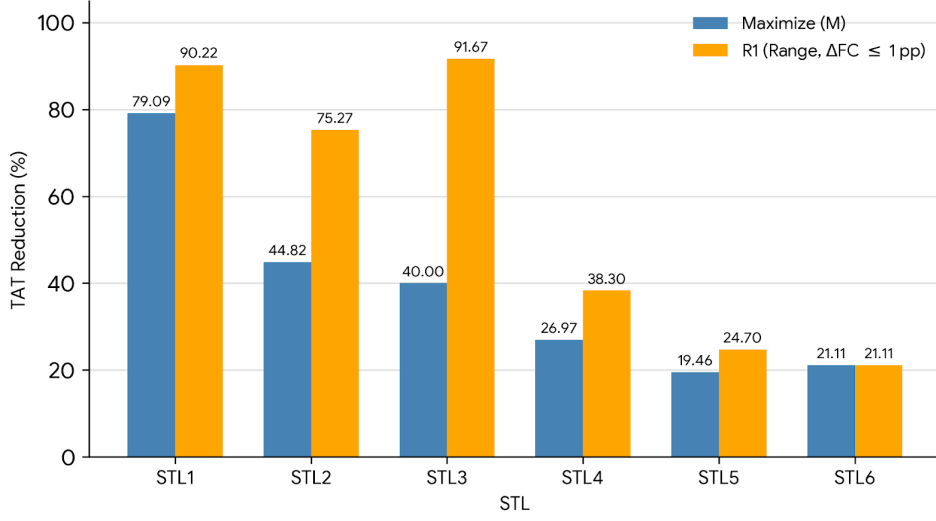


Figure 5.1: TAT reduction achieved by the Maximize (M) and R1 acceptance policies across all STLs (A1F3 + P4).

**CPU time.** The CPU effect of R1 is mixed. For four STLs (1, 4, 5, and 3), R1 reduces CPU time: the most significant saving is for STL3, which drops from 612 to 392 min ( $-36\%$ ), because a higher acceptance rate means the algorithm reaches convergence faster, after which fewer expensive fault simulation calls are needed to issue a restore verdict on the remaining candidates. STL6 is essentially unchanged ( $+1$  min). STL2 is the exception: CPU increases from 392 to 549 min ( $+40\%$ ), because R1 accepts more blocks and therefore performs more successful removals per run, each requiring a fault simulation call; the algorithm does more productive work, which costs more time, but it achieves a substantially better TAT reduction in return.

**FC difference.** FC difference remains within the  $-1$  pp bound enforced by R1 in all cases. STL2 reaches exactly the bound ( $-1.00$  pp), and STL3 stays well within it ( $-0.46$  pp). For the remaining STLs, FC difference is positive or zero, meaning that even with the relaxed policy the compacted STL does not lose coverage relative to the original.

**Summary.** R1 consistently improves TAT reduction compared to Maximize, with gains ranging from 5 pp to 52 pp across the six STLs. The cost in fault coverage is small and always within the 1 pp tolerance: in four out of six cases FC remains non-negative, and in the two cases where it drops the decrease is at most 1 pp. CPU time is reduced or unchanged for five of the six STLs; the one increase (STL2) is a direct consequence of performing

more accepted removals, not of wasted effort. These results indicate that R1 is a strictly more effective policy than Maximize when a 1 pp coverage tolerance is acceptable.

### 5.3 A2xx Algorithm

Having established P4 preprocessing and the R1 policy as the best-performing configuration for A1xx in Sections 5.1 and 5.2, we now evaluate the A2xx algorithm under the same default settings. A2xx (Section 3.3) selects candidate blocks of randomly chosen non-adjacent instructions rather than fixed adjacent groups. Two block sizes are evaluated: F2 (two randomly selected instructions per block) and F3 (three). Section 5.3.1 compares the two block sizes within A2xx, and Section 5.3.2 compares A2xx against A1xx.

#### 5.3.1 Block Size Comparison within A2xx

Table 5.4 compares A2xx under PP + R1 for block sizes F2 and F3 across all STLs.

Table 5.4: A2xx with P4 preprocessing and R1 policy: block size F2 vs F3.

| STL  | PP_R1 F2     |               |           | PP_R1 F3     |               |           |
|------|--------------|---------------|-----------|--------------|---------------|-----------|
|      | TAT Red. (%) | FC Diff. (pp) | CPU (min) | TAT Red. (%) | FC Diff. (pp) | CPU (min) |
| STL1 | 90.00        | −1.00         | 434       | 95.00        | +0.40         | 259       |
| STL3 | 96.00        | +0.35         | 380       | 94.00        | −0.70         | 259       |
| STL4 | 41.00        | +9.20         | 34        | 11.51        | +11.00        | 52        |
| STL6 | 0.05         | −0.03         | 12        | 0.05         | −0.03         | 9         |
| STL5 | 25.00        | +0.07         | 27        | 24.00        | +0.40         | 23        |
| STL2 | 62.22        | −0.58         | 286       | 49.00        | −0.81         | 290       |

**TAT reduction.** F2 and F3 produce similar results for STL1, STL3, STL5, and STL6. The most significant divergence occurs for STL4: F2 achieves 41% TAT reduction while F3 achieves only 11.51% (−29.5 pp). For STL1, F3 is slightly better (95% vs 90%, +5 pp), and for STL2, F2 achieves notably higher TAT reduction (62.22% vs 49%, +13.2 pp).

**CPU time.** F3 is consistently faster: it saves 175 min for STL1 (−40%), 121 min for STL3 (−32%), and 4 min for STL5. The exceptions are STL4, where F3 is slower (+18 min), consistent with its much lower TAT reduction, and STL2, where the two block sizes are essentially equal (286 vs 290 min).

**FC difference.** All FC differences remain within the −1 pp bound of R1 for both block sizes. STL1 under F2 reaches exactly −1 pp; all other entries are positive or negligibly negative.

**Summary.** Neither block size uniformly dominates the other. F2 is clearly better for STL4, while F3 is faster and slightly better for STL1. For the remaining STLs, the two sizes are comparable. Both block sizes are therefore included in the comparison against A1xx.

### 5.3.2 A2xx vs A1xx

Table 5.5 directly compares the best A1xx configuration (PP + R1 + F3, from Section 5.2) against both A2xx configurations (PP + R1, F2 and F3) across the shared STL set.

Table 5.5: A1xx PP+R1 vs A2xx PP+R1 (F2 and F3) under the default configuration.

| STL  | A1xx PP_R1 F3 |               |           | A2xx PP_R1 F2 |               |           | A2xx PP_R1 F3 |               |           |
|------|---------------|---------------|-----------|---------------|---------------|-----------|---------------|---------------|-----------|
|      | TAT Red. (%)  | FC Diff. (pp) | CPU (min) | TAT Red. (%)  | FC Diff. (pp) | CPU (min) | TAT Red. (%)  | FC Diff. (pp) | CPU (min) |
| STL1 | 90.22         | +1.98         | 281       | 90.00         | -1.00         | 434       | 95.00         | +0.40         | 259       |
| STL2 | 75.27         | -1.00         | 549       | 62.22         | -0.58         | 286       | 49.00         | -0.81         | 290       |
| STL3 | 91.67         | -0.46         | 392       | 96.00         | +0.35         | 380       | 94.00         | -0.70         | 259       |
| STL4 | 38.30         | +9.69         | 34        | 41.00         | +9.20         | 34        | 11.51         | +11.00        | 52        |
| STL5 | 24.70         | +0.25         | 40        | 25.00         | +0.07         | 27        | 24.00         | +0.40         | 23        |
| STL6 | 21.11         | 0             | 19        | 0.05          | -0.03         | 12        | 0.05          | -0.03         | 9         |

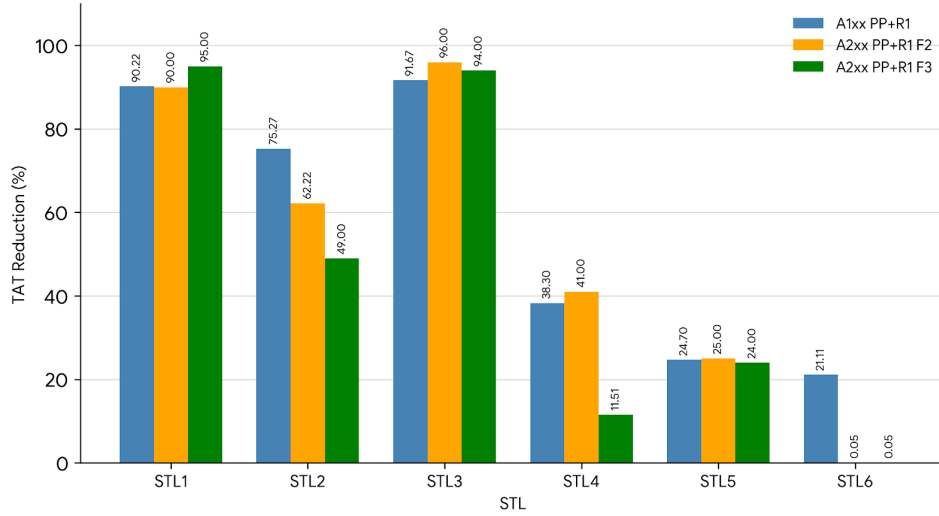


Figure 5.2: TAT reduction achieved by A1xx PP+R1, A2xx PP+R1 F2, and A2xx PP+R1 F3 across all STLs.

**TAT reduction.** A2xx matches or exceeds A1xx for STL1 (F3: 95% vs 90.22%), STL3 (F2: 96% vs 91.67%), STL4 (F2: 41% vs 38.30%), and STL5 (F2: 25% vs 24.70%). However, A1xx is substantially better for STL2 (75.27% vs 62.22% for F2 and 49% for F3) and STL6 (21.11% vs near-zero for

both A2xx block sizes). The failure of A2xx on STL6 is particularly striking: neither F2 nor F3 achieves meaningful compaction (0.05%), whereas A1xx achieves 21.11% TAT reduction on the same STL.

**CPU time.** A2xx F3 is faster than A1xx for STL1 (259 vs 281 min), STL3 (259 vs 392 min), STL5 (23 vs 40 min), and STL6 (trivially, since it does nothing). A1xx is faster for STL2 (549 min for A1xx vs 286–290 min for A2xx; note A1xx spends more time because it achieves better TAT). STL4 is comparable across all three configurations (34–52 min).

**FC difference.** All configurations remain within acceptable bounds. STL1 under A2xx F2 reaches  $-1$  pp (the R1 limit); all other entries are within  $\pm 1$  pp and mostly positive.

**Summary.** A2xx (random non-adjacent blocks) does not consistently outperform A1xx (adjacent blocks) under the same default configuration. For some STLs (STL3, STL4), A2xx F2 achieves slightly better TAT reduction, and A2xx F3 is faster for STL1 and STL3. However, A1xx is substantially better for STL2 and STL6, suggesting that adjacent block removal is more effective when the redundant instructions in an STL are spatially clustered. A2xx may offer an advantage in STLs where redundant instructions are scattered across the program, but the current benchmark set does not provide a consistent advantage for random blocking over adjacent blocking. It should be noted that, because A2xx selects candidate blocks randomly, the results reported here reflect single runs and are subject to run-to-run variability; a statistically more reliable comparison would require multiple independent runs with averaged outcomes. The same caveat applies to the block size comparison in Section 5.3.1. This is discussed further as a limitation and direction for future work in Chapter 6.

## 5.4 Impact of Fault Filtering

Fault filtering (FF, Section 3.5) reduces per-trial fault simulation cost by telling the fault simulator to skip faults that cannot be activated by the current candidate block. Specifically, before each fault simulation call, the algorithm identifies two categories of faults to exclude: (1) faults mapped exclusively to instructions that appear before the candidate block in the same STL file, and (2) faults that have any mapping to a different STL file. Only faults that were *detected* in the baseline are ever excluded; undetected faults are always kept in simulation to prevent masking real coverage loss. The excluded faults are written as an `Exclude` block in the fault simulator input, so the simulation runs only on the remaining fault subset — directly reducing the cost of each fsim call.

Because the excluded faults are not simulated, their coverage contribution must be reconstructed analytically. Their statuses are taken from the most recent full-simulation baseline snapshot, and the adjusted observational coverage is computed by combining the current (filtered) simulation counts with the baseline counts of the excluded faults (Section 3.5.4). To keep this snapshot aligned with the evolving STL, a periodic full unfiltered fault simulation is triggered every  $R$  accepted calls to rebuild the baseline; between refreshes, the excluded-fault statuses are frozen at their last recorded values.

To isolate the effect of fault filtering, we compare the established default configuration (PP + R1, results from Section 5.2) against the same configuration with fault filtering enabled (FF + PP + R1). The early-exit threshold is set conservatively so that early exit does not trigger, ensuring the observed differences are attributable solely to fault filtering. Table 5.6 and Figure 5.3 report the results.

Table 5.6: Impact of fault filtering: PP+R1 (without FF) vs FF+PP+R1.

| STL  | PP + R1 (without FF) |               |           | FF + PP + R1 |               |           |
|------|----------------------|---------------|-----------|--------------|---------------|-----------|
|      | TAT Red. (%)         | FC Diff. (pp) | CPU (min) | TAT Red. (%) | FC Diff. (pp) | CPU (min) |
| STL1 | 90.22                | -1.00         | 281       | 90.22        | +2.00         | 370       |
| STL2 | 75.27                | -1.00         | 549       | 59.58        | -0.66         | 148       |
| STL3 | 91.67                | -0.46         | 392       | 84.36        | -0.18         | 337       |
| STL4 | 38.30                | +9.69         | 34        | 44.78        | +0.85         | 49        |
| STL5 | 24.70                | +0.25         | 40        | 24.73        | +0.25         | 23        |
| STL6 | 21.11                | 0             | 19        | 21.11        | 0             | 17        |

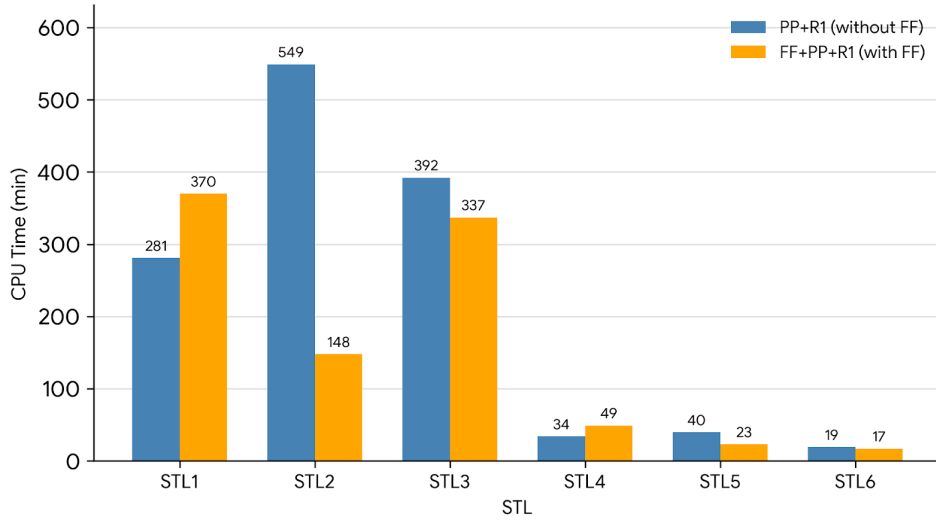


Figure 5.3: CPU time comparison between PP+R1 (without FF) and FF+PP+R1 across all STLs.

**CPU time.** Fault filtering reduces CPU time for four of the six STLs. The reduction comes directly from running each fsm on a smaller fault set: fewer faults to simulate means each call finishes faster. The largest saving is for STL2 (549  $\rightarrow$  148 min,  $-73\%$ ), which has 16 242 faults and therefore benefits the most from reducing the active fault set per trial. STL5 saves  $-43\%$ , STL3 saves  $-14\%$ , and STL6 saves  $-11\%$ . STL1 and STL4 are exceptions: CPU time increases by +89 min ( $+32\%$ ) and +15 min ( $+44\%$ ). In these cases, the adjusted coverage computed from the filtered simulation leads to different acceptance decisions than the full simulation, sending the compaction algorithm down a trajectory that requires more trials to converge — outweighing the per-trial speed gain.

**TAT reduction.** For three STLs (STL1, STL5, STL6), TAT reduction is unchanged ( $\leq 0.03$  pp difference), confirming that fault filtering has no effect on compaction quality when the excluded faults do not influence the acceptance decisions. STL4 improves by +6.5 pp (38.30%  $\rightarrow$  44.78%): excluding faults that are unrelated to the candidate block removes noise from the coverage estimate, allowing the algorithm to correctly accept blocks that would have been unnecessarily rejected under the full fault set. STL2 and STL3 degrade by  $-15.7$  pp and  $-7.3$  pp respectively. In these STLs, the adjusted coverage — which combines the filtered simulation result with frozen baseline statuses of excluded faults — diverges from the true coverage as the STL evolves between baseline refreshes. This divergence causes some acceptance decisions to be wrong: blocks that genuinely maintain coverage are rejected, or blocks that slightly hurt coverage are accepted, steering the compaction toward a worse final result.

**FC difference.** FC difference improves with fault filtering for most STLs: STL1 moves from  $-1.00$  pp to  $+2.00$  pp, STL2 from  $-1.00$  pp to  $-0.66$  pp, and STL3 from  $-0.46$  pp to  $-0.18$  pp. This occurs because fault filtering makes each per-trial coverage estimate more focused — only faults relevant to the candidate block are simulated — so the accepted removals tend to have a smaller negative impact on the true final coverage. STL4 is an exception: its FC difference falls from  $+9.69$  pp to  $+0.85$  pp because fault filtering directs the algorithm toward a different set of accepted removals than without filtering, producing a compacted STL with less coverage headroom.

**Heuristic nature and refresh trade-off.** Fault filtering is a heuristic: the adjusted coverage is an approximation, not the true coverage of the full fault set. Its accuracy depends on how fresh the baseline snapshot is. A smaller refresh interval  $R$  (refresh more often) keeps the excluded-fault statuses close to their current values, producing more accurate coverage estimates and more reliable acceptance decisions, at the cost of additional

full fault simulation calls. A larger  $R$  (refresh less often) reduces the number of expensive refresh calls and saves more total CPU time, but allows the baseline to drift as the STL is progressively modified, increasing the risk that the adjusted coverage diverges from the true coverage and acceptance decisions become incorrect. The degradation observed for STL2 and STL3 is consistent with this drift effect: the baseline snapshot becomes stale between refreshes, causing the algorithm to misjudge coverage and follow a suboptimal compaction path.

**Summary.** Fault filtering primarily targets CPU time reduction by shrinking the fault set processed in each trial. It is most effective for STLs with large fault counts where the per-trial simulation cost is the dominant bottleneck (e.g. STL2,  $-73\%$ ). Its effect on TAT reduction is STL-dependent: three STLs are unaffected, one improves, and two degrade due to baseline drift. Practitioners can tune the refresh interval  $R$  to balance coverage accuracy against runtime saving: a smaller  $R$  reduces drift at higher cost, while a larger  $R$  maximises savings at the risk of trajectory degradation.

## Chapter 6

# Conclusions

### 6.1 Summary

This thesis addressed the problem of compacting assembly-level Software Test Libraries (STLs) for safety-critical processors: reducing test application time (TaT) while keeping fault coverage (FC) within acceptable bounds, in a workflow that integrates with an industrial VC-Z01X fault simulation environment. Five contributions were developed and evaluated on STLs targeting the CV32E40P RISC-V processor under the stuck-at fault model.

**Trace-based preprocessing.** The first contribution is a trace-based preprocessing step that reduces the compaction search space before the main algorithm starts. By querying the gate-level simulation trace, the method identifies which instructions fall within the activation window of each fault and excludes them from the candidate set, since removing them would directly destroy fault coverage. Experimentally (Section 5.1), preprocessing with a window size of 4 (P4) reduces CPU time for the majority of STLs, with savings reaching 43% for smaller STLs. Its effect on TAT reduction is STL-dependent: for some STLs it guides the algorithm toward better compaction outcomes, while for others the pruned candidate set restricts achievable TAT reduction.

**Range acceptance policy.** The second contribution is the Range acceptance policy, which relaxes the strict Maximize criterion by tolerating a bounded decrease in fault coverage (up to a configurable limit  $\Delta FC$ ) in exchange for larger TAT reductions. Under the R1 variant ( $\Delta FC \leq 1$  pp), the policy consistently improves TAT reduction compared to Maximize across all six benchmark STLs (Section 5.2), with gains ranging from 5 pp to 52 pp. Fault coverage always remains within the specified tolerance, and CPU time is reduced or unchanged for five out of six STLs. Range is the most uniformly effective contribution introduced in this thesis.

**A2xx random non-adjacent block algorithm.** The third contribution is the A2xx algorithm, which generalises the block-level compaction strategy by selecting candidate blocks of randomly chosen non-adjacent instructions rather than fixed adjacent groups. The evaluation (Section 5.3) shows that A2xx with the default configuration (PP + R1) does not consistently outperform A1xx: it achieves slightly better TAT reduction for some STLs (STL3, STL4) but falls significantly behind for others (STL2, STL6). Since A2xx relies on random selection, the results of single runs are subject to run-to-run variability, and a more rigorous evaluation would require averaging over multiple independent runs.

**Fault filtering.** The fourth contribution is the fault filtering mechanism, which reduces the per-trial fault simulation cost by excluding faults that cannot be activated by the current candidate block from each fsm call. Excluded faults’ coverage contribution is reconstructed analytically by combining the current simulation result with their statuses from the most recent baseline snapshot, and the baseline is periodically refreshed via a full unfiltered simulation to limit drift. Experimentally (Section 5.4), fault filtering saves up to 73% CPU time for STL2 (the largest STL by fault count), confirming that the technique is most effective when per-trial simulation cost is the dominant bottleneck. For three STLs the TAT outcome is unchanged, for one it improves, and for two it degrades due to baseline drift between refreshes. The refresh interval  $R$  provides a controllable trade-off between coverage accuracy and CPU saving.

**Early exit.** The fifth contribution introduces two complementary early-exit mechanisms that reduce wasted computation during compaction. The first is a TaT-conditioned fsm skip: since logic simulation is cheaper than fault simulation, TaT is checked first, and if it already worsens beyond an acceptable threshold the fault simulation step is skipped entirely and the candidate is immediately rejected. This avoids launching expensive fsm calls on candidates that are already known to be unacceptable on TaT grounds alone. The second mechanism is a consecutive-restore threshold: if a configurable number of candidate blocks in a row all receive a restore verdict (i.e., are rejected), the compaction run terminates early, on the premise that the remaining candidates in that region of the STL are unlikely to yield productive removals. Together, the two mechanisms target different sources of wasted effort: the first eliminates unnecessary fsm calls within trials, while the second avoids exploring unproductive regions of the candidate space.

**Overall assessment.** Taken together, the contributions demonstrate that the TestCrush compaction flow can be made more effective and more efficient through targeted extensions. The Range policy is the most impactful quality

improvement, and fault filtering offers the largest potential CPU saving for fault-intensive STLs. Preprocessing provides a useful cost reduction in most scenarios at little quality risk. A2xx broadens the search strategy but does not show a clear advantage over adjacent blocking in the current benchmark set. All techniques remain within the VC-Z01X simulation workflow and introduce no changes to the underlying fault model or STL structure.

## 6.2 Limitations

The results and conclusions of this thesis must be interpreted in light of several methodological and scope constraints.

**Single processor and fault model.** All experiments target the CV32E40P RISC-V processor under the stuck-at fault model. Whether the observed behaviour — including the STL-dependent effectiveness of preprocessing, fault filtering, and A2xx — generalises to processors with different microarchitectures or to other fault models (e.g. transition faults, cell-aware faults) is unknown and cannot be inferred from the current data.

**Limited benchmark set.** The evaluation uses six STLs targeting a single processor. Six instances provide useful initial evidence but are insufficient to draw statistically robust conclusions about the relative merit of compaction strategies across the space of possible STL structures, sizes, and initial coverage profiles.

**Single-run A2xx results.** Because A2xx selects candidate blocks randomly, the outcome of a single run is not representative of the expected value. The A2xx results in Section 5.3 are each based on one run per configuration, which means the comparison against A1xx — and the block-size comparison within A2xx — cannot be treated as statistically reliable. No ranking between A2xx and A1xx can be firmly established from the current data.

**Fault filtering coverage accuracy.** The fault filtering mechanism introduces two sources of coverage estimation error. The first is baseline drift: excluded faults’ statuses are taken from the most recent baseline snapshot rather than the current simulation, and this approximation degrades between refreshes. The second is a structural assumption: the mechanism excludes faults whose activation points lie exclusively in earlier instructions or in unrelated source files, on the premise that the current candidate block cannot affect them. This assumption does not hold in all cases — for instance, when the STL contains branches, modifying a later instruction can alter control flow and thereby change which earlier faults are activated or observed. Strictly excluding all such faults can therefore underestimate coverage changes and

introduce inaccuracy independent of baseline refresh rate. Together, these two sources mean that the adjusted coverage estimate used to accept or reject a candidate can diverge from the true value.

## 6.3 Future Work

Several directions emerge naturally from the limitations and open questions identified in this work.

**Statistical evaluation of A2xx.** The A2xx algorithm selects candidate blocks randomly, which means the compaction outcome varies across independent runs. The results reported in Section 5.3 are based on single runs and therefore do not provide a statistically reliable basis for comparing A2xx against A1xx. A rigorous evaluation should execute A2xx multiple times on each STL, report averaged TAT reduction, FC difference, and CPU time, and quantify the variance across runs. This would clarify whether the cases where A2xx fell behind A1xx (e.g. STL2, STL6) reflect a structural disadvantage of random blocking or simply unfavourable single-run outcomes.

**Adaptive baseline refresh for fault filtering.** The fault filtering baseline is currently refreshed at a fixed interval of  $R$  accepted fault simulation calls. A fixed  $R$  cannot adapt to the rate at which the STL changes: if many instructions are accepted in a short span, the baseline drifts rapidly and a shorter interval would be needed; if few are accepted, a longer interval wastes refresh calls. An adaptive refresh policy that monitors the divergence between adjusted and raw coverage estimates — and triggers a refresh when drift exceeds a configurable threshold — could maintain accuracy while minimising unnecessary full fault simulation calls.

**Extension to other processors and fault models.** All experiments were performed on STLs targeting the CV32E40P RISC-V processor under the stuck-at fault model. Applying the same techniques to other processor architectures (e.g. processors with deeper pipelines, out-of-order execution, or different register files) and to additional fault models (e.g. transition faults or cell-aware faults) would test the generalisability of the results and reveal whether the STL-dependent behaviour observed here is architecture-specific or a general property of the compaction problem.

**Automated parameter selection.** Several parameters — the preprocessing window size, the A2xx block size, the Range tolerance  $\Delta_{FC}$ , and the fault filtering refresh interval  $R$  — were selected manually based on a limited set of experiments. Developing a lightweight profiling step that estimates suitable parameter values for a given STL before compaction starts, or an

adaptive scheme that adjusts parameters online during compaction, could reduce the need for manual tuning and improve robustness across diverse STLs.

# References

- [1] Srivaths Bommu et al. “Static compaction using overlapped restoration and segment pruning”. In: *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*. 1998, pp. 140–146.
- [2] cad-polito-it. *TestCrush: A prototypical toolkit for Software Test Library (STL) compaction*. GitHub repository. Prototype Python toolkit implementing compaction algorithms for pre-existing assembly STLs, based on prior work on functional test program compaction. 2026. URL: <https://github.com/cad-polito-it/testcrush> (visited on 04/15/2026).
- [3] Marco Gaudesi, Matteo Sonza Reorda, and Irith Pomeranz. “On test program compaction”. In: *Proceedings of the 20th IEEE European Test Symposium*. IEEE. 2015.
- [4] Michele Gaudesi et al. “New Techniques to Reduce the Execution Time of Functional Test Programs”. In: *IEEE Transactions on Computers* 66.7 (July 2017), pp. 1268–1273. DOI: 10.1109/TC.2016.2643663.
- [5] Min-Shien Hsiao et al. “Fast algorithms for static compaction of sequential circuit test vectors”. In: *Proceedings 15th IEEE VLSI Test Symposium (VTS '97)*. 1997, pp. 188–195.
- [6] Min-Shien Hsiao and S. T. Chakradhar. “State relaxation based subsequence removal for fast static compaction in sequential circuits”. In: *Design, Automation and Test in Europe*. 1998, pp. 577–582.
- [7] Xiaoqing Lin et al. “SIFAR: Static test compaction for synchronous sequential circuits based on single fault restoration”. In: *Proceedings 18th IEEE VLSI Test Symposium*. 2000, pp. 205–212.
- [8] Anuraag Narang et al. “An Exploration of Microprocessor Self-Test Optimisation Based On Safe Faults”. In: *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2021. DOI: 10.1109/ETS54228.2021.9568326.

- [9] Stavros N. Neophytou and Maria K. Michael. “Test Set Generation With a Large Number of Unspecified Bits Using Static and Dynamic Techniques”. In: *IEEE Transactions on Computers* 59.3 (Mar. 2010), pp. 301–316.
- [10] Irith Pomeranz. “Concatenation of functional test subsequences for improved fault coverage and reduced test length”. In: *IEEE Transactions on Computers* 61.6 (June 2012), pp. 899–904.
- [11] Irith Pomeranz. “Two-dimensional static test compaction for functional test sequences”. In: *IEEE Transactions on Computers* 64.10 (Oct. 2015), pp. 3009–3015.
- [12] Irith Pomeranz and Sudhakar M. Reddy. “On static compaction of test sequences for synchronous sequential circuits”. In: *Proceedings of the 33rd Design Automation Conference*. 1996, pp. 215–220.
- [13] Irith Pomeranz and Sudhakar M. Reddy. “Vector restoration based static compaction of test sequences for synchronous sequential circuits”. In: *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 1997, pp. 360–365.
- [14] Irith Pomeranz and Sudhakar M. Reddy. “Vector replacement to improve static test compaction for synchronous sequential circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.2 (Feb. 2001), pp. 336–342.
- [15] Irith Pomeranz and Sudhakar M. Reddy. “Enumeration of test sequences in increasing chronological order to improve the levels of compaction achieved by vector omission”. In: *IEEE Transactions on Computers* 51.7 (July 2002), pp. 866–872.
- [16] Irith Pomeranz and Sudhakar M. Reddy. “Vector restoration based static compaction using random initial omission”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.11 (Nov. 2004), pp. 1587–1592.
- [17] Michalis Psarakis et al. “Microprocessor Software-Based Self-Testing”. In: *IEEE Design & Test of Computers* 27.3 (May 2010), pp. 4–19.
- [18] A. Touati et al. “An effective approach for functional test programs compaction”. In: *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE. 2016.