Mahtab: Phase-wise Acceleration of Regression Testing for C

Shouvick Mondal, Rupesh Nasre

Department of Computer Science and Engineering, Indian Institute of Technology, Madras, Chennai 600036, India

Abstract

Software regression testing consists of offline, online, and execution phases which are executed sequentially. The offline phase involves code instrumentation and test-coverage collection. Subsequently, the online phase performs program differencing, test-suite selection and prioritization. Finally, the selected test-cases are executed against the new version of software for its re-validation. Regression testing is a time-consuming process and is often on the critical path of the project. To improve the turn-around time of software development cycle, our goal is to reduce regression testing time across all phases using multi-core parallelization. This poses several challenges that stem from I/O, dependence on third-party libraries, and inherently sequential components in the overall testing process. We propose parallelization test-windows to effectively partition test-cases across threads. To measure the benefit of prioritization coupled with multi-threaded execution, we propose a new metric, EPSilon, for rewarding failure observation frequency in the timeline of test-execution. To measure the rate of code-change coverage due to regression test prioritization, we introduce ECC, a variant of the widely used APFD metric. We illustrate the effectiveness of our approach using the popular Software-artifact Infrastructure Repository (SIR) and five real-world projects from GitHub.

Keywords: Regression test selection, test-prioritization, parallelization window, relevance-and-confinedness

1. Introduction

Software testing is one of the core components in the development cycle of evolving software. Different stages of this evolution result in versions of the software, where a new version is built from its clean older version. The older version is deemed to be clean due to a successful testing performed while validating the version for its correctness, functionality, and performance. Therefore, the goal in such a setting is to test the new version for software faults. This approach is known as regression testing and devising techniques to improve the same has been an active area of research for almost three decades. Former research mentions that regression testing may eat up to 80% of total testing budget, and 50% of the software maintenance cost ([Bertolino, 2007] [Harrold, 2009]). While there are human factors involved in the process, the automated parts such as identifying regression, test selection, and execution can be accelerated with parallel execution. Parallelizing the testing process is challenging because it involves multiple expensive steps rather than a single critical step. Therefore, we need a holistic approach towards parallelizing testing. Such an approach needs to parallelize different testing phases using varied mechanisms and provide an end-to-end solution towards effective parallelization of regression test selection.

At a high level, regression testing enables running only those test-cases that are required to test the changed portion of a code. This necessitates maintaining a map between test-cases and the portions of code each test-case executes (that is, a static execution path). Such a map, called coverage map or test-case dependency graph, is created using the original code, while the change can be detected using the diff between the original and the modified codes. Based on the diff, only the relevant test-cases are identified using the coverage map. This process is called regression test selection (RTS) ([Rothermel and Harrold, 1996]). To check if any of the changed portions contains bugs, the test-cases can be executed in any arbitrary order. However, for early identification of a bug, test-cases are often prioritized. This process is called test-case prioritization, which can use varied criteria such as the number of lines covered, or number of unique branches covered, etc. Creation of the coverage map is the offline phase. RTS and prioritization constitutes the online phase. Actual execution of the test-case is called execution phase. Our goal in this work is to find effective ways to parallelize each of these three phases, which pose different challenges. We utilize parallelization windows to achieve it.

1.1. Background

We present necessary background related information used throughout the paper, in this subsection. Faults, Failures, and Errors: We adopt these terminologies from existing works ([Avizienis et al. 2004] [Perez et al. 2017]) and customize/restate as follows.

- Failure: an event that occurs when the delivered service (that is, the current output) deviates from the correct service (that is the expected/gold output)
- Error: a system state or system configuration that might cause a failure
• **Fault**: the root cause which gives rise to an error in the system-under-test

**Test oracle:** A test oracle (Howden [1978] Yu et al. [2013] Barr et al. [2015] Jahangirov [2017]) is a mechanism that acts as an indicator of whether a test-case has passed or failed, when applied on a system-under-test in a given environment.

**Fault-aware analysis:** Conventional program analyses targeting controlled experiments (Do et al. [2005]) always relied on [test-case] ↔ [fault] mapping for measuring the efficacies of existing techniques. Having complete fault-knowledge requires execution of the system-under-test twice: once for knowledge gathering, and subsequently for experimentation. However, fault-knowledge is unavailable in the real-world setting. But the availability of such test oracles is a common assumption in existing literature primarily used to gauge the effectiveness of a proposed technique. So, we too assume implicit test oracles.

**Fault-blind analysis:** Our proposed approach is fault-blind and is based entirely on failure observation with full support from test oracles. This eliminates the need for expensive gathering of fault-knowledge in the first place, and makes it efficient.

**Test-set independence:** A set of test-cases is independent if the observed outcomes of all ordering does not differ when executing all possible orderings. A test-set has dependencies (Bell et al. [2015] Candido et al. [2017] Gambi et al. [2018]) among test-cases if there exists an ordering with deviated behavior. Inter-test dependencies can be broken (Lam [2016] Kappler [2016]) to speed-up test-execution.

**Observable test timeline:** A timeline composed of a sequence of discrete time events is observable if the outcome of a test-execution event can be classified as failure or success.

1.2. Contributions

To the best of our knowledge, none of the existing works target parallelization of different stages of regression testing. The main contributions of this paper are as follows.

- We propose **Mahtab** a framework for multi-core parallelization of end-to-end regression testing which covers offline, online, and execution phases.

- We introduce a test-suite prioritization heuristic based on a combination of change-relevance and confinedness.

- We introduce a metric **ECC** (variant of APFD [Elbaum et al. 2000]) for measuring the effectiveness of change-coverage during regression test-prioritization.

- We propose acceleration of regression testing using parallelization windows (ω). We define two fault-blind metrics, **EPSilon** (EPS) and **EPSilon₂** (EPS₂), for quantifying prioritization strategies based on their test-schedule similarity with the optimal prioritization.

- We show that for SIR programs, parallel regression testing achieves an end-to-end geometric mean speedup of 4.72× compared to sequential RTS (and 2.44× against RetestAll). We achieve a geometric mean boost (EBF) of 1.6× in effectiveness (EPS) of test prioritization, using up to 16 threads. For GitHub projects used in our study, we observed end-to-end speedup of 3.90× compared to sequential RTS, and EBF of 1.43×, using up to 32 threads.

The rest of this paper is organized as follows. Section 2 presents our approach Mahtab in detail. Section 3 presents experimental evaluation of Mahtab. Section 4 compares with and contrasts against the relevant related work. Finally, Section 5 mentions our findings and states directions for future work.

2. Our Approach: Mahtab

This section describes the design of our regression testing engine Mahtab, which employs parallel regression test selection in combination with test-case prioritization. Following the literature, we work with two assumptions: (i) the program-under-test is sequential, and (ii) the dependency graph across various phases of testing is static. Both the assumptions are quite common in practice.

2.1. Running Example

We first explain the working of our test-parallelization engine using an example.

```c
// source.c
#include <stdio.h>
int g = 5;
int main()
{
    int n;
    scanf("%d", &n);
    if (n >= 0)
    {
        ...
        n = n - 64; // change
    }
    else
    {
        ...
    }
    else
    {
        ...
    }
}
```

```c
// source_v1.c
#include <stdio.h>
int g = 7; // change
int main()
{
    int n;
    scanf("%d", &n);
    if (n >= 0)
    {
        ...
        n = n - 64; // change
    }
    else
    {
        ...
    }
}
```

Figure 1: Original source code (source.c) and its new version (source_v1.c).

Figure 1 presents a base version source.c and its modification source_v1.c, due to single or multiple checkins. To avoid false positives due to minor syntactic changes (such as addition of whitespace) and to be robust in identifying dependencies, Mahtab works at the intermediate representation (IR) level. We use LLVM (Lattner and Adve [2004]) for writing our analysis passes. IR for the example programs are presented in Figure 2. The test-suite is simple: it consists of ten test-cases test1 through test10, with odd and even numbered tests containing integer values {1, 3, 5, 7, 9} and {-2, -4, -6, -8, -10} respectively.

http://pace.cse.iitm.ac.in/tools.php

In Persian, Mahtab means moonlight.
The first step in Mahtab is to create a dual-mapping from test-cases to program statements a test-case passes through. This needs program instrumentation and running the test-suite. Mahtab uses C++ file handling mechanisms for instrumentation. Figure 3 shows part of the LLVM IR for function main(). While most of the earlier works operate at the granularity of methods, Mahtab operates at the granularity of basic blocks. This makes Mahtab more precise, and test-case execution more efficient (as we illustrate in Section 5). The figure shows labels at the basic block boundaries, with predecessors inserted as a comment (comments start with semicolon). The instrumentation tracks basic blocks by inserting a call in every basic block to print the label corresponding to the basic block. This is shown in Figure 4. This way, whenever a test-case is executed, that is, when the underlying program is run with an input, the labels of only those basic blocks that get executed for the test-case are printed due to the instrumentation. By running it for all the test-cases (ten in our running example), we find a mapping of each test-case to the basic blocks executed while running the test-case. Such a coverage map (graph) is pictorially shown in Figure 5.

**Global-insensitive test selection** (globals_off). If the change in global variable (line 3) is not considered, the set of affected blocks due to IR differencing is: \{main | if.then\}. Therefore, the set of selected test-cases is \{test1, test3, test5, test7, test9\}. This means, for the change shown in Figure 1 only the values \{1, 3, 5, 7, 9\} should be provided as input, which execute the added line 11. In other words, even numbered test-cases cannot identify the faults induced by the change.

The selected test-cases are ordered (statically) based on a heuristic (Section 2.4) to uncover faults faster. The execution phase follows this prioritization order. The tests may be executed either sequentially or using parallel windows as shown in Figure 6. For our running example, parallel test-execution will achieve a theoretical speedup of at most 5 \times on using a window of size |ω| = 5. In general, with \( k \) test-cases, \( \lceil \frac{k}{5} \rceil \) windows are required for execution, the last window-size (ωlast) being \( k \mod |ω| \). This is illustrated in Figure 6 (bottom right), where \( k = 5, |ω| = 3, \) and \( |ω_{\text{last}}| = 2 \). The above processing is performed as part of the execution phase.

**Parallelization Windows.** Sequential test-execution follows the observable test timeline, where at each time instant, we observe and measure a particular event. A parallel execution window (PEW), \( ω \) of size \(|ω| = 1\), is a sequential execution window.
When $|\omega| > 1$, test-execution is multi-threaded. We assume that: (i) test-cases belonging to the same execution window are independent, and (ii) test-executions do not write/read to shared data causing races. A PEW $\omega$ is said to have failed if there exists a failed test-case $t_j \in \omega$. We generalize a window as parallel $\mu$ window $(P_{\mu}W)$, where $\mu \in \{\text{coverage-collection, map-population, award-valuation}\}$ is analogously defined with an exception that the window failure is undefined or does not occur. In general, a window of size $|\omega|$ is assigned a set of $|\omega|$ threads which are in one-to-one correspondence with the members (tasks) of that window.

Windows may be viewed as supersteps wherein the tasks across windows are executed in sequence, but the tasks within a window are executed in parallel. Therefore, there is an implicit barrier between a window $\omega_z$ and the next higher window $\omega_{z+1}$ in the window-sequence. The difference parallelization brings in is that if a failure is reported, it is reported within a window, not strictly ordered by the test-cases. Thus, it is possible that a test-case last in the window may report a failure while a previous test-case within the same window is still executing. From parallelization perspective, the positions of the two test-cases are indistinguishable – as both execute concurrently. This changes the usual sequential notions and definitions of dealing with effectiveness of a test-case prioritization mechanism, which we address in Section 2.4.3

2.2. Test Engine’s Software Architecture

Mahtab architecture is shown in Figure 7 which describes the module flow graph. The shaded modules exhibit parallelism opportunities. We explain Mahtab module-wise below, covering the three phases of regression testing: offline, online, and execution.

2.2.1. Offline phase

Offline phase works with the original version of the code along with the test-suite, and creates a map between test-cases and source elements. This stores a mapping between test-cases and source elements. Such a mapping is useful in identifying the subset of the test-cases to run when a program changes. Several proposed techniques (Zhang, 2018; Gligoric et al., 2015; Rothermel et al., 2001; Elbaum et al., 2002; Celik et al., 2017; Chen et al., 1994; Dini et al., 2016; Elbaum et al., 2000; Gligoric, 2015; Harrold et al., 2001; Jasz et al., 2012; Legunsen et al., 2017; Rothermel and Harrold, 1997; Rothermel et al., 2000) consider functions or classes as the source elements. To obtain a more precise information, Mahtab operates at the basic-block granularity (similar to a few former works (Vokolos and Frankl, 1997; 1998; Srivastava and Thiagarajan, 2002)). Such a fine-grained processing increases the size of the coverage map, but also reduces the number of test-cases to run on the new version.

We now explain various steps of the offline phase.

old_IRgen_rem.C: The old version source.c arrives at time $t_0$ and is translated into LLVM IR source.ll using the clang front-end of LLVM. Subsequently, comments appearing against basic-block labels are removed from the old version. The new version (Figure 2) will have its comments retained (appearing after basic-block names) and used in the online phase (Section 2.2.2). The absence of comments enables the block names to appear as diff between IRs of the corresponding versions. For instance, in Figure 2(line 7), the change appears as: [if.then: > if.then: ; preds ...]. The ’>’ character
Algorithm 1: Offline-phase Parallelization

**Input:** Old-version: source.c, Test-suite: Σ, Window-size: |ω|
**Output:** Test-coverage graph: map[1..|Ω|] Old-version IR: source.ll

1. source.ll := old/IRgen/source.c; Old-version IR: source.ll
2. source_new.ll := instrument_old(source.ll);
3. a.out := translateToExecutable(source_new.ll);
4. /* parallelizing test_coverage with coverage-collection windows */
5. for t_i ∈ Ω do parallel do
6. t_i.cover := execute a.out with input t_i;
7. end
8. map[1..|Ω|] := NULL;
9. /* parallelizing create_fwd_map with map-population windows */
10. for i := 1 to |Ω| in |Ω| parallel do
11. map[i] := growList (t_i.cover);
12. end

represents the direction of addition, which in this case occurs in the file corresponding to the new IR (for code removal, it would be ‘−’).

**instrument_old:** This module statically instruments the augmented source.ll at the granularity of basic-blocks. For our example, the corresponding instrumented IR appears in Figure 4.

The instrumentation point is chosen to be the first instruction of a basic block. LLVM stores the IR in partial static single assignment (SSA) form. In case of a sequence of phi instructions, instrumentation occurs past the end of the sequence. Since globals are stored in a globals block in LLVM, they need to be considered separately. For a change in the global definition, all the basic blocks that participate in the test-case are potentially affected, leading to selection of all the test-cases. In Mahtab, the corresponding global variables are added in the first (global) segment of the IR as shown in Figure 4 (lines 4–9). This enables the RTS to choose all the test-cases, leading to safe analysis.

**test_coverage:** This module compiles the instrumented code and executes it against all the test-cases (Σ) for generating test-coverage. A bottleneck while parallelizing coverage generation is the sequential I/O involved in storing the coverage information (mapping from test-cases to basic blocks). Mahtab exploits parallelism across test-cases to improve execution time of coverage generation. This is achieved by having a dedicated coverage file per test-case.

**create_fwd_map:** After coverage is recorded as files on disk, Mahtab records the forward mapping from [test-case]→ {basic-block} in memory as an array of linked-lists. Mahtab populates the individual lists in various array locations in parallel across test-cases. Parallelism is achieved by mapping threads to a group of array locations, with each location corresponding to a particular test-case. Note that a list suffices for our purpose, as a test-case is assigned to one thread. Parallelism within a test-case is possible, but not required since the number of multi-core machine busy.

Parallelization of the offline phase is outlined in Algorithm 1. The translateToExecutable function in line 3 transforms the instrumented IR to the executable form. The test-dependency graph map is initialized to NULL in line 7. In lines 5 and 9, computation occurs via parallel coverage-collection and map-population windows, respectively. The function growList in line 9 populates the linked-list of basic-blocks covered by a particular test t_i.

Parallelization of the offline phase becomes challenging due to sequential bottlenecks in IR generation and code instrumentation (which involves disk-file processing).

2.2.2. Online phase

The online phase transforms the incoming new source code into IR. Subsequently, it checks for discrepancies in the global segments corresponding to both IRs. Changes in this segment lead to a conservative selection of all the test-cases. Otherwise, diff determines changed blocks and test-selection is performed by searching the map for test-cases associated with the affected blocks. Finally, a heuristic based on relevance and confinedness of changes prioritizes the regression test-suite for early failure observability.

We now explain various steps of the online phase.

**new IRgen:** The new version source_v1.c arrives at time t_i (t_i > t_0) and is translated into LLVM IR source_v1.ll (Figure 2) using the clang front-end of LLVM.

**crt_global_set_old/new:** This step involves recording global segments corresponding to old and new versions in parallel. There are multiple ways in which the globals can be stored. However, we observed that the members of the global segment may get reordered in the IR due to addition or deletion of global variables or string literals. Hence, we resort to using an unordered set.

**global_compare:** We perform global-sensitive analysis and compute set comparison to detect changes. If the global segments are intact, we detect changes in the rest of the IR. Otherwise, we conservatively select all test-cases (RetestAll) for re-testing the new version.

**fold_temps_old/new:** An important issue in program differencing at the IR arises due to the names of the temporary variables. Temporary variables are typically generated in sequence using a number (such as %1, %2, ...). For instance, due to addition of an instruction, a temporary variable may get used for the new instruction, and then all the succeeding instructions would use the next temporary. Since temporaries are numbered, all the succeeding instructions show an unnecessary diff with respect to the original version. Ideally, only the structural diff should be computed (which is a bigger problem beyond the scope of this work). To circumvent this specific issue, we fold unnamed temporaries in IR, matching the regular expression %[0-9][0-9]* to a sentinel $ as highlighted in Figure 8 (lines 4 and 5). These two modules (on the old and the new versions) execute in parallel.

**unix_diff_grep_tail:** Former literature has advocated the use of Linux diff utility (Hunt and McIlroy, 1976; Saha et al., 2015; Elbaum et al. 2002; Vokolos and Frankl, 1997; 1998; Elbaum et al. 2000; Rothermel et al. 2001; Romano et al. 2018)
Algorithm 2: Online-phase Parallelization.

Input: Old-version IR: source.ll, Test-suite: Σ, Test-coverage graph: map[1..|Σ|], New-version: source_v1.ll; Window-size: |ω|

Output: Dynamic array: selected_tests

1 /* parallelizing execution of modules */
   source_v1.ll := new IR gen (source_v1.ll);
   /* parallelizing execution of modules */
   Parallel (crt_global_set_old and crt_global_set_new) /*
   crt_global_set_old := crt_global_set_old (source.ll),
   (source_v1.ll := fold_temps_new (source_v1.ll));
   File mydiff := unix_diff grep tail (source.ll, source_v1.ll);
   Set δ := prc_mydiff_crt_affblk_set (mydiff);
   Array affected block[1..|δ|] := crt_affblk_array (δ);
   Parallel (global_sets) := global compare (global_sets, 1); /*
   if changed := false then */
   Parallel (fold_temps_old and fold_temps_new) /*
   fold_temps_old := fold_temps_old (source.ll),
   fold_temps_new := fold_temps_new (source_v1.ll);
   File mydiff := unix_diff grep tail (source.ll, source_v1.ll);
   Set δ := prc_mydiff_crt_affblk_set (mydiff);
   Array affected block[1..|δ|] := crt_affblk_array (δ);
   Parallel (global_sets) := global compare (global_sets, 1); /*
   if changed := false then */
   Parallel (fold_temps_old and fold_temps_new) /*
   fold_temps_old := fold_temps_old (source.ll),
   fold_temps_new := fold_temps_new (source_v1.ll);
   File mydiff := unix_diff grep tail (source.ll, source_v1.ll);
   Set δ := prc_mydiff_crt_affblk_set (mydiff);
   Array affected block[1..|δ|] := crt_affblk_array (δ);
   Parallel (global_sets) := global compare (global_sets, 1); /*
   if changed := false then */

   for i := 1 to 32 in (i) parallel do
     for j := 1 to |δ| do
       List temp := map[i]
       while temp ≠ NULL do
         if temp.block_name == affected block[|δ|] then
           /* match found, select test ti */
           map[i], selected = 1;
           break;
         end
       end
       if map[i], selected = 1 then
         break;
       end
     end
     if map[i], selected = 1 then
       map[i], rank = setAward ()
     else
       map[i], rank = 0;
     end
   end
   selected_tests := sort_tests (map);
   if changed := false then
     selected_tests := Σ;
   end

Yang et al. [2014] Le and Pattison [2014] Palkareva et al. [2016] Yi et al. [2015]. This module invokes diff and the output is filtered using grep and tail and is restricted to contain function names (define), basic-block labels (:), added (>), deleted (<), and modified (!) portions of the IR. A minor issue in this phase is that comments do not appear against the entry block of main() (illustrated in Figure 8). This situation is remedied by passing the flag (-e "*:") to grep for block labels. The output of this step is stored into a disk-file for further processing.

prc_mydiff_crt_affblk_set: Mahtab scans the filtered output for (function name, block name) pairs and records in memory as a set of affected blocks. In Figure 2, the result is a singleton set: (main, if.then).

crt_affblk_array: The set of affected blocks is a collection and is not thread-safe. Therefore, we transform it into a contiguous chunk to enable multi-threaded processing during the later stages of this phase.

prioritize_tests: Each test-case is awarded a value based on the amount of overlap between its coverage and the set of affected blocks. The value depends on the underlying prioritization heuristic. In case of zero overlap, a test-case is not selected, and we award a value ∞. The awarding is performed in parallel for various test-cases. In Figure 5, array locations and the affected blocks corresponding to the selected test-cases are shaded in gray. Parallelism is achieved by accessing array elements (each of them being a list of basic-blocks) independently.

sort_tests: The test-cases are sorted by award value and only the tests with value other than ∞, are selected. Ties may occur in the prioritized permutation due to rank collisions, which is resolved by stability (instability) of the underlying sorting algorithm employed. In our example, the sorted dependency graph (with ties broken by test-case identifiers) is shown in Figure 9.

Figure 8: Folding of unnamed temporaries in LLVM IR.

Figure 9: Sorted test-case dependency graph.

Parallelization of the online phase is outlined in Algorithm 2. Lines 2 and 5 execute their code inside parallel constructs in two-way parallel fashion. Implicit barriers exist immediately after lines 2 and 5 for synchronization. If global segments are intact, the function call in line 5 returns false. The function setAward at line 24 assigns heuristic-specific values when a test-case in selected, otherwise ∞ is awarded.

Limitations of online-phase parallelization

Challenges arise due to sequential static analyses involving disk-file processing: (i) IR generation corresponding to new version, (ii) creation of global sets, (iii) comparison of sets, (iv) normalization of IR temporaries, (v) diff and its post-processing, and (vi) sorting of test-cases. Although prioritization is computed via parallelization windows, the above bottleneck acts as hindrance towards parallelism in the online phase.
Parallelization of the execution phase is outlined in Algorithm 3. The source code for the new version is translated to executable in line 2. Subsequently, the program is executed with parallel windows where execution instance $i$ runs test-case $t_i$ in $\text{selected}_{\text{tests}}$.

**Limitations of execution-phase parallelization**

Challenges for parallelizing the execution phase are due to: (i) inherent sequentiality of the program-under-test, and (ii) unbalanced test-execution paths through the associated control-flow graph.

### 2.3. Parallelism Style of Mahtab

When all phases are considered end-to-end, we observe an *ice-cream-cup* style of parallelism as shown in Figure 11 (right), as compared to its sequential counterpart (left). As conventional RTS is sequential, control-flow is dictated by a single thread of execution. On the other hand, Mahtab exhibits phase-wise multi-threading as shown with multiple threads of control. Phase change involves significant delays (shaded in green) due to barrier synchronization and passage of inter-phase control-flow. Parallelism available in offline and execution phases is due to the presence of dominating dynamic analyses using multi-threaded windows. The online phase exhibits a narrow bottleneck (Figure 11 (right)) due to a long chain of sequential dominance as explained in Section 2.2.2. On an average, the base (execution phase) of the ice-cream-cup is relatively smaller as fewer test-cases need to be executed than all the test-cases (full coverage) involved in the offline phase. In worst-case test-selection, that is, RetestAll, the parallelism takes the shape of an hourglass as execution phase runs all test-cases.

### 2.4. Prioritization Strategy in Mahtab

While executing test-cases using parallelization windows, the test team may choose a particular ordering for early identification of faults (failures). Priorities to test-cases can be assigned in various ways.

*Confinedness*. This strategy prioritizes tests based on change propagation. Priority is given to that test-case which covers affected blocks and, at the same time, as few as possible of the unaffected blocks. This heuristic tries to propagate potential fault-inducing changes through shorter execution paths from fault potential-sites to the terminating basic block. This way
the propagation occurs faster than the case when too much path diversion happens through unaffected code sections. Confinedness of a test-case \( t \) is defined as: \( \text{conf}(t) = |\text{cov}(t) - \Delta|^{-1} \), where \( \text{cov}(t) \) is the test coverage and \( \Delta \) is the set of affected blocks. The higher the value of confinedness of a test-case, the higher is its priority.

**Relevance.** This strategy prioritizes tests based on change coverage. Priority is given to that test-case which covers more affected blocks as much as possible without considering the coverage of additional basic blocks. This heuristic tries to cover as many potential fault-inducing changes as possible via a single execution path, irrespective of the path’s length. Relevance of a test-case \( t \) is defined as: \( \text{rel}(t) = |\text{cov}(t) \cap \Delta| \), where \( \text{cov}(t) \) is the test coverage and \( \Delta \) is the set of affected blocks. The higher the value of relevance of a test-case, the higher is its priority.

**Relevance and Confinedness.** **Mahtab** uses a combination of the two heuristics: relevance and confinedness. Priority is given to that test-case which covers as much change as possible but still remains focused to the potential fault-sites via shorter propagation paths. For a test-case \( t \), the combined value is calculated as sum of: (i) normalized relevance, and (ii) confinedness:

\[
\text{relcon}(t) = \frac{|\text{cov}(t) \cap \Delta|}{|\Delta|} + \frac{1}{|\text{cov}(t) - \Delta|} \tag{1}
\]

where \( \text{cov}(t) \) is the test coverage and \( \Delta \) is the set of affected blocks. Normalization is necessary to ensure that combined score is not dominated by relevance in its entirety. A higher value of \( \text{relcon} \) implies higher priority. In case of priority collisions, test-cases are applied in the order determined by stability (instability) of the underlying sorting algorithm used for reordering, and breaking ties.

Prioritizations involving these heuristics are illustrated using an example in Figure 12. The set of affected-blocks, \( \Delta = \{b_1, b_4, b_7\} \), is shown on top, followed by test-coverage (forward) map annotated with award-valuation in parentheses. Finally, the prioritized test-case permutation and the associated heuristic are shown at the bottom. For this example, each test-case covers at least one of the affected blocks. As a result, all test-cases are selected. The difference is reflected via prioritization score and its associated permutation. As depicted in Figure [12] although \( t_1 \) and \( t_2 \) have the same relevance score (3) due to equal magnitude of intersection with \( \Delta \), their differ via confinedness scores: \( t_1(0.5) \) versus \( t_2(0.2) \). This implies that although both \( t_1(3) \) and \( t_2(3) \) cover the same number of affected blocks, \( t_1(0.5) \) is more confined than \( t_2(0.2) \), due to shorter coverage of additional unaffected blocks. \( t_1 \) covers two additional unaffected blocks: \( \{b_2, b_3\} \), whereas \( t_2 \) covers five additional unaffected blocks: \( \{b_2, b_3, b_5, b_6, b_7\} \). Combining relevance (normalized) and confinedness, \( t_1 \) and \( t_2 \) score 1.5 and 1.2, respectively. Although the tie is broken by combination, confinedness alone was able to achieve this. The situation where relevance acts as the rectifier, is explained as follows. If we consider \( t_2(0.2) \) and \( t_9(0.25) \) due to confinedness scores, \( t_9 \) is more confined than \( t_2 \), but ideally one should prefer \( t_2 \) to \( t_9 \). This is because \( t_2 \) covers two more affected blocks: \( \{b_4, b_7\} \), as compared to \( t_9 \) covering only \( \{b_4\} \). This is rectified in the combination as \( t_2(1.2) \) preferred to \( t_9(0.59) \), which clearly demarcates between the two tests and provides a more informed ordering.

We now discuss metrics for quantifying the effectiveness of prioritization strategy employed in **Mahtab**. We investigate three potential goals of test-suite prioritization: (i) software-fault detection, (ii) code-change coverage, and (iii) test-case failure observation. The corresponding metrics are APFD, ECC, and EPS, respectively, which we discuss below.

### 2.4.1. Average Percentage Faults Detected (APFD)

APFD is one of the most widely used [Miranda et al., 2018] metrics to evaluate the effectiveness of a test-case prioritization technique based on fault-detection rate. This is defined as:

\[
\text{APFD} = 1 - \frac{\sum_{i=1}^{m} T_{F_i}}{n \times m} + \frac{1}{2 \times n} \tag{2}
\]

where \( n \) is the cardinality of the full test-suite and \( m \) is the number of faults. \( T_{F_i} \) is the rank of the first (failing) test-case in the prioritized permutation of test-cases which reveals the fault \( f_i \). The value of APFD varies from 0 to 1. A value closer to 1 implies more fruitful prioritization.

---

![Figure 12: Example prioritizations using heuristics in Mahtab. Scores involved in the combination are highlighted in blue.](image-url)
2.4.2. Effectiveness of Change Coverage (ECC)

ECC is built upon APFD and evaluates prioritization based on coverage rate of affected blocks. It is defined as follows:

$$ECC = 1 - \frac{\sum_{i=1}^{\delta} x_i}{n \times \delta} + \frac{1}{2 \times n}$$

(3)

where $n$ and $\delta$ are cardinalities of the full test-suite and the set of affected blocks, respectively. $x_i$ is the rank of the first test-case in the prioritized permutation, up to which all the affected blocks are cumulatively covered. The value of ECC also varies from 0 to 1. Similar to APFD, an ECC value closer to 1 implies more fruitful prioritization.

2.4.3. EPSilon (EPS) and EPSilon$_\omega$ (EPS$_\omega$)

The EPS metric is adapted to consider parallelization windows $\omega$ (EPS$_\omega$) and measures effectiveness of prioritization coupled with benefit of acceleration by multi-threading the execution phase.

Given a test-suite $\Sigma$ with $\sigma$ test-cases, let us consider an instance where $n$ test-cases are selected and prioritized for an execution timeline $T(m) = (\omega_1, \ldots, \omega_m)$ with $m \geq 1$ windows, each of size $|\omega_i| (\geq 1)$. If $k$ failures (failed-windows) are observed as a vector of positions $\vec{P} = (p_1, \ldots, p_k)$ in $T$, then the best-case scenario corresponds to $k$ failures occurring subsequently starting from $\omega_1$. We denote this vector as $\vec{P}^k = (1, \ldots, k)$. The worst-case occurs when no test-case fails and the position vector is represented as $\vec{P}^\infty = (\sigma + 1, \ldots, \sigma + k)$ where $k$ imaginary failures are assumed beyond the size $\sigma$ of the full test-suite. This assumption is necessary for comparison with vectors having dimension $k$. Choice of the starting position $\sigma + 1$ is justified as follows. The worst case occurs with conservative test selection (100%) and parallel execution windows of size $|\omega| = 1$, i.e., sequential test-execution, and we do not observe failures even if we run all $\sigma$ test-cases. This situation arises when the version-under-test is fault-free or contains faults undetectable by the currently applied test-suite. The hope is to add $k$ more fictitious fault-revealing tests to the existing set of $\sigma$ test-cases. As this is the worst-case, test-execution is sequential and hence, $k$ failures appear sequentially. To avoid position conflicts with the existing $\sigma$ test-cases, the imaginary tests are placed immediately (+1) after the last test-case in $\Sigma$.

The distance vector $\vec{d}(P)$ is calculated in terms of Manhattan distance between $\vec{P}$ and $\vec{P}^i$:

$$\vec{d}(P) = \sum_{i=1}^{k} |P_i - P^i|$$

(4)

We calculate the distance value $\epsilon(P)$ as the sum of the components in $\vec{d}(P)$. Maximum distance vector $\vec{d}_{max}$ is calculated by substituting $P_i$ by $P^i$ in Equation 4.

$$\epsilon(P) = 1 - \vec{d}(P)$$

(5)

Finally, the distance vector $\vec{d}(P)$ normalizes to $\hat{d}(P)$ using $|\vec{d}_{max}|$ and similarity value $\epsilon(P)$ (EPS) is calculated as in Equation 5. EPS ranges from 0 to 1, quantifying degree of similarity with the best-case. The higher the value, the better is the prioritization for early failure observation. Geometrically, EPS measures the shortest-path similarity in $k$-dimensional discrete hyper-volume.

Example of failed test-cases and windows is presented in Figure 13 for sequential test-execution (top), execution windows of sizes 5 (left) and 3 (right). For this illustration $n = 5$, $\sigma = 10$, and failing test-cases are test7 and test9. For sequential test-execution, there is only one execution unit $E$, while five test-cases are sequentially applied against the executing program. Failing test-cases are highlighted and corresponding time-frames are shaded in red. In the parallel setting with five threads and execution units, the window of size 5 is considered failed due to last two test-cases even if the first three test-cases pass (highlighted in blue). For a window of size 3, the first window has no failing test-cases but the last fragmented window has two failing test-cases. Although two test-cases fail for $|\omega| = 5$ and $|\omega| = 3$, the failed window count is $k = 1$ in both the cases. On the other hand, sequential execution has $k = 2$ failures.
Computation of EPS and EPS\(_{\infty}\) for Figure 1 are shown in Table 1. First column enlists intermediate components used in EPS calculations. Subsequent columns quantify corresponding intermediate calculations when test-execution is sequential, execution window of width 5, and 3, respectively. Associated EPS values are shown in the last row for each configuration. We find that sequential execution (|\(u| = 1\)) achieves the lowest value of 0.7, whereas when windows of size 3 and 5 are employed, EPS boosts to 0.9 and 1.0, respectively. Corresponding interpretation says that execution of the test-permutation \(\langle test1, test3, test5, test7, test9\rangle\) using parallel windows of size |\(u| = 1, 3, 5\} achieves (70%, 90%, 100%) similarity in terms of failure observability, compared to an optimal permutation.

2.4.4. Limitations of APFD

Although the APFD metric has been widely used since its inception, it has two limitations.

- Calculation of APFD is possible only when prior fault-knowledge is available (Srivastava, 2005). Our EPS and ECC metrics (described next) address this limitation. We perform a fault-blind analysis and the mapping: [test-case] \(\leftrightarrow\) [basic-block] for calculation of ECC is known during the offline phase when test coverage is collected.

- If \(f\), \(k\), and \(t\) are the number of faults, failures, and test-cases, the space complexity is \(O(ft)\) for calculating APFD. This is due to the associated fault-map (matrix) which takes \(O(ft)\) space and acts as a prerequisite for computing APFD values. In addition, an array length \(f\) can store at each index \(t\) the rank of the first test-case revealing fault \(f\).

Structure of evaluated artifacts (Dataset #1)

- Twelve real-world artifacts from SIR (Do et al., 2005) were selected.
- Out of the twelve programs, eight subjects are of type single-source multiple-versions, where gold output corresponds to the base version.
- Four of the remaining are sequential versions corresponding to various releases. In this case, gold output corresponds to clean releases and the corresponding faulty release is tested.  

3. Experimental Results

We evaluate Mahtab using real-world programs from Software-artifact Infrastructure Repository (Dataset #1) (Do et al., 2005), used extensively in software regression testing research, and five projects from GitHub (Dataset #2). Our evaluation is driven by the following research questions.

- **RQ1**: What is the overall speed-up achieved by our parallel framework phase-wise and end-to-end?
- **RQ2**: What is the effectiveness of the proposed test-suite prioritization technique?
- **RQ3**: How does Mahtab perform against the RetestAll approach?
- **RQ4**: Does Mahtab scale to large codes?

### 3.1. Experimental Setup

All the experiments were performed on a system with a 20-core (40 threads with hyper-threading) Intel Xeon CPU E5-2640 v4 clocked at 2.40GHz having 64GB RAM running CentOS Linux release 7.5.1804 (Core) operating system. While Mahtab framework was compiled using g++ + 5.3.1, the benchmark programs were compiled using clang frontend of LLVM.

Characteristics of the programs in our testbed, denoted as Dataset #1 and Dataset #2, are presented in Table 2 and Table 3, respectively. In Table 2, the first column lists the benchmark (project) name. Number of physical source lines of code (SLOC) for the base version, number of versions, number of test-cases, number of functions in the program are listed in subsequent columns. Next column mentions the average cyclomatic number (ACCN) (McCabe, 1976) for each program, depicting the code structure complexity. The last column shows the average end-to-end sequential regression testing time in seconds, the average being computed across all the versions of a subject. The rows are sorted by the total SLOC count spanning all versions. In Table 3, which lists Dataset #2, the additional column (#Stars) specifies the number of stars received by the project in GitHub, denoting its popularity.

![Data and plots available at (Mondal and Nasre, 2019).](https://github.com/cr-marcesteves/shalcollisionsdetection)
![https://github.com/lotabout/write-a-C-interpreter](https://github.com/beegowyfriday/CuckooFilter)
![https://github.com/rapixr/c4](https://github.com/rapixr/c4)
![https://github.com/rapixr/c4](https://github.com/rapixr/c4)
![https://github.com/rapixr/c4](https://github.com/rapixr/c4)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SLOC</th>
<th>#Ver.</th>
<th>#Tests</th>
<th>#Fun.</th>
<th>ACCN</th>
<th>Seq. (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>6200</td>
<td>39</td>
<td>13555</td>
<td>136</td>
<td>6.5</td>
<td>155.35</td>
</tr>
<tr>
<td>grep</td>
<td>7986</td>
<td>6</td>
<td>809</td>
<td>144</td>
<td>12.6</td>
<td>70.56</td>
</tr>
<tr>
<td>flex</td>
<td>8737</td>
<td>5</td>
<td>670</td>
<td>151</td>
<td>9.2</td>
<td>26.12</td>
</tr>
<tr>
<td>sed</td>
<td>5504</td>
<td>5</td>
<td>370</td>
<td>83</td>
<td>15.2</td>
<td>25.60</td>
</tr>
<tr>
<td>gzip</td>
<td>4342</td>
<td>6</td>
<td>214</td>
<td>30</td>
<td>10.0</td>
<td>90.74</td>
</tr>
<tr>
<td>replace</td>
<td>512</td>
<td>33</td>
<td>5542</td>
<td>21</td>
<td>5.5</td>
<td>62.95</td>
</tr>
<tr>
<td>totinfo</td>
<td>346</td>
<td>24</td>
<td>1052</td>
<td>7</td>
<td>6.7</td>
<td>15.51</td>
</tr>
<tr>
<td>tcas</td>
<td>135</td>
<td>45</td>
<td>1608</td>
<td>9</td>
<td>3.8</td>
<td>19.30</td>
</tr>
<tr>
<td>printtokens2</td>
<td>401</td>
<td>11</td>
<td>4115</td>
<td>19</td>
<td>5.3</td>
<td>47.54</td>
</tr>
<tr>
<td>schedule</td>
<td>292</td>
<td>10</td>
<td>2650</td>
<td>18</td>
<td>3.0</td>
<td>32.82</td>
</tr>
<tr>
<td>schedule2</td>
<td>297</td>
<td>11</td>
<td>2710</td>
<td>16</td>
<td>4.0</td>
<td>41.61</td>
</tr>
<tr>
<td>printtokens</td>
<td>475</td>
<td>8</td>
<td>4130</td>
<td>18</td>
<td>5.6</td>
<td>54.16</td>
</tr>
</tbody>
</table>

Table 2: Characteristics of benchmarks from SIR (Dataset #1).
Structure of evaluated artifacts (Dataset #2)

- Five active projects from GitHub written in C, having more number of stars, were selected. Projects with at least one of the following characteristics were omitted: (i) very few test-cases ($\leq 100$), (ii) absence of `main()` function, (iii) usage of multi-threading, and (iv) usage of non-determinism such as usage of `rand()` and functions producing time-varying outputs.

- Projects have sequential versions corresponding to commits. Starting from the latest revision, older commits involving code changes and passing builds, were manually selected.

- Gold output is associated with the oldest revision. Each test-case corresponds to the latest revision.

Measurement Strategies

The end-to-end analysis time takes into account the time spent during all three phases (offline, online, and execution). This time is a mixture of CPU and wall-clock time on our system. The wall-clock time is due to running the executable (for each test-case) via `system()` function call from within `Mahtab`. In the sequential implementation (Table 2 and Table 3), we leverage the Linux utility `time` to capture running times (`user+system`, i.e., CPU time) of invoked executables (sequentially) outside our engine. The wall-clock time is attributed by scheduling overhead of the running executable. For parallelization, we have specifically used the OpenMP parallel construct, `#pragma omp parallel for`. With timers placed outside the body of for, calls to `system()` from inside, invoke executables in parallel. Although each iteration is processed by a different thread, the executable-invocation by `system()` leads to `fork()` and `exec()` system calls. The timers, in this case, capture the CPU-time of launching the `system()` function, and wall-clock time of executing the invoked executable which gets captured as the elapsed time between call and return of `system()`.

APFD, ECC, and EPS behave differently based on the type of the test-suite provided and the nature of changes. Their behavior is guided by (i) the change-sites, (ii) positions of the affected blocks in the control-flow graph, and (iii) the global sections. Global sections play a vital role in determining the safety of test selection and the approach of selective re-testing (RetestAll versus RTS). By safety of test selection we mean that the set of failing test-cases reported by RTS matches exactly with the set of failing test-cases by RetestAll approach. In other words, a safe selection does not miss out on any failing test-case. The very purpose of testing is to detect failures, and safety of our analysis follows from this safe selection. `Mahtab` treats the global section as a start-up block (critical) which must be visited by any control path through the program. Therefore, all tests are conservatively chosen when its global section changes.

3.2. RQ1: Overall Performance

The overall phase-wise improvement achieved across Dataset #1 is shown in Figure 14. The speedup for the of-
As for Dataset #2, the speedup for the online phase is minimal compared to the other two phases. This is primarily due to sequential bottleneck that involves disk-file processing. The execution phase speedup reaches a maximum value of 4.04× for 32 threads.

The overall end-to-end speedup across Dataset #1 is shown in Figure 16 (left). We observe a peak end-to-end speedup of 4.72× for 16 threads. Mahtab’s analysis across the three phases is dynamic-static-dynamic in nature. A further increase in the number of threads exhibits dip in observed speedup caused by context-switching overheads during dynamic analyses. For Dataset #2, the overall end-to-end speedup is shown in Figure 17 (left). The geometric mean value reaches a maximum of 3.90× for 32 threads. This is attributed by trace collection.

### Table 6: Execution-phase speedup.

<table>
<thead>
<tr>
<th>Dataset #2</th>
<th>#Threads</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdc</td>
<td>0.69</td>
<td>1.24</td>
<td>1.32</td>
<td>1.36</td>
<td>1.39</td>
<td>1.27</td>
</tr>
<tr>
<td>xc</td>
<td>1.43</td>
<td>1.17</td>
<td>1.15</td>
<td>1.28</td>
<td>1.50</td>
<td>1.64</td>
</tr>
<tr>
<td>ckf</td>
<td>1.59</td>
<td>1.33</td>
<td>0.84</td>
<td>0.76</td>
<td>0.63</td>
<td>0.47</td>
</tr>
<tr>
<td>c4</td>
<td>2.42</td>
<td>1.27</td>
<td>1.51</td>
<td>1.89</td>
<td>2.23</td>
<td>2.53</td>
</tr>
<tr>
<td>mli2p</td>
<td>2.22</td>
<td>1.09</td>
<td>0.94</td>
<td>0.90</td>
<td>0.81</td>
<td>0.59</td>
</tr>
<tr>
<td><strong>Geometric mean</strong></td>
<td>1.83</td>
<td>1.22</td>
<td>1.07</td>
<td>1.17</td>
<td>1.19</td>
<td>1.08</td>
</tr>
</tbody>
</table>

### Figure 16: Overall end-to-end speedup and boost factor achieved across benchmarks (Dataset #1).

### Figure 17: Overall end-to-end speedup and boost factor achieved across Dataset #2.
Regression test-suite prioritization exhibits gradual increase in EPS values (EPS boost factor (EBF)) on increasing the number of threads. Overall EBF values for Dataset #1 and Dataset #2 are shown in Figure 16(right) and Figure 17(right), respectively against the number of threads employed. The monotonically increasing profile reaches a peak value of 1.62x and 1.43x for 32 threads, respectively. Although boost values are not as high as speedup factors, they signify that 32-threaded execution windows help us achieve 62% and 43% increase in effectiveness of regression test prioritization. Unlike speedup values, boost factor profiles do not exhibit a dip because the former measures execution time which may be affected by synchronization and load-imbalance overheads. This problem does not arise in the latter case as it considers only effectiveness values.

### 3.2.1. Parallelization of Offline Phase

Table 4 shows the speedup achieved for each benchmark (Dataset #1) in the offline phase. The highest speedup for each subject appears in bold font. The maximum speedup of 9.77x was achieved by printtokens with 16 threads. Offline speedup was observed to be directly proportional to the number of threads for grep, flex, sed, and gzip. For all other benchmarks, peak speedup was achieved with 16 threads. For Dataset #2, projects xc, ckf, and c4 exhibit peak speedups of 3.68x, 5.11x, and 7.00x, respectively with 32 threads, whereas scd and mlisp achieve the maximum speedups of 3.29x and 8.09x, respectively for 16 threads. Since majority of computation in this phase is driven by dynamic analysis in a multi-processed manner, the available parallelism is higher compared to other two phases. It is to be noted that in offline phase, all test-cases are executed in-order as opposed to executing (prioritized order) significantly smaller amount of selected test-cases in execution phase. The offline phases in Dataset #2 show lesser speedups compared to Dataset #1. This is due to significantly higher disk-space consumed per test-case (Dataset #2), leading to higher values of execution time, in the offline phase.

### 3.2.2. Parallelization of Online Phase

Table 5 shows the speedup achieved for each benchmark (Dataset #1) in the online phase. The highest speedup for each subject appears in bold font. The maximum speedup of 2.33x was achieved by printtokens2 with 32 threads. Sequential bottleneck being a major hindrance to effective parallelization, slowdown reached a maximum of 5% due to 8 threads for gzip. For Dataset #2, peak speedup of 153% was observed for c4 with 32 threads. Maximum slowdown of 53% was observed for ckf with 32 threads. Geometric mean peak speedup for Dataset #2 reached a value of 1.22x for only 2 threads. The second highest benefit of 19% was obtained for 16 threads. This is expected as global sensitivity was turned on by default which resulted in conservatively following the RetestAll approach for more than 95% of all versions in this dataset. This resulted in bypassing parallelization opportunities which is otherwise mandatorily available when global sections are not analyzed. Effectively speaking, Dataset #2 exhibits a two-threaded online phase parallelization with remaining threads only negatively contributing towards performance by adding synchronization overheads. Unlike in the offline phase, the speedup values in the online phase do not differ significantly. This is due to a slower overall online phase where sequential computation dominates the total execution time.

#### 3.2.3. Parallelization of Execution Phase

Table 6 shows the speedup achieved for each benchmark (Dataset #1) in the execution phase. The highest value for each subject appears in bold font. The maximum speedup of 10.40x was achieved by tcas for 16 threads. Worst-case slowdown was observed for space at 77% for 2 threads. The primary reason for the slowdown is extensive file I/O and context-switching among threads. For space, the geometric mean test selection (Table 7) turns out to be 18.02% (2448 out of 13,585) which is 122.4x the number of available cores; this led to an increased overhead in scheduling and context-switching. For flex, sed, and gzip, the speedup increases monotonically with increase in the number of threads. Other benchmarks from SIR (grep, replace, totinfo, printtokens, printtokens2, schedule, and schedule2) exhibit maximum benefit with 16 threads, beyond which we observed a dip in execution speedup. The speedup behavior in execution phase is dominated by dynamic analysis which executes test-cases and verifies their outcomes. For Dataset #2, peak execution phase speedup of 10.69x was achieved for ckf with 32 threads. Projects scd, c4, and mlisp exhibit maximum speedup values of 2.74x, 3.79x, and 3.53x for 8 threads, respectively beyond which doubling the window size introduces context-switching (with hyper-threading enabled) which resists parallelization. However, xc and ckf reached peak speedups of 6.19x and 10.69x, respectively for up to 32 threads.

<table>
<thead>
<tr>
<th>Dataset #1</th>
<th>sel_on (%)</th>
<th>sel_off (%)</th>
<th>#fails_on</th>
<th>#fails_off</th>
<th>#faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>15.65</td>
<td>71840</td>
<td>71840</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>grep</td>
<td>3.91</td>
<td>857</td>
<td>69</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>flex</td>
<td>99.70</td>
<td>2649</td>
<td>2649</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>sed</td>
<td>99.00</td>
<td>501</td>
<td>501</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>99.53</td>
<td>417</td>
<td>417</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>replace</td>
<td>36.83</td>
<td>3422</td>
<td>3422</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>totinfo</td>
<td>65.92</td>
<td>1926</td>
<td>1926</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>tcas</td>
<td>59.40</td>
<td>1579</td>
<td>1579</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>printtokens2</td>
<td>50.14</td>
<td>2237</td>
<td>2237</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>schedule2</td>
<td>92.83</td>
<td>296</td>
<td>296</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>printtokens</td>
<td>41.30</td>
<td>484</td>
<td>270</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Geometric mean</td>
<td>64.55</td>
<td>1455</td>
<td>1124</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset #2</th>
<th>sel_on (%)</th>
<th>sel_off (%)</th>
<th>#fails_on</th>
<th>#fails_off</th>
<th>#faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>scd</td>
<td>80.00</td>
<td>1160</td>
<td>1160</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>xc</td>
<td>40.58</td>
<td>2100</td>
<td>2100</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>ckf</td>
<td>100.00</td>
<td>945</td>
<td>945</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>c4</td>
<td>62.26</td>
<td>5129</td>
<td>4222</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>mlisp</td>
<td>100.00</td>
<td>2394</td>
<td>2394</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>Geometric mean</td>
<td>74.36</td>
<td>70.13</td>
<td>1951</td>
<td>1877</td>
<td></td>
</tr>
</tbody>
</table>
3.2.4. Effect of Global-sensitivity on Performance

We compare performance and safety of Mahtab in two modes. The first mode (globals_on) is the default mode and involves detecting change in global segments of LLVM IR. The second mode (globals_off) skips this step and detects changes only in the basic-blocks of various functions (including main()). The latter mode follows several works from literature. Note that globals_off may result in unsafe RTS. Results for both the modes are presented in Table 7 in terms of test selection ratio (sel_on vs sel_off), i.e., the percentage of test-pool selected for testing regressions, and the total number of failures (#fails_on vs #fails_off) observed across all the versions tested for each benchmark. The safest value of selection ratio and corresponding observed failures for each subject, appear in bold font. Our analysis assumes zero fault-knowledge, which occurs in the real setting. The numbers along #fails_on and #fails_off report the number of failures, i.e., the number of selected test-cases that failed. This is greater than the number of seeded faults available for SIR programs, even if the cumulative count is considered across versions. This is because for some subjects from SIR (grep, flex, sed, gzip), multiple faults are kept active simultaneously by turning on all available #define macros in the header FaultSeeds.h corresponding to faulty versions. In Table 7 the sixth column titled #faunts displays the number of faults cumulatively considered across all the chosen versions in our dataset. For example, flex has four faulty versions and the number 62 (#faunts) against flex is the total number of faults summed up across all four FaultSeeds.h header files for the faulty versions. The same explanation holds for grep, gzip, and sed. Only in case of single faults, for space and Siemens programs, the number of faults is equal to the number of available versions.

Discussion. It is the nature of code-under-test (memory-intensive versus I/O-intensive), type (change in global versus non-global segments) and amount (percentage of blocks affected) of change, and change-sites (criticality of affected blocks) that play a dominating role in choosing the trade-off between safety and precision. Memory-intensive subjects are likely to have light-weight and short running test-cases, whereas typical I/O-intensive subjects mostly involve disk-file processing. In the former case, globals_off may be preferred from the viewpoint of precision where conservative RetestAll approach should be avoided due to strict time budgets. For lightweight subjects, conservative RetestAll due to globals_on, may still be feasible. Type and amount of change can be speculated by difference in timestamps between the version-pair under change-analysis. Greater differences indicate major changes, in which case the global section may be bypassed (globals_off). This is almost always safe as major changes involve \( \approx \)100% test-selection. This necessitates test-prioritization which is otherwise not available with globals_on but always guaranteed with globals_off, and major code revisions have high possibilities of global segments being changed. Another factor affecting amount of test-selection is criticality of the changed blocks. A critical basic-block (for example, the entry block of main()) is always visited by any test-case and when affected, leads to a worst-case selection of all test-cases. If the mode of analysis is globals_off, prioritization of test-cases are enforced which provides opportunities for further test-case pruning by running top-\( k \) of the prioritized permutation. This is fruitful in case of shorter time budgets.

For Dataset #1, printtokens and grep are two subjects where globals_off resulted in 214 and 788 missed failures, respectively. Corresponding table entries show bigger difference in selection ratio compared to other benchmarks (Table 7). For others, except space, marginal or no difference is observed in selection ratio. As a result, no failure was missed. Surprisingly, for space, a reduction in test selection from 18.02% to 15.65% did not result in missing any of the 71,840 failures. For Dataset #1, the geometric mean test selection turns out to be 64.55% with 1,455 observed failures for globals_on.
and 47.11% with 1,124 observed failures for globals_off. In comparison, for Dataset #2, the geometric mean test selection turns out to be 74.36% with 1,951 observed failures for globals_on, and 70.13% with 1,877 observed failures for globals_off. For xc, the amount of test selection turned out to be invariant to global sensitivity. On an average, test selection for this project remained ≤40%. In case of acd, the reduction of 10 percentage points in average test selection did not miss any failure when the global section was not processed. However, the same amount of reduction for c4 resulted in missing 907 failures. Interestingly, for both ckf and mlisp, global-invariant test selection of 100% was observed consistently across versions. This is due to major changes in global as well as non-global code segments, with every incremental commit (version) for these subjects.

Comparison of speed-up achieved for both the modes (Dataset #1) is shown in Figure 18. As the offline phase is devoid of processing global sections, no difference in speed-up was observed. Turning off global sensitivity only involved processing of basic-block segments in the online phase. However, turning it on introduced additional computation for processing global segments. Typically (which we observe in our dataset too), only a few subjects have their global sections affected, and that too for a few versions. This resulted in extra computational overhead and a comparatively lower observed speed-up when global-sensitivity was turned on (by default). This is shown in Figure 18 (top). On the other hand, in the execution phase, the speed-up observed was more for globals_on due to a significantly higher percentage of selected test-cases (64.55% for globals_on compared to 47.11% for globals_off). A larger number of test-cases favored parallelism with an increasing value of execution window size. This is shown in Figure 18 (bottom). In comparison, speed-up lost in online phase was compensated for during the execution phase, ultimately resulting in similar overall end-to-end speed-ups for both the modes. For 16 threads, overall end-to-end speed-ups were 4.69x for globals_off and 4.72x for globals_on.

For Dataset #2, the speedup comparison is presented in Figure 19 (top). As expected, due to the property of projects in the dataset, globals_on conservatively followed RetestAll approach, by bypassing remaining inter-phase modules. In this case, effective threading factor remained almost sequential due to bottleneck involving disk-file processing. For globals_off, Mahtab bypassed the global sections, forcing the control-flow to reach the most parallelism-favoring module in the online phase: test prioritization. With globals_off, the benefit realized was directly proportional to the window size. This resulted in a larger difference in online-phase speedup for Dataset #2. The discrepancy in the online-phase speedup with increasing number of threads is due to global sections being affected, for almost all the versions. In this situation, a global-insensitive analysis may be preferred. In the execution phase (Figure 19 (bottom)), Mahtab achieves positive benefits on increasing the window size, reaching 4.04x (globals_on) and 3.88x (globals_off), for up to 32 threads. Overall end-to-end speed-ups are 4.04x for globals_off and 3.90x for globals_on.

3.3. RQ2: Effectiveness of Prioritization

We discuss effectiveness of our proposed prioritization heuristic using several analyses spanning different dimensions of regression testing.

3.3.1. APFD, ECC, and EPS Values

The effectiveness of regression test prioritization for both datasets is shown in Table 8 for different metrics. APFD values are shown for Siemens suite and space from SIR repository, where each version contains a seeded fault for Siemens programs and a real fault in case of space. Therefore, we track the rank of the first failing (fault-revealing) test-case for these benchmarks. This can be achieved by simply setting m = 1 while calculating APFD in Equation 2, where m is the number of faults. All other benchmarks (projects) have versions containing multiple-faults and as per our fault-blind analysis, we report their APFD as undefined (—) in Table 8. For each benchmark, the most effective values are shown in bold font.

We observe that APFD reaches a maximum value of 0.93 for schedule. Maximum ECC of 0.99 occurs for totinfo. Minimum ECC of 0.00 was observed in some cases when the changes occur in global segments which are not tracked. It is to be noted that both APFD and ECC are independent of the window size, hence, we report their values only for the sequential regression testing when |ω| = 1. On the other hand, EPS (EPSω) is defined for a window. We observe that higher the window size, higher is the boost (EBF) in the EPS values with respect to EPSω|ω|=1. Minimum EPS was observed as 0.42 for grep (Dataset #1) and 0.07 for acd (Dataset #2) at window size |ω| = 1, while peak EPS of 1.00 was observed for flex, acd, and ckf for |ω| = 32. For APFD and EPSω, a value close to one implies that failing test-cases appear very early in the prioritized permutation. The analogous interpretation for EPS with parallelization windows is that first few windows contain all failing test-cases. A window of size |ω| = k can be viewed as a container where k consecutive sequential positions are mapped to one parallel position. Thus, all failing test-cases inside this window are treated as failing at the same parallel position. It is to be noted that EBF entries in Table 8 are average values for each benchmark, across versions. For example, the entry EBF for space is 1.04 obtained by averaging the EBF across its 38 versions and should not be interpreted as EPS2/EPSS. Therefore, plots in Figure 16 (right) and Figure 17 (right) correspond to geometric mean EBF values across benchmarks in respective datasets.

3.3.2. Mahtab’s Prioritization versus State-of-the-art

Greedy additional prioritization is popularly accepted as a state-of-the-art technique (Rothermel et al., 2001; Elbaum et al., 2002; Luo et al., 2016; Miranda et al., 2018; Chen et al., 2018). The original greedy additional strategy was intended for additional statement coverage prioritization. For our evaluation, this strategy is tailored to greedy additional affected basic block coverage, which we explain below. The technique is iterative in

\[^{11}\text{The subscript } k \text{ in } \text{EPS}_k \text{ denotes the window size } |\omega|=k\]
nature and begins by selecting the test-case which achieves the greatest affected-block coverage. Coverages of all other tests are then modified to reflect only the uncovered basic-blocks. The next test-case with the highest amount of adjusted coverage is greedily selected. This process continues until all the basic-blocks are covered. A situation may arise when the adjusted coverages of the remaining test-cases are empty, even though they cover at least one affected block. Coverages of such test-cases are re-initialized to their initial list of basic-blocks and the algorithm is recursively applied until all the test-cases originally covering at least one basic-block is selected for regression test prioritization.

In the rest of this subsection, we perform a quantitative comparison of Mahtab’s prioritization by relcon (relevance and confinedness, as introduced in Section 2.4) against the greedy-additional technique.

### ECC Values, Affected, and Untested Basic Blocks

Newly added basic blocks will not have coverage information recorded in the test-coverage map corresponding to the older version. This reduces the ECC value as there are no regression test-cases covering these newly added blocks. This reduction is more pronounced for more number of basic blocks added in the new version. Thus, ECC not only measures the rate of change coverage, but also reflects the degree to which new code is added. A very low value indicates major changes besides indicating fruitless prioritization. A very high value indicates minor change besides indicating fruitful prioritization. Therefore, ECC values are also good indicators of the amount of escaped (untested) basic blocks by the existing regression test-cases. Due to this property, ECC values can also be viewed as indicators to refresh test-coverage in the new version. Besides reporting the number of test failures, ECC, and EPS values, Mahtab also reports the percentage of tests selected and the number of blocks affected. For untreated code portions, Mahtab explicitly reports the names of the basic blocks not covered by the existing test-cases. This information may help the development team to introduce additional test-cases for testing the escaped code sections. When used together with APFD values, ECC may act as a supplement by providing a reliability index of or confidence on the test-coverage map. Figure 20 demonstrates these features of ECC using version-wise plots for ECC versus affected blocks, and affected blocks versus escaped blocks for tcas, xc, and ckf. y-axes show ECC values (top row) and percentage of escaped blocks (bottom row). In all the cases, x-axes represent percentage of affected blocks corresponding to the version-under-test, for each subject. ECC values due to additional strategy always remain on or above the associated relcon value. This is expected as faster additional change coverage forms the crux of the greedy additional prioritization, whereas relcon clubs together test-cases having similar coverages without considering the coverage of the already prioritized test-cases. tcas and ckf exhibit similar trends, i.e., when the escape value is high, the ECC value is low, and vice-versa. However, xc exhibits a different profile. ECC value for testing xc (v15) is zero as no test selection happens (see Figure 21 for xc), hence no blocks are escaped. v2 and v3 have extreme values of ECC. Although both have 0.15% affected blocks, the 100% test-selection for v3 covers all the affected blocks. For v5, no test selection with 0.15% affected blocks reports full escape (0.15) with an ECC value of zero. v5 of xc reports 19.49% affected blocks, but even with 100% test-selection, the ECC value is less than one, the amount of escaped blocks being 3.17%. For ckf (v1), the value of 120.40% for affected blocks indicate presence of a significant amount of newly added basic blocks besides changes occurring in existing basic blocks.

### Time Complexity of Prioritization

If s and d denote the number of selected test-cases and the number of affected basic-blocks respectively, then for greedy additional heuristic, the selection-and-adjust step takes $O(sd)$ time for each test-case. This amounts to $O(s^2d)$ time for

<table>
<thead>
<tr>
<th>Dataset #1</th>
<th>APFD</th>
<th>ECC</th>
<th>EPS1</th>
<th>EPS2</th>
<th>EPS3</th>
<th>EPS4</th>
<th>EPS8</th>
<th>EPS16</th>
<th>EPS32</th>
<th>EBF2</th>
<th>EBF4</th>
<th>EBF8</th>
<th>EBF16</th>
<th>EBF32</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>0.87</td>
<td>0.85</td>
<td>0.78</td>
<td>0.84</td>
<td>0.87</td>
<td>0.88</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>1.04</td>
<td>1.11</td>
<td>1.14</td>
<td>1.16</td>
<td>1.17</td>
</tr>
<tr>
<td>grep</td>
<td>—</td>
<td>—</td>
<td>0.00</td>
<td>0.42</td>
<td>0.62</td>
<td>0.72</td>
<td>0.77</td>
<td>0.79</td>
<td>0.79</td>
<td>1.20</td>
<td>1.38</td>
<td>1.47</td>
<td>1.51</td>
<td>1.52</td>
</tr>
<tr>
<td>flex</td>
<td>—</td>
<td>—</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sed</td>
<td>—</td>
<td>—</td>
<td>0.00</td>
<td>0.73</td>
<td>0.87</td>
<td>0.94</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
<td>1.24</td>
<td>1.35</td>
<td>1.42</td>
<td>1.45</td>
<td>1.46</td>
</tr>
<tr>
<td>gzip</td>
<td>—</td>
<td>—</td>
<td>0.00</td>
<td>0.62</td>
<td>0.73</td>
<td>0.78</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.96</td>
<td>1.04</td>
<td>1.07</td>
<td>1.07</td>
<td>1.08</td>
</tr>
<tr>
<td>replace</td>
<td>0.85</td>
<td>0.95</td>
<td>0.68</td>
<td>0.83</td>
<td>0.90</td>
<td>0.94</td>
<td>0.95</td>
<td>0.96</td>
<td>0.96</td>
<td>1.25</td>
<td>1.39</td>
<td>1.46</td>
<td>1.49</td>
<td>1.51</td>
</tr>
<tr>
<td>totinfo</td>
<td>0.87</td>
<td>0.99</td>
<td>0.56</td>
<td>0.79</td>
<td>0.90</td>
<td>0.95</td>
<td>0.98</td>
<td>0.99</td>
<td>0.99</td>
<td>1.59</td>
<td>1.88</td>
<td>2.02</td>
<td>2.09</td>
<td>2.12</td>
</tr>
<tr>
<td>tcas</td>
<td>0.78</td>
<td>0.91</td>
<td>0.69</td>
<td>0.85</td>
<td>0.93</td>
<td>0.97</td>
<td>0.98</td>
<td>0.99</td>
<td>0.99</td>
<td>1.62</td>
<td>1.92</td>
<td>2.07</td>
<td>2.15</td>
<td>2.18</td>
</tr>
<tr>
<td>printtokens2</td>
<td>0.90</td>
<td>0.91</td>
<td>0.55</td>
<td>0.78</td>
<td>0.89</td>
<td>0.95</td>
<td>0.98</td>
<td>0.99</td>
<td>0.99</td>
<td>1.59</td>
<td>1.88</td>
<td>2.03</td>
<td>2.10</td>
<td>2.13</td>
</tr>
<tr>
<td>schedule</td>
<td>0.93</td>
<td>0.98</td>
<td>0.71</td>
<td>0.86</td>
<td>0.94</td>
<td>0.97</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>1.26</td>
<td>1.38</td>
<td>1.44</td>
<td>1.47</td>
<td>1.48</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.74</td>
<td>0.97</td>
<td>0.48</td>
<td>0.70</td>
<td>0.80</td>
<td>0.85</td>
<td>0.88</td>
<td>0.89</td>
<td>0.89</td>
<td>1.44</td>
<td>1.70</td>
<td>1.82</td>
<td>1.88</td>
<td>1.91</td>
</tr>
<tr>
<td>printer2</td>
<td>0.74</td>
<td>0.68</td>
<td>0.47</td>
<td>0.74</td>
<td>0.87</td>
<td>0.94</td>
<td>0.97</td>
<td>0.99</td>
<td>0.99</td>
<td>1.90</td>
<td>2.34</td>
<td>2.56</td>
<td>2.67</td>
<td>2.72</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>—</td>
<td>—</td>
<td>0.62</td>
<td>0.80</td>
<td>0.88</td>
<td>0.91</td>
<td>0.93</td>
<td>0.94</td>
<td>0.94</td>
<td>1.31</td>
<td>1.48</td>
<td>1.56</td>
<td>1.60</td>
<td>1.62</td>
</tr>
</tbody>
</table>

| Dataset #2 |
|------------|------|-----|------|------|------|------|------|-------|-------|------|------|------|-------|-------|
| sdc        | —    | 0.20 | 0.07 | 0.14 | 0.17 | 0.19 | 0.19 | 0.20  | 0.20  | 1.55 | 1.82 | 1.96 | 2.02  | 2.06  |
| xc         | —    | 0.35 | 0.35 | 0.38 | 0.39 | 0.40 | 0.40 | 0.40  | 0.40  | 1.03 | 1.04 | 1.05 | 1.05  | 1.05  |
| ckf        | 0.00 | 0.61 | 0.63 | 0.93 | 0.98 | 0.99 | 1.00 | 1.00  | 1.00  | 1.51 | 1.75 | 1.86 | 1.90  | 1.92  |
| c4         | 0.34 | 0.59 | 0.65 | 0.68 | 0.69 | 0.70 | 0.70 | 0.70  | 0.70  | 1.08 | 1.12 | 1.13 | 1.14  | 1.15  |
| mclsp      | 0.00 | 0.42 | 0.51 | 0.56 | 0.58 | 0.59 | 0.60 | 0.60  | 0.60  | 1.12 | 1.19 | 1.22 | 1.24  | 1.25  |
| Geometric mean | —   | 0.33 | 0.43 | 0.42 | 0.49 | 0.50 | 0.50 | 0.50  | 0.50  | 1.24 | 1.33 | 1.39 | 1.42  | 1.43  |

Table 8: Quantifying effectiveness of prioritization using APFD, ECC, and EPS values.
the entire process. Comparatively, the crux of relevance-and-confinedness (relcon) lies in searching the coverage map which takes $O(sd^2)$ in total, and $O(sd)$ time for computing the test-case priorities. This amounts to $O(sd^2)$ time for relcon. For our datasets, we found $s > d$ and empirically observed relcon to achieve an $O(d^2)$ asymptotic speedup (w.r.t. greedy additional) in the sequential setting. Figure 21 shows version-wise plots for time taken by online phases of tcas, cff, and xc. In this case, we observed relcon to be more time-efficient than greedy additional prioritization. tcas and cff exhibit similar trend in online time. This means that for relcon, online time remains almost constant whereas for greedy additional, online time is in direct proportion to the number of affected blocks. The only difference between tcas and x{c} is that for cff, 100% test selection occurs for every version-under-test as each of them builds a major revision upon the immediately preceding commit (version). For xc, spikes are observed due to $v_4$ and $v_8$ with 64.91% and 19.49% affected blocks, respectively. Large difference with corresponding online-phase times for relcon appear due to the presence of $O(d^2)$ factor that is sensitive to the number of selected test-cases $s$. This boost in online-phase time for xc ($v_2$ and $v_3$) is depicted in Figure 21 (top row). For xc ($v_4$ and $v_5$), although only 0.15% of the blocks are affected, a larger value of test-selection (100% for $v_4$ versus 0% for $v_5$) leads to greater online time for $v_4$. It is to be noted that despite having zero blocks affected and no test-selection for xc ($v_4$ and $v_10$), the online-phase time is still non-zero due to execution of other modules of the online phase. In this case, we observe that online times are indeed the same.
Effect of Prioritization on Test-load Distribution

Load (time) imbalance may arise inside parallelization windows when test-load distribution is uneven. A long running test-case may monopolize the window and render it ineffective for performance benefits. Although they differ heuristically, both greedy additional and relcon are coverage-based prioritization strategies keeping test-cases in close proximity if they have close priority values. Figure 22 shows load distribution profiles during regression testing of mlisp (v4 → v5), xc (v1 → v2), and ckf (v1 → v2). For mlisp (v4 → v5), Load imbalance does not arise if execution phase (including test-execution and verification of outcomes) is operated sequentially. Imbalance comes into play when this prioritized test-permutation is otherwise executed using parallel execution windows. Figure 22 (top), (middle), and (bottom) show associated load distributions for test-permutations due to: (i) unprioritized, (ii) greedy additional, and (iii) relcon prioritizations, respectively. We observe that for mlisp (v4 → v5) and ckf (v1 → v2), load profiles due to additional strategy are similar. This means that if windows are employed, the end-to-end time due to load imbalance will be marginally different. Maximum benefit is observed for mlisp (v4 → v5) where very lightly loaded test-cases appear in the first 50% of the permutation, excluding the first three spikes. This is expected as relcon is time-unaware. Compared to greedy additional, with alternating light and heavy loads, a heavy band occurs due to relcon in the region having test-case ids: 3348 – 2943. Moreover, due to relcon, greater run-lengths of lighter loads are observed. As a result, multi-threaded execution phase of mlisp (v4 → v5) exhibits better load-balance due to prioritization by relcon when compared to that of greedy additional. In this case, the load-imbalance due to no prioritization emulates the load profile of greedy additional, therefore their execution-phase running times will be marginally different. Thus, for mlisp (v4 → v5) relcon outperforms greedy additional by reaching a minimum of 221.23 seconds for up to 32 threads. For xc (v1 → v2) and ckf (v1 → v2), the curves corresponding to execution phase running times almost overlap each other. This is due to similar end-to-end load-imbalance which gives rise to marginally different end-to-end times for the execution phase. The load-pattern due to relcon is similar for xc...
Figure 23: Comparison of (sequential) end-to-end time \{tca\,mli\} for basic-block and function granularities. Test selection ratio is shown for tca only.

Figure 24: Comparison of elements affected, untested (escaped), ECC, and (sequential) EPS values \{tca\} for basic-block and function granularities.

Figure 25: Comparison of elements affected, untested (escaped), ECC, and (sequential) EPS values \{mli\} for basic-block and function granularities.

\((v_1 \rightarrow v_2)\) when compared to no prioritization. In this case, the lowest observed execution phase running time (32 threads) is 6.53 seconds due to relcon and 5.83 seconds due to unprioritized strategy. For cklf \((v_1 \rightarrow v_2)\), load-distribution due to additional and relcon are nearly mirror images of each other. The benefit with relcon in this case is due to the presence of lighter loads earlier in the permutation. This means that under tight time-constraints, top 50% of the permutation due to relcon would be faster than that due to greedy additional. Lowest runtime values of 0.82 seconds (32 threads) and 1.17 seconds (16
3.3.3. Impact of Granularities: Basic-block versus Function

Evaluation of Mahtab for fine-grained (basic-block level) and coarse-grained (function level) for tcas and mlisp are shown version-wise in Figure 23, Figure 24, and Figure 25. End-to-end time for tcas turns out to be lesser for fine-grained approach and is directly proportional to the amount of tests selected. For mlisp, 100% test-selection was consistently observed for all the five versions. In this case, end-to-end time due to coarser granularity was marginally lower. As expected, percentage of elements affected due to basic-block granularity was much less compared to coarse-grained approach. For example, two affected basic-blocks of one function will be reported coarsely as only one function being affected. The most important information is given by the amount of escaped code elements that were not tested. For tcas, the amount of escaped functions reported by function-granularity is consistently zero for all 41 versions, whereas basic-block level analysis safely reports those escaped basic-blocks. The feature of fine-grained analysis may be used as a staring point for test-suite evolution as correlated to code-evolution. Reported block-level ECC values remain on or below function-level values. The trend in ECC values appears complementary to that of escaped entities. One of the features of ECC is that it quantifies the degree to which changed code elements are not tested. This is observed for tcas with function granularity where ECC values mislead us by remaining almost close to one when no elements remain untested. Throughout its 41 versions, Basic-block level analysis rectifies this by reporting low ECC values when basic-blocks truly escape regression testing. EPS values remain higher with fine-grained analysis for tcas for 38 out of 41 versions. In case of mlisp, the granularity profiles flip each other at the region defined by \( v_4 \) and \( v_5 \).

3.4. RQ3: Speedup Achieved against RetestAll

Another dimension of benefit from our proposed approach stems from the end-to-end speed-up achieved when compared with RetestAll_seq (running all test-cases sequentially). We plot the regression testing time by Mahtab against this baseline for each benchmark in (Dataset #1) in Figure 26. We observe that except for space and replace, single-threaded RTS is the slowest for all the benchmarks. This is attributed to the nature of the system-under-test, changes between versions, and the test selection ratio. As depicted from the plots, increasing the number of threads leads to, as expected, a decrease in regression testing time. However, there are exceptions where the testing time increases; this occurs due to scheduling and context-switching. Part of the overhead also stems from hyper-threading enabled in our experiments. For Dataset #1, we summarize the end-to-end RTS time and the corresponding speed-up in Figure 27. The RTS time achieves a peak of 43.35 seconds for a single thread, reaches the lowest value of 9.97 seconds for 16 threads. As the execution platform has 20 cores in total, presence of hyper-threading leads to an increase in RTS time; resulting in 10.51 seconds for 32 threads. The corresponding speed-up profile reaches a peak value of 2.44× for 16 threads. RTS times and speedup values for Dataset #2 are shown in Figure 28. In this case, the major role was played by relatively higher disk space consumed per test-case. Memory distribution comprises of 18.77 MB (2.08 KB per test-case) for acd, 20.52 MB (17.96 KB per test-case) for c4, 753.28 KB (0.20 KB per test-case) for mlisp, 22.66 MB (29.01 KB per test-case) for xc, and 329.77 MB (1.49 MB per test-case) for cTKf. Sequential RTS times (seconds) for acd, c4, and mlisp are 161.29, 43.08, and 237.51, respectively. The heavier subjects xc, and cTKf have RTS times (seconds) of 517.78, and 1925.43, respectively. Due to these reasons, speedup values for acd, c4, and mlisp are in direct proportion to the number of threads, achieving the value of 1.86× for 32 threads. However, xc, and cTKf show negative performance, i.e., a slowdown of 85% even if parallelization windows of size 32 are employed. This is primarily attributed to the heaviness in terms of disk space consumed by its test-suites. Due to heavy disk-IO, their offline phases are highly time-consuming. It is to be noted that RetestAll_seq executes test-cases sequentially, therefore operates only in the execution phase, whereas RTS_seq and RTS_i operate all three phases: offline, online, and execution. We note that a single-threaded execution was found to be less efficient than the sequential implementation in the first place.

3.5. RQ4: Scalability of Mahtab

We performed a scalability study to determine the running time of Mahtab where number of SLOC is in millions. For this, we generated synthetic artifacts described as follows.

Structure of synthetic subjects
- This category comprises of five synthetic artifacts created by our large-code generator. The subjects have been named casedn and have three versions each.
- Our goal here is to generate a complicated program to test scalability of Mahtab. A subject consists of a main() function with \( n \) cases of a large switch construct. Each case consists of a function call to a unique function each containing an if-else-if ladder, which is again of size \( n \). Thus, total number of paths through the program is \( n^2 \). Figure 29 (left) shows how code size varies with control parameter \( n \in \{200, 400, 600, 800, 1000\} \).
- The generated programs comprise of unbalanced execution paths with break statements omitted at random cases of the switch construct. The resulting fall-through helped us generate \( a \times n \) test-cases with non-uniform load distribution in terms of time and basic blocks covered. The scaling factor \( a \) denotes the number of rounds of random test-generation and helps generate a test-pool \( a \) times bigger than the number of cases in main(). We set \( a = 10 \) in our experiments. This choice is driven by the need to show benefit of parallelization windows when both SLOC and test-sets are bigger.
- All five subjects are of type single-source multiple-versions, where gold output corresponds to base version.

Code-change was performed at random functions (other than main) of the base version for generating multiple versions. Geometric mean change per artifact is shown in Figure 29 (middle) in terms of percentage of basic-blocks affected in the base version. Plots for (16-threaded) phase-wise and end-to-end running times are presented in Figure 29 (right). We observe that with increase in SLOC, growth rate of online time is bounded by growth rates of offline time (above) and execution time (below). When all phases are considered together, end-to-end testing time of Mahtab grows quadratically: \( f(x) = 150x^2 - 136x + 102 \), primarily attributed by a similar growth rate of code size (SLOC), as shown in Figure 29 (left).

![Figure 26: Subject-wise end-to-end regression testing time when compared against running all the test-cases sequentially (red line). RTS\_j corresponds to execution of Mahtab with thread j (Dataset #1).](image)

![Figure 27: Geometric mean time and corresponding speedup, across benchmarks in Figure 26 (Dataset #1).](image)
### 3.6. Threats to Validity

This section discusses limitations of the current architecture of Mahtab and possible threats to our findings.

**External validity:** The reported performance, speed-up, and effectiveness of prioritization may not generalize beyond the subjects in our dataset. The programs in our dataset are a subset of the existing and yet to be developed software artifacts. To mitigate this threat, following the extensive usage in existing research works, we primarily focused on widely used C programs from the SIR repository. Currently, we have evaluated on 12 out of 15 programs made publicly available on SIR website. The rest three programs have issues with build scripts using the make utility. Generalized builds are not yet supported in Mahtab but this can be done by adjusting the implementation a little. As this is more of an implementation issue rather than a technical one, we believe this does not, in principle, provide any technical limitations to the applicability of Mahtab.

**Internal validity:** There might be bugs in implementation of Mahtab and external utilities on top of which our engine is built. Following prior works, we chose Linux diff as a core component in LLVM IR differencing and it may have bugs, but its implementation is beyond our control. To mitigate this threat, we verified that smdiff (our pre-processed textual differencing program) conservatively mimics the output of llvm-diff. During initial stages of developing Mahtab, we actually integrated llvm-diff but after inspecting its inefficiency on generated large codes, we found smdiff to be \( \approx 50\% \) faster with a little loss in precision. We minimized this loss by folding undefined temporaries before IR differencing. For similar reasons of inefficiency with LLVM pass for instrumenting the IR, we wrote sminstru (our static instrumentation program) and utilized it in Mahtab. Implementation of the underlying sorting algorithm may affect test-case orderings in the presence of priority collisions, which may exhibit difference in values of prioritization effectiveness. Our implementations may have missed corner cases and we leave this possible implementation issue for inspection in the near future to reduce this threat.

**Construct validity:** The construction of the experimental setup and assumptions of the platform and execution environment certainly form a subset of all the existing software testing frameworks widely adopted in industry and academia. Regarding the usage of test-suites from SIR repository, we did not sample from the test-pools (universes) to create different test-suites, but instead chose the entire test-pools as a single test-suite for each subject. This remains as our choice of evaluation with respect to test-suite parallelization. We also did not leverage the test-case to fault mapping (available in SIR website) as the goal of our work is to observe failures instead of faults. For single
faults, in case of space and Siemens programs, the fault mapping is straightforward, hence we report their APFD values. For other subjects from SIR (grep, flex, sed, gzip), multiple faults are simultaneously kept active for each faulty version. Although the SIR programs and space are mainly used in controlled experiments, our decision to not leverage the fault information (matrix) is justified as follows. In the real setting, when a software evolves, we only have test-cases and not the \{test-case\}→\{fault\} mapping. While measuring effectiveness of prioritization, the APFD metric requires such a mapping but we chose to treat the first failing test-case as the fault-revealing one. The system-under-test may contain multiple faults activated at the same time. If a test-case fails, the cause of the failure may be due to a single fault or a composite fault, in which case the \{fault\}→\{failure\} mapping is not known and the fault-matrix fails to pin-point exactly which fault has occurred. This directly affects the APFD calculation when there is a collision of multiple faults in a single test-case. A different approach where each failure is treated as a different fault, has been recently used by (Chen et al., 2018). The assumptions of test-set independence, sequentiality of the system-under-test, and that the test-cases are not flaky, do not always occur in practice. We plan to address these limitations in future.

3.7. Test Outcomes: RTS versus RetestAll

For correctness, we compared Mahtab outcomes against those of executing the full test-suite. Specifically, the comparator checks for an exact match in the set of failing test-cases as determined by both retest-all and retest-selected approaches. The order of failures reported turned out to be different for RetestAll and RTS. This is because RetestAll is unprioritized, whereas RTS is not. In all cases, we found that (unordered) sets of reported failures are indeed equal.

4. Related Work

Although regression test selection is a generic approach to optimize regression testing, the fundamental challenge lies in determining the change which varies across programming languages in which software is written (Chen et al., 1994). There exists a multitude of work in the area of software testing and analysis. However, we list a few approaches that resemble our technique. To the best of our knowledge, none of the existing works targets phase-wise parallelization of regression testing.

Test acceleration using GPUs (Rajan et al., 2014; Yaneva et al., 2017): Today’s Graphics Processing Units enable massive parallelism due to its hardware architecture and have also been used for accelerating execution of test-cases. While Rajan et al. (2014) manually transform embedded C programs to CUDA kernels, Yaneva et al. (2017) introduce ParTeCL, which automates this process by producing compiler generated OpenCL code. However, typical testing necessitates processing of files on disk. As the file system support is still in its infancy and the existing approaches for the same are yet to be widely adopted, a very limited (Candido et al., 2017) category of software such as embedded systems and programs with vector and array based data-structure manipulations can be tested. Real-world software consists of file handling, library and system calls, and these features are yet to be incorporated into software running on GPUs.

Using multiple-CPU and multi-core CPUs: Existing literature on test-suite parallelization (Candido et al., 2017) leverages either multiple single-threaded CPUs or multi-core CPUs. There are two kinds of works in this space: parallelization assuming that the test-cases are independent (Misailovic et al., 2007; Briand et al., 2009; Harrold et al., 2001; Nanda et al., 2011; Orso et al., 2004; Zhang et al., 2012; Elbaum et al., 2000; Jiang et al., 2009; Kim and Porter, 2002; Rummel et al., 2005; Srivastava and Thaugurajan, 2002) and detecting test dependencies (Bell et al., 2015; Candido et al., 2017; Gambi et al., 2018; Schwahn et al., 2019) and breaking them (Lam, 2016; Kappler, 2016) to achieve parallelization. Mahtab belongs to the first category. Briand et al. (2009) present regression testing based on Unified Modeling Language (UML) designs for object-oriented software. Harrold et al. (2001) present the first safe regression test selection for Java programs by identifying dangerous (affected) edges of control flow graphs. Jiang et al. (2009) present a family of adaptive random testing (ART) techniques. (Rummel et al., 2005) describes data-flow based regression test prioritization. Kim and Porter (Kim and Porter, 2002) propose history-based test-prioritization for resource-constrained regression testing. Elbaum et al. (2000) report a study on version-specific test-case prioritization. (Nanda et al., 2011) present a methodology for regression testing due to changes in non-code components. (Orso et al., 2004) describe DejaVoo which performs safe, precise, and scalable test selection targeted towards Java software. (Zhang et al., 2012) unifies regression testing and mutation testing, and introduce mutation-specific test-case prioritization. (Misailovic et al., 2007) present Korat for parallel search, test-generation and load-balanced test-execution. Guarnieri et al. (2017) present test-isolation for web applications using test-execution checkpointing. Bell and Kaiser (Bell and Kaiser, 2014) propose unit test virtualization for isolating test-cases. (Gyori et al., 2015) formalize the problem of test pollution and present POLDET which detects test-cases that pollute shared state of heap and file-system. Our parallelization work is complementary to these techniques, and can be combined for improved benefits.

TestTube (Chen et al., 1994): This is a regression test selection approach for testing both deterministic and non-deterministic C programs. TestTube performs a hybrid static-dynamic analysis at the source-code level. The dynamic analysis is used to collect test-coverage, whereas test selection is performed statically. In this respect, this is the closest work to Mahtab which employs a combined dynamic-static-dynamic analysis. However, as mentioned by Celik et al. (2018), there is no existing tool that implements this approach. Compared to TestTube, Mahtab differs by coupling the end-to-end analysis with prioritization and parallelization. TestTube tracks associations of variables, types, functions, and macros present in the source code, with the existing test-suite. On the other hand,
Mahtab tracks only basic-blocks at the IR level associated with each test-case. In addition, Mahtab keeps track of global sections where a change triggers conservative test selection. TestTube has two assumptions: (i) every memory segment must be associated with an identifier, (ii) pointer expressions and arithmetic must be well-bounded. The former assumption is not required in Mahtab. For example, the change: `*(char*)0x5678)=0;`→`*(char*)0x5678)=5;` will be successfully reported by Mahtab in the associated basic-block but not by TestTube. The latter case also does not arise in Mahtab as it does not need to track base variables. This eliminates pointer analysis overhead which, when added to end-to-end time, may be costlier than the RetestAll approach. Changes in pointers and base variables are seamlessly detected by Mahtab. However, the basic-block instrumentation of Mahtab is costlier that the function-level instrumentation of TestTube. From the viewpoint of offline phase (instrumentation and coverage collection), TestTube performs a lightweight but imprecise dynamic analysis.

Mahtab in the space of recent RTS tools for C: In Table 9 we qualitatively compare Mahtab against two most recent approaches, namely Selfection whose code is publicly available, and RTS++, towards regression test selection for C projects. While Mahtab performs a dynamic fine-grained analysis, Selfection and RTS++ perform function coarse-grained static RTS. Selfection is targeted for testing a particular project written in C, namely TizenRT, which compiles to ARM ELF binary. RTS++ is targeted towards testing projects in C/C++ that compile to LLVM IR. Both Selfection and RTS++ assume test-cases to be in the form of test-methods, which are themselves C programs driving the system-under-test. RTS++ also has integration with Google Test framework. However, there exist major architectural differences. While Selfection, and RTS++ do not involve: (i) parallelization, (ii) prioritization, and (iii) handling of pointers and globals, Mahtab supports all of these.

Ekstazi (Gligoric et al., 2015): This is a regression test selection tool for testing Java software. Ekstazi does not calculate the diff (program difference) between two program versions. Instead, it keeps track of dependencies and class checksums as files on disk. In particular, the dependencies have the format: [test class]++[files, external resources] or [test methods]++[files, external resources], and are collected using class loaders, Java agents, reflection, etc. Ekstazi performs dynamic instrumentation of the bytecode and monitors the execution to collect accessed files. Instrumented code points are: (i) start of a constructor, (ii) start of a static initializer, (iii) start of a static method, (iv) access to a static field, (v) use of a class literal, (vi) reflection invocations, and (vii) invocation through the bytecode instruction: invokevirtual. A test-case is selected if the checksum changes for at least one of its dependent files. However, Ekstazi does not perform test-case prioritization. Ekstazi also tests only those frameworks that have JUnit integration.

Echelon (Srivastava and Thiagarajan, 2002): It is a test-case prioritization approach that tracks test coverage and performs program differencing at the granularity of basic blocks. Echelon prioritizes tests based on affected blocks (diff) and performs analysis entirely at the binary level (that is, on the executable) in Windows platform. Echelon has three major architectural differences from Mahtab: (i) it performs binary analysis instead of IR, (ii) its change-analysis uses a successor-predecessor heuristic for determining affected blocks, which is very different from that of Mahtab, and (iii) it employs a variant of the greedy additional heuristic. Echelon is highly platform dependent and results may become unstable once the target hardware changes. Compared to Echelon, Mahtab’s approach is specific only at the LLVM IR and can target different front ends and multiple languages. Therefore, Mahtab can potentially test a family of source languages. Echelon runs all tests (RetestAll), batch-wise in sequence. Earlier batches are more relevant to change; relevance decreases with increase in sequence (batch) number, essentially forming batch-wise prioritization of RetestAll ap-

<table>
<thead>
<tr>
<th>Tool available</th>
<th>Selfection (Celik et al., 2018)</th>
<th>RTS++ (Fu et al., 2019)</th>
<th>Mahtab (this paper)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy for RTS</td>
<td>Static RTS.</td>
<td>Static RTS.</td>
<td>Dynamic RTS.</td>
</tr>
<tr>
<td>Granularity</td>
<td>Function level.</td>
<td>Function level.</td>
<td>Basic-block level.</td>
</tr>
<tr>
<td>Level of analysis, framework used</td>
<td>ARM ELF binary.</td>
<td>LLVM IR and uses Google testing framework.</td>
<td>LLVM IR.</td>
</tr>
<tr>
<td>Targeted towards</td>
<td>TizenRT <a href="https://github.com/samsung/TizenRT">https://github.com/samsung/TizenRT</a>]</td>
<td>C/C++ projects that compile to LLVM IR and has Google Test integration.</td>
<td>C projects that compile to LLVM IR.</td>
</tr>
<tr>
<td>Build system for subject-under-test</td>
<td>Specific build system for TizenRT project.</td>
<td>Generalized build systems Make, CMake, Autowake, etc.</td>
<td>Simplest build approach Single-line compilation using clang, and gcc with appropriate flags.</td>
</tr>
<tr>
<td>Handling of global variables and pointers</td>
<td>No.</td>
<td>No.</td>
<td>Yes.Globals handled conservatively. For pointers, change is detected while performing diff between IRs.</td>
</tr>
<tr>
<td>Involvement of Parallelization</td>
<td>No.</td>
<td>No.</td>
<td>Yes. Proposes parallelization windows.</td>
</tr>
<tr>
<td>Involvement of Prioritization</td>
<td>No.</td>
<td>No.</td>
<td>Yes. Proposes relevance and confinedness. Proposes metrics: (i) ECC, and (ii) EPS.</td>
</tr>
<tr>
<td>Baselines used for comparison</td>
<td>RetestAll.</td>
<td>RetestAll.</td>
<td>Sequential implementation, RetestAll, and greedy additional prioritization.</td>
</tr>
</tbody>
</table>

Table 9: Qualitatively comparing Mahtab with most recent RTS approaches for C.
proach. Echelon is not safe as a program change may not trigger the associated test-case. The authors [Srivastava and Thirarajan, 2002] mention the misprediction rate of 1%-6%.

5. Conclusions and Future Work

We studied the benefit of parallelizing different phases in the workflow of regression testing and demonstrated a methodology for performing platform-independent dynamic/static analysis leveraging LLVM IR. Mahtab’s code-change aware test-prioritization with multi-threaded execution windows improved the failure observation rate. We proposed EPS (fault-blind metric) to measure effectiveness of this combination. ECC (change-aware metric), a variant of APFD (fault-knowledge-aware metric), was introduced for rewarding the effectiveness of code-change coverage. Despite having different design principles, both EPS and ECC address some of the limitations of widely used APFD metric. We showed that SIR programs achieve an end-to-end geometric mean speedup of 4.72×, and EBF of 1.6×, up to 16 threads. We reported that our selected GitHub projects achieve an end-to-end speedup of 3.90× and EBF of 1.43×, up to 32 threads.

A possible future work is to evaluate Mahtab on larger real-world industry projects having real regression faults. It would also be interesting to study evolving test-suites and to explore dynamic dependency graphs. Our globala on mode of analysis conservatively follows the RetestAll approach. Following the design principles of Ekstazi [Gilgoric et al., 2013], and SymDiff [Lahiri et al., 2012], we plan to increase the precision of test selection, when change is detected in global definitions. In general, there may be test-dependencies and load imbalance inside windows when running test-cases in parallel. This can be mitigated by introducing non-uniform windows. For dependencies, only independent test-cases should be accommodated inside a window and then dependent windows executed in sequence. Parallelization windows may be dependent on the nature of the test-ool employed. When targeting load-balancing, load-groups may give rise to different window sizes. A prioritization strategy targeting test-dependencies and load-balancing, can be designed based on parallelization windows.

Acknowledgments

We are thankful to the department of Computer Science and Engineering, IIT Madras, for providing the necessary support and infrastructure for carrying out this research work.

References

doi:10.1145/2950290.2983974