Dynamic Mutation Scheduling: Highly Parallel, Efficient Evaluation of Mutations for Rust Programs through Program Splitting

Zalán Lévai
University of Sheffield

Donghwan Shin
University of Sheffield

Phil McMinn
University of Sheffield

***Abstract*—Mutation analysis has been considered prohibitively expensive for large software projects due to its high computational cost. Even with efficient mutant schemata approaches, the evaluation of the constituent mutants still takes considerable time. Classical mutation analysis systems introduce large overheads by evaluating mutants as separate processes. State-of-the-art Static Mutation Batching uses threads to evaluate batches of multiple mutations at the same time; however, it suffers from a "waiting tail" problem causing idling, often hindering it in practice. Our novel Dynamic Mutation Scheduling approach addresses these limitations and solves the online mutation scheduling problem: maximizing parallelism by safely scheduling new, compatible mutations and their tests as capacity becomes available, resulting in reduced mutation evaluation runtimes. Mutations are compatible with regard to parallel evaluation if they appear in distinct parts of the program. We implement our approach by extending mutest-rs, the state-of-the-art mutation analysis tool for the Rust programming language. We perform an extensive empirical evaluation of our novel Dynamic Mutation Scheduling approach on 25 top subject Rust programs, comparing it to a basic thread-based mutation evaluation approach, and to state-of-the-art Static Mutation Batching. We find that Dynamic Mutation Scheduling effectively reduces wasted time and mutation evaluation runtimes over Static Mutation Batching, by up to 75.6%, while also reducing exhaustive mutation analysis runtimes, by up to 25.2%.**

I. Introduction

Mutation analysis is a technique for evaluating the ability of a program's test suite to detect faults [1], [2], [3], [4], [5]. By generating mutated versions of the program, known as mutants, and running the test suite against each mutant, mutation analysis can determine the test suite's ability to detect potential future program faults that may be introduced by developers. One of the most costly parts of mutation analysis is the evaluation of the mutations and their corresponding test cases. As the program size grows, both the number of test cases and the number of mutations grow as well, with mutation evaluation runtimes scaling with both of these metrics. The improvement of test suites based on the results of mutation analysis is a highly iterative process, requiring the reevaluation of the improved test suite against mutations. Because of this, any improvement in mutation analysis runtimes accumulates across hundreds and thousands of repetitions, and thus has a significant positive impact on the speed of test development. As a notable example, Google performs limited mutation analysis on every commit to their centralized repository [6], [7], [8], which in 2015 received an average of $40{,}000$ commits per workday [9]. For them, a 10 second improvement results in $400{,}000$ fewer seconds (4.63 days) of not just compute, but corresponding developer waiting time per workday.

Due to the general nature of modern computer systems being parallel — growing "*horizontally*" with e.g., increased number of CPU cores, it is natural to try to apply parallelization techniques to mutation analysis, more specifically to the evaluation of mutations. Classical mutation analysis systems use separate processes to evaluate each mutant against each test case; however, this introduces large overheads over thread-based approaches. While the test cases for each mutant can be run on parallel CPU cores without issue, running the test cases for multiple mutants at once can yield even better CPU utilization, especially when mutations have few relevant test cases each. However, constraints must be imposed on any such parallelization of mutants, as they might affect each other's behavior if multiple are enabled without consideration.

To address the limitations of existing parallel mutation evaluation approaches, we propose Dynamic Mutation Scheduling. While state-of-the-art Static Mutation Batching [10], [11] improves efficiency by grouping mutations into static "batches" before evaluating their test cases on threads within a process, it suffers from a "waiting tail" problem caused by straggler test cases. Because Static Mutation Batching cannot react to differences in mutated test runtimes, threads often sit idle while waiting for the slowest tests in a batch to complete, rather than progressing onto other mutations. Dynamic Mutation Scheduling solves the online mutation scheduling problem: maximizing parallel thread utilization by safely inserting new, compatible mutations (and their test cases) into the execution whenever capacity becomes available.

The transition to scheduling mutations dynamically introduces two significant challenges: the need to encode mutation compatibility wrt. parallel evaluation to avoid interactions between mutations within a process, and the development of a scheduling algorithm that can resolve these constraints without significant overheads. We solve these in our novel Dynamic Mutation Scheduling by introducing an efficient encoding of compatible mutations (Section III-B), and a dynamic heuristic scheduling algorithm that maximizes thread and resource utilization throughout the evaluation of mutations (Section III-C).

Zalán Lévai is supported by the EPSRC grant EP/W524360/1. Donghwan Shin is supported by the EPSRC grant EP/Y014219/1. Phil McMinn is supported, in part, by the EPSRC grant "Test FLARE" (EP/X024539/1).

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The efficiency gains can be illustrated with the evaluation sequence produced by each approach: a sequence of tuples for each unit time of the evaluation, each tuple representing the active test cases and mutations at the time. We use the following task set of relevant test (t_i) and mutation (m_i) pairs as an example: $\{(t_1, m_1), (t_2, m_1), (t_1, m_2), (t_3, m_3), (t_4, m_3)\}$, each task taking unit time, on a machine capable of running three tasks in parallel. Sequential mutation evaluation results in the evaluation sequence $\langle (t_1, t_2, m_1), (t_1, m_2), (t_3, t_4, m_3) \rangle$, a 55.5% utilization over three time units. Static Mutation Batching produces $\langle (t_1, t_2, t_3, m_1, m_3), (t_4, m_1, m_3), (t_1, m_2) \rangle$, by batching mutations m_1 and m_3 together, resulting in the same 55.5% utilization over three time units, because m_2 cannot be in the same batch as m_1. Dynamic Mutation Scheduling however produces $\langle (t_1, t_2, t_3, m_1, m_3), (t_4, t_1, m_3, m_2) \rangle$, by inserting (t_1, m_2) as soon as m_1 is completed, resulting in an 83.3% utilization over two time units. In addition, our dynamic approach also reduces mutation scheduling overheads compared to Static Mutation Batching, by only having to check compatibility against the running subset of mutations.

Lévai et al.'s mutest-rs [11], [12] is an established mutation analysis tool for the increasingly used Rust programming language, which also implements Static Mutation Batching. Rust is a memory-safe systems programming language [13] that has seen increased adoption throughout the industry, especially in performance- and safety-critical systems [14]. We implement our novel Dynamic Mutation Scheduling technique by extending mutest-rs, adding it as an alternative mutation execution technique. We perform an extensive empirical evaluation (Section IV) lasting 69,974 CPU minutes over 25 top subject Rust programs totaling 773,655 SLoC to evaluate Dynamic Mutation Scheduling, running mutest-rs with seven different versions of Dynamic Mutation Scheduling, each with different heuristics for mutation selection, comparing it to state-of-the-art Static Mutation Batching, as well as process-based and thread-based sequential mutation evaluation.

The contributions of this paper, therefore, are as follows:

1) A technique and algorithm for dynamically scheduling multiple, compatible mutations for parallel evaluation during mutation evaluation (Section III-C);
2) The necessary efficient embedding of mutation compatibilities in a meta-mutant program (Section III-B);
3) An implementation of Dynamic Mutation Scheduling in the mutest-rs Rust mutation analysis tool;
4) The results of an empirical evaluation (Section IV) of Dynamic Mutation Scheduling against 25 top Rust subject programs, showing that our approach effectively reduces wasted time and the runtime of mutation evaluation, by up to 75.6%, while also reducing the runtime of exhaustive mutation analysis, by up to 25.2%. (Section V).

Our replication package is available online [15].

II. Background

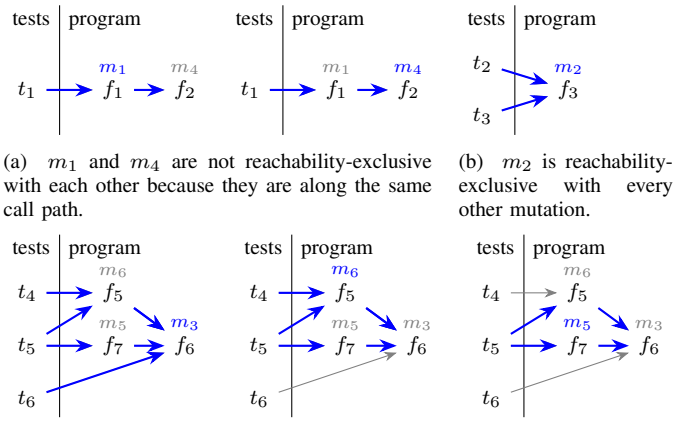
Our Dynamic Mutation Scheduling approach relies on the ability to run every generated mutant program within the same process, to alleviate the overheads of processes for each test execution. This requires embedding every single mutation into a single meta-mutant program, which was first suggested by Untch et al. in their work on mutant schemata approaches [16], [17], [18], and expanded on by Just et al.'s conditional mutation technique [19]. Rather than generating a separate standalone mutant program for each mutation, a single meta-mutant program is generated, where every mutated code path is behind conditional expressions. These conditional expressions can be toggled to activate mutations within the meta-mutant program. Dynamic Mutation Scheduling works using a meta-mutant program, and controls the active mutations within the meta-mutant throughout the evaluation of mutations.

Static Mutation Batching [10], [11] was the first technique to not just evaluate mutations on threads, but also attempt to run multiple mutations and their test cases in parallel, alongside each other. By carefully activating multiple mutations together within a meta-mutant program, ensuring that only mutations that cannot affect the behavior of other mutations were selected, the technique is able to run the test cases corresponding to batches of multiple mutations, not just test cases for a single mutation at a time. This work established the requirements for a set of mutations to be compatible wrt. parallel evaluation through the property of reachability-exclusivity (Section II-A), which can be determined based on a conservative call graph.

The mutest-rs tool [10], [11], [12] is the first and most established tool for mutation analysis of Rust programs. It generates mutations using 18 mutation operators, mostly derived from traditional mutation operators. The mutest-rs tool embeds all mutations into a meta-mutant program, which is controlled by the injected mutation testing harness. To determine the test cases that reach each mutation, mutest-rs creates a complete, conservative call graph of the program, which is ideal for establishing mutation compatibility (Section II-A). The mutest-rs tool implements traditional Static Mutation Batching, making it appropriate for comparing our technique against.

A. Finding Mutation Compatibilities for Parallel Evaluation

To be able to run mutations on parallel threads in a single program we must first establish which mutations can be enabled at the same time without these mutations affecting each other's behavior. This criterion for parallelization must also ensure that the detection of individual mutations remains uniquely identifiable through individual test results, regardless of which "compatible" mutation our Dynamic Mutation Scheduling runs alongside it. Static Mutation Batching established an approach for determining whether a pair of mutations can be run in parallel using the property of reachability-exclusivity [10], [11]. For a set of mutations, reachability-exclusivity is a property that holds iff for each possible walk of all call paths from all active test cases for the set of mutations, at most one active mutation may be encountered. This ensures, amongst any set of reachability-exclusive mutations, both that each mutation retains the exact same behavior as if the mutations were evaluated individually, and that all constituent mutations uniquely correspond to all test cases that may reach them. More formally, this property can be defined as follows:



(a) m_1 and m_4 are not reachability-exclusive with each other because they are along the same call path.

(b) m_2 is reachability-exclusive with every other mutation.

(c) m_3, m_5 and m_6 are not reachability-exclusive with each other because they all share some common call paths.

Fig. 1. Example of reachability-exclusive mutations in parts of a call graph. Functions (f_i) are annotated with the mutations placed into them (m_i). Highlighted edges represent the call paths that reach mutations from their corresponding test cases. Inactive call paths and mutations are dimmed.

Definition 1 (Mutation reachability-exclusivity): Two mutations m_1 and m_2, reachable from the set of tests T_{m_1} and T_{m_2}, respectively, are *reachability-exclusive* iff $T_{m_1} \cap T_{m_2} = \emptyset$.

To determine the traces in the program, mutest-rs builds a static, conservative call graph, starting from the individual test cases as entry point functions [11]. Mutation reachability-exclusivity can be derived from this call graph by tracing all possible call paths from every test case, and recording the mutations found along the way, reducing it to a correspondence map between test cases and the mutations reachable from each of them. Figure 1 shows examples of mutations and their reachability-exclusivity, based on their location along traces of the test case call graph. This can also be used for test case filtering, by only executing the test cases that may reach each individual mutation. This is a common technique in mutation analysis [20]. Because this correspondence map is already used for test case filtering, Dynamic Mutation Scheduling does not introduce any additional overhead over other approaches.

III. Approach

Overview: Our Dynamic Mutation Scheduling approach is based on a traditional meta-mutant mutation analysis pipeline [16], [17], [18], [19]. Such approaches first generate all of the mutations up front, then generate the single meta-mutant program from them, and finally invoke the meta-mutant program multiple times to evaluate each mutation. Our Dynamic Mutation Scheduling technique extends this approach, similarly to traditional Static Mutation Batching. First, our approach adds two additional steps during program analysis; one for building a test suite call graph (Section II-A), and another for determining pairs of mutations that can be run in parallel (Section II-A) and embedding this information into the meta-mutant program (Section III-B). Last, it changes how mutants in the meta-mutant are evaluated (Section III-C).

As such, at a high level, our Dynamic Mutation Scheduling approach performs mutation analysis using a five step pipeline. Figure 2 shows the steps taken by our approach to determine

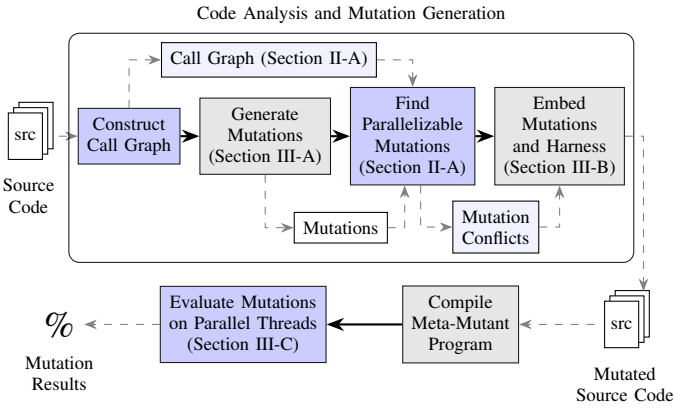


Fig. 2. High-level overview of mutation analysis pipeline showing the dependencies of Dynamic Mutation Scheduling. Highlighted, in shades of blue, are additions over meta-mutant approaches.

compatibilities between mutations for parallelization, and to evaluate program mutations and their test cases in parallel. Given a Rust program, our approach first builds a test suite call graph (Section II-A). The resulting call graph is used both for traditional mutation filtering techniques, and later for determining which mutations can be run in parallel. Second, our approach generates mutations for the program, similarly to other meta-mutant approaches (Section III-A). Third, our approach determines for all pairs of mutations whether they can be evaluated in parallel (Section II-A) based on the test suite call graph it constructed earlier, embedding this information into the meta-mutant program (Section III-B). Following these steps, our approach embeds the generated mutations into a meta-mutant program, alongside our custom mutation testing harness driving the mutation evaluation. Lastly, this meta-mutant program is executed to perform the evaluation of all mutations using our Dynamic Mutation Scheduling rules (Section III-C). Rather than embedding static batches into the meta-mutant program, like with traditional Static Mutation Batching, our approach instead embeds the mutation compatibilities for the following evaluation.

We implement our Dynamic Mutation Scheduling approach as an extension of the mutest-rs [12] Rust mutation analysis tool, which already implements traditional Static Mutation Batching. Our novel Dynamic Mutation Scheduling approach utilizes the existing idea of using a test suite call graph to "split" programs, and to determine whether mutations belong to distinct splits of the program, i.e., whether they can be evaluated in parallel. However, our technique improves upon traditional Static Mutation Batching by dynamically scheduling compatible mutations and their tests *during* mutation evaluation, rather than sequentially evaluating pre-determined, static batches of compatible mutations. This means that our Dynamic Mutation Scheduling approach can "fill the gaps" in the parallel schedule of test cases much more effectively, and at a lower added computational cost.

A. Generating Mutations for Rust Programs

Our approach directly extends the mutest-rs Rust mutation testing tool. As such, it uses the same set of mutation operators

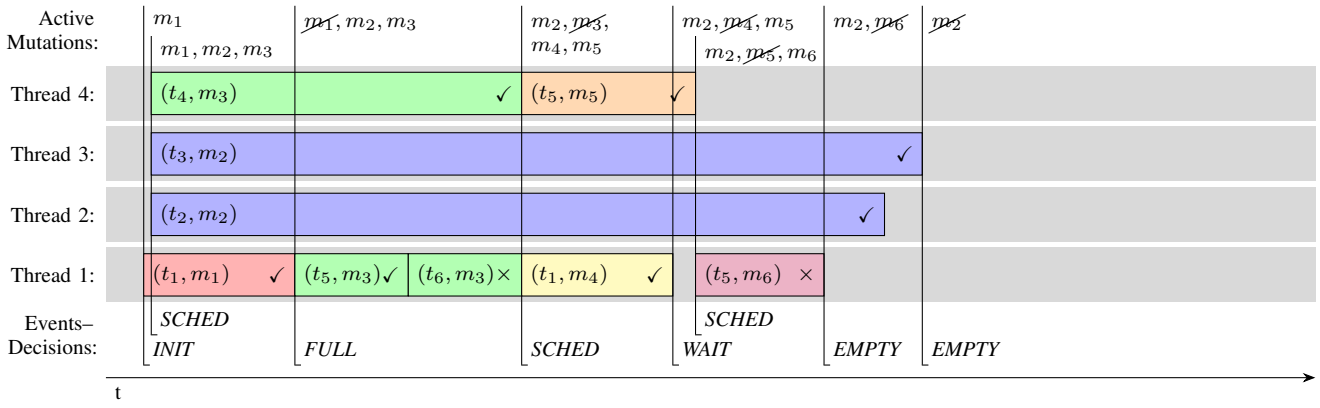


Fig. 3. Example timeline diagram showing how our Dynamic Mutation Scheduling allocates mutations, and schedules test cases. The mutations and tests in this example are as described in Figure 1. Each bar represents a running test in the test-running thread pool, with pass (✓) and fail (×) denoted at the end.

resulting in the same set of generated mutations. We use the complete set of 18 mutation operators mutest-rs supports, to ensure a large variety of mutations in our evaluation.

The mutest-rs tool embeds all generated mutations into a single meta-mutant program using conditional expressions. These conditional expressions check a generated global variable to determine which mutated code paths to execute. This is managed by our approach's modified mutation testing harness throughout the evaluation of mutations, enabling and disabling individual mutated code paths. Such meta-mutants are ideal for implementing our Dynamic Mutation Scheduling technique.

B. Embedding Mutation Compatibility for Parallel Evaluation

Our approach uses the property of mutation reachability-exclusivity directly to establish compatibility of mutations for parallel evaluation (Section II-A). Our Dynamic Mutation Scheduling technique embeds the compatibility of mutations for parallel evaluation into the meta-mutant program pairwise, by listing each compatible pair of mutations in a compressed matrix representation. This representation is also analogous to storing the edges of a graph of the mutations, with each edge representing compatibilities between them. This compatibility matrix representation can be used not just to extract pairwise mutation compatibility at runtime, but also to extract any compatible set of mutations by checking whether all pairs of mutations within the set are compatible with each other.

Dynamic Mutation Scheduling uses this information during mutation evaluation to dynamically enable and disable mutations (and run their test cases). Rather than evaluating statically created batches of mutations, our mutation testing harness utilizes these compatibilities to make decisions about which mutations can be scheduled at various points throughout the evaluation of the mutations. We describe our mutation testing harness and the family of Dynamic Mutation Scheduling algorithms in Section III-C.

C. Scheduling Parallel Mutations throughout Test Evaluation

Once our approach determines the test case traces (Section II-A), program mutations (Section III-A), and mutation compatibilities (Section III-B), the resulting meta-mutant program is executed to perform the evaluation of mutations. This meta-mutant program embeds our mutation testing harness, which drives the evaluation of mutations and their test cases.

Our mutation testing harness begins mutation evaluation by first measuring the runtime of each test case without any mutations applied. This is primarily used to determine individual timeouts for all future evaluations of the corresponding test case. Test case timeouts may occur if a mutation introduces potentially infinite- or long-running code segments. Based on the initial timing measurements T, our approach automatically determines generous timeouts T' for each test case i as follows: $t'_i = t_i + max(0.1 \cdot t_i, 1s)$. If a test case exceeds its timeout during mutation evaluation, our mutation testing harness attempts to terminate the test case in a safe, cooperative manner [21], [22], and the corresponding mutation is marked as detected through a timeout. However, the test case will occupy capacity until its execution reaches a generated cancellation point, such as reentering the mutated code section.

After mutest-rs determines all test case timeouts, our mutation testing harness is able to evaluate mutations and their corresponding test cases in one of three ways; through either sequential evaluation of the mutations — corresponding to classical mutation evaluation techniques, sequential evaluation of mutation batches — corresponding to the traditional Static Mutation Batching approach, or our novel technique of continuously scheduling compatible mutations to be run simultaneously, i.e., Dynamic Mutation Scheduling.

Our mutation testing harness performs the sequential evaluation of mutations by, for each mutation, first enabling the mutation in the meta-mutant program, then running the test cases in parallel on a fixed size thread pool until a test case detects the mutation through a test failure, or until all relevant test cases for the mutation have been evaluated, finally disabling the mutation in the meta-mutant program before moving onto the next mutation. It repeats this until it has evaluated all mutations. This is the simplest approach to using multiple threads in a single meta-mutant process to evaluate mutations. Alternatively, our mutation testing harness also supports evaluating the test cases in isolated processes, which corresponds to classical process-based mutation evaluation.

Mutation batches resulting from traditional Static Mutation Batching are evaluated similarly; however, rather than han-

dling individual mutations one-by-one, our harness handles a sequence of mutation batches. Constituent mutations are enabled in the meta-mutant together, and all relevant test cases are evaluated in parallel. We use the same implementation of Static Mutation Batching as in our previous work [10], [11].

In comparison to both of these existing approaches, our Dynamic Mutation Scheduling technique continuously schedules new, compatible mutations and their test cases, rather than handling mutations in sequence. Dynamic Mutation Scheduling makes use of the same fixed size thread pool as our other threaded approaches; however, scheduling decisions are made while other mutations and their tests are still running. Figure 3 shows an example of our approach's mutation- and test scheduling, with key event–decision pairs highlighted. To start off the schedule, our mutation testing harness first chooses an initial mutation to schedule, and starts running its relevant test cases. This is denoted as *INIT* in Figure 3. Following this initial decision, there are four distinct types of scheduling event–decision pairs that may occur during the evaluation of mutations with Dynamic Mutation Scheduling:

SCHED: A successful scheduling event, whenever capacity in the test-running thread pool becomes available, and there is a compatible mutation to schedule (Section II-A). Upon detecting the availability of capacity in the test thread pool, our mutation testing harness first disables the just finished mutation in the meta-mutant. Then, our mutation testing harness selects a mutation that is compatible with all remaining running mutations. It then activates this mutation in the meta-mutant (in addition to all of the mutations already running), and schedules its test cases to run in the test thread pool.

WAIT: An unsuccessful scheduling event, whenever capacity in the test-running thread pool becomes available, but there are no remaining mutations to schedule that are compatible with all remaining running mutations.

FULL: Whenever the final test run for a mutation (either a test case that detects the mutation or the last relevant test case for the mutation) is completed, the mutation is disabled. When there are further test cases to schedule for the already running mutations, these are scheduled first, before any new mutations are considered. In such cases, it is possible that no capacity in the test-running thread pool becomes available after these additional test cases are scheduled, in which case no new mutations are scheduled by our approach.

EMPTY: The final event, when the final test run for the final mutation is completed, and there are no remaining mutations.

Figure 4 shows the algorithm used to schedule concurrent mutations and their test cases for evaluation under Dynamic Mutation Scheduling. The algorithm effectively performs the entire mutation evaluation process for all program mutations in the meta-mutant program. The Dynamic Mutation Scheduling algorithm takes the set of mutations M to evaluate, a mutation compatibility matrix G_{m_c} — as described in Section III-B, and the maximum number N of concurrent tests to run, i.e., the size of the thread pool in which to evaluate test cases for all concurrent mutations. The algorithm's outer loop is responsible for all concurrent mutation scheduling, and

Require: Set of Mutations M
Mutation Compatiblity Matrix G_{m_c} ▷ *(Section III-B)*
Maximum Number of Concurrent Tests $N \in \mathbb{N}$
Ensure: Mapping $M_{\checkmark}$: Mutation → Mutation Result
Mapping: Mutation → Mutation Result $M_{\checkmark} \leftarrow \{\}$
Set of Threads $P \leftarrow \{t_1, t_2, \ldots, t_N\}$
Set of Mutations $M_{\rightarrow} \leftarrow \{\}$
while $M \neq \{\} \vee M_{\rightarrow} \neq \{\}$ **do** ▷ *Process all remaining mutations.*
  ▷ *Schedule remaining, compatible mutations.* ◁
  while IsConcurrencyAvailable$(P, N) \wedge |M| \geq 1$ **do**
    if Mutation $m' \leftarrow$ CompatibleMutation$(M_{\rightarrow}, G_{m_c}, M)$ **then**
      $M \leftarrow M \setminus \{m'\}$ ▷ *Corresponds to* SCHED.
      $M_{\rightarrow} \leftarrow M_{\rightarrow} \cup \{m'\}$
      EnableMutationInMetaMutantProgram(m')
      ScheduleRelevantTestCasesInBackground(P, m')
    else ▷ *Corresponds to* WAIT.
      break
  if $(m, m_{\checkmark}) \leftarrow$ EvaluatedMutation$(P, M_{\rightarrow})$ **then**
    $M_{\rightarrow} \leftarrow M_{\rightarrow} \setminus \{m\}$
    DisableMutationInMetaMutantProgram(m)
    $M^m_{\checkmark} \leftarrow m_{\checkmark}$

Fig. 4. Algorithm for evaluating mutations on a thread pool with Dynamic Mutation Scheduling. CompatibleMutation is dependent on the mutation selection method used.

the recording of each concurrent mutation's completions and results, until all mutations are evaluated. Inside of this loop, the first inner loop enables one or more additional mutations inside the meta-mutant program (compared to those already enabled with corresponding test cases running), and schedules their relevant test cases for evaluation on the test-running thread pool. The correctness of the concurrent evaluation of mutations comes from the mutation compatibility matrix supplied to the algorithm. The rest of the main loop is responsible for collecting the results of evaluated mutations, and disabling completed mutations in the meta-mutant program.

During Dynamic Mutation Scheduling, the method used to select the next compatible mutation to schedule can have a considerable impact on the resulting schedule. At any given decision point, this is decided based on a mutation selection heuristic function. This acts as an additional quasi-input to the Dynamic Mutation Scheduling algorithm, and determines the behavior of the CompatibleMutation function. The mutation selection heuristic functions we evaluate in our empirical study (Section IV) to find the best option for Dynamic Mutation Scheduling are as follows:

$\mathbb{D}_1$ Select the "first" compatible mutation (according to the order they are generated by mutest-rs, which is in order of source code appearance). *This approach acts as a baseline.*

$\mathbb{D}_\sim$ Select a compatible mutation at random. *This approach acts both as a baseline, and a way of estimating the variance in efficiency amongst the different mutation selection methods.*

$\mathbb{D}_<$ Select the compatible mutation with the lowest number of relevant test cases. *Rationale: Scheduling fewer test cases leaves more capacity in the thread pool to concurrently run test cases for other compatible mutations as well.*

$\mathbb{D}_>$ Select the compatible mutation with the highest number of relevant test cases. *Rationale: Scheduling more test cases maximizes the utilization of the thread pool, even if few remaining mutations are compatible with the chosen mutation.*

$\mathbb{D}_\ll$ Select the compatible mutation whose relevant test cases take the shortest total time to run, based on their un-

mutated runtimes. *Rationale: Running mutations with shorter test cases will free up the thread pool for remaining mutations more quickly than mutations with long-running test cases.*

$\mathbb{D}_{\gg}$ Select the compatible mutation whose relevant test cases take the longest total time to run, based on their unmutated runtimes. *Rationale: Starting the evaluation of test cases that are expected to take a long time early allows the scheduler to run other compatible mutations alongside it, and avoid having to run long-running test cases late, with no remaining compatible mutations to parallelize with.*

$\mathbb{D}_{\times}$ Select the compatible mutation with the lowest "conflictingness" value wrt. all remaining mutations. Conflictingness is the number of conflicts the mutation has with all other remaining mutations. *This approach aims to maximize the number of concurrent mutations by selecting mutations that are likely to be compatible with as many other remaining mutations as possible. This should always keep the pool of options high at any given mutation scheduling decision point.*

Regardless of which technique is used to evaluate the mutations in the meta-mutant program, for any given mutation, our mutation testing harness considers running only the test cases that may reach that mutation, according to mutest-rs's conservative call graph analysis described in Section II-A. This is a common test filtering technique amongst existing mutation analysis methods and tools [20]. As such, our Dynamic Mutation Scheduling technique's reliance on this call graph information does not incur any additional computational costs besides the added step of determining mutation compatibilities, and embedding them in the meta-mutant program, described in Section II-A. In addition, in all cases, our approach does not evaluate any remaining test cases for a given mutation after a corresponding test case already detected the mutation. This is again an already common optimization technique [20].

IV. Evaluation

To evaluate our Dynamic Mutation Scheduling technique, we perform an empirical analysis, comparing it against process-based mutation evaluation, thread-based sequential evaluation of mutations with parallel test cases, and traditional Static Mutation Batching: the state-of-the-art technique that our dynamic approach aims to improve upon. Using our empirical data, we evaluate our novel Dynamic Mutation Scheduling approach using the following three research questions:

***RQ1**: How much is thread utilization increased, and idle time reduced by Dynamic Mutation Scheduling?*
Reason: Investigate exactly how Dynamic Mutation Scheduling parallelizes the work of mutation evaluation compared to other approaches using a wasted time metric (lower is better).

***RQ2**: How much is mutation evaluation runtime reduced by Dynamic Mutation Scheduling?*
Reason: Show how effective Dynamic Mutation Scheduling is at reducing the time it takes to evaluate all mutations.

***RQ3**: How much is the evaluation runtime of exhaustive mutation analysis reduced by Dynamic Mutation Scheduling?*
Reason: Show that exhaustive mutation analysis techniques — used in e.g., test generation, test suite prioritization, fault localization, and mutation subsumption analysis — also benefit from the use of Dynamic Mutation Scheduling, reducing the time to evaluate all reachable test cases for all mutations.

TABLE I
Subject Rust programs used in our empirical analysis.
The *Exh.* column shows the mean percentage of test case evaluations needed for determining mutation detections compared to exhaustive evaluation. The *Crit.* column shows the OpenSSF criticality score of the subject [23], [24].

Subject	SLoC	Tests	Mutations	Exh.	Crit.
aho-corasick	14,063	163	487	42.6%	0.54
alacritty	19,511	79	1,808	96.9%	0.66
alacritty_terminal	10,921	130	2,620	74.2%	0.66
base64	6,678	71	376	97.7%	0.64
bevy_input	4,933	57	337	99.3%	0.70
bevy_math	20,206	276	5,109	94.7%	0.70
bytes	9,111	982	509	40.3%	0.65
chrono	23,934	274	3,239	78.2%	0.69
clap_builder	18,115	77	1,418	89.2%	0.79
gleam-core	163,864	4,285	4,556	5.5%	0.66
hashbrown	14,547	110	291	88.9%	0.67
itertools	17,922	301	560	100.0%	0.66
log	4,646	18	70	100.0%	0.70
polars-core	65,103	126	3,249	95.6%	0.54
pyo3	88,169	718	579	76.4%	0.69
rand	7,801	90	1,024	98.9%	0.76
rand_core	1,120	14	195	100.0%	0.76
regex	4,228	7	9	100.0%	0.72
regex-automata	40,083	134	2,532	79.0%	0.72
regex-syntax	53,942	147	1,916	55.7%	0.72
rustls	66,752	227	1,825	76.4%	0.64
ryu	3,822	34	48	97.3%	0.57
serde_json	20,189	98	559	76.6%	0.73
tokio	87,011	136	320	91.0%	0.79
url	6,984	66	1,753	83.1%	0.72

A. Subjects

To select a representative, wide-ranging sample of Rust programs, we first listed the top 500 Rust crates (Rust's notion for library packages and programs), according to crates.io, Rust's crate registry, in order of "Recent Downloads". In addition, we listed the top 500 GitHub projects with Rust as their primary language, in order of "Most Starred". We combined the two datasets, manually extracting the multiple individual crates that make up each GitHub project. We excluded crates which are Rust-language bindings to external libraries, contain non-executable code, have non-standard tests or make use of a custom test runner (mutest-rs does not support custom tests), or have no unit tests. We then cross-referenced this dataset with the `2022-06-07` OpenSSF criticality score dataset [23], [24], to ensure that our list of subjects represents safety-critical software and components. Finally, we sampled this dataset, resulting in our 25 subjects, while ensuring that programs, test suites, and mutations of various sizes, runtimes, and quantities are represented accordingly.

Table I lists the subjects we use to perform our empirical evaluation, alongside their number of source lines of code (SLoC), number of unit test cases, and the number of mutations mutest-rs's mutation operators generated for them. The *Exh.* column displays the mean percentage of test–mutation evaluations needed for determining mutation detections compared to the total number needed for exhaustive evaluation. We used this to select subjects with many remaining test–mutation

pairs for our discussion in **RQ3**. The *Crit.* column displays the criticality score of the subject from the OpenSSF dataset.

B. Methodology

To perform our empirical evaluation, we built an automated experiment runner, which is responsible for running our modified version of mutest-rs (that implements our Dynamic Mutation Scheduling technique) with various settings corresponding to our compared approaches. It runs each experiment with isolated, exclusive access to identical CPU, RAM and storage resources. It also instructs mutest-rs to output structured data that we analyze as part of our research questions. This includes mutation, test evaluation, and detailed timing information.

We use the mutest-rs [10], [11], [12] Rust program mutation analysis tool for generating and evaluating program mutations. We build our Dynamic Mutation Scheduling approach directly into mutest-rs, by modifying the mutation analysis test harness that mutest-rs uses to evaluate mutations and their test cases in the generated meta-mutant program. We configure our experiment runner to run mutest-rs 10 times (or in the case of $\mathbb{D}_{\sim}$, with 30 different random seeds) with the following mutation evaluation techniques for each subject (described in more detail in Section III-C), with identical sets of mutations:

- $\mathbb{S}_p$ process-based sequential mutation evaluation (one mutation at a time) with parallel test case evaluation;
- $\mathbb{S}_t$ thread-based sequential mutation evaluation (one mutation at a time) with parallel test case evaluation;
- $\mathbb{B}$ traditional Static Mutation Batching [10], [11];
- $\mathbb{D}_{\square}$ Dynamic Mutation Scheduling, with the various heuristics ($_{\square}$ is a placeholder) described in Section III-C.

For each mutation analysis run, mutest-rs records the runtime of the generation and evaluation of mutations separately. The total runtime of mutation evaluation includes the baseline test suite evaluation required to determine test timeouts, and we report on these total runtimes in **RQ2** and **RQ3**, primarily comparing to Static Mutation Batching. We also configure mutest-rs to log test thread allocations during mutation evaluation, which we use to determine the thread utilization percentage metric used in **RQ1**. To provide a comparable metric based on thread utilization across various approaches with varying runtimes, we report a derived *idle time* metric for each approach in **RQ1**. This metric shows the amount of time each mutation evaluation strategy wasted compared to a supposed maximum allocation of resources. The idle time metric T_I for a mutation evaluation with thread utilization percentage ρ, and runtime T is derived as follows: $T_I = (1-\rho) \cdot T$. These metrics are adapted from their queueing theory counterparts [25].

We ran the experiments concurrently on 10 core (20 thread) "slices" of a 64 core AMD Ryzen Threadripper 7980X processor with 32 GiB of RAM allocated to each slice, with the machine running Fedora Server 39. We isolated each 10 core slice of the host computer using jobs managed by the Slurm workload manager. Each job was responsible for running mutest-rs with one of the listed configurations for a particular subject. This setup effectively emulates multiple, identical 10 core computers — representative of common developer systems — at the same time. Within each 10 core slice, we configured mutest-rs to evaluate tests on a thread pool of 16 threads. We built our modified mutest-rs using the `nightly-2025-06-06` version of the Rust compiler.

C. Threats to Validity

We considered and mitigated the threats to the validity of the empirical evaluation of our experiments.

The sets of evaluated mutations are a threat to internal validity. Depending on the set of program mutants used, the runtime and utilization metrics could be different. Because we use an established, state-of-the-art mutation analysis tool, which implements a standard set of mutation operators, we consider the evaluated mutations to be representative.

The choice of subjects is a threat to external validity. The results for a different set of subjects may differ from our findings. To reduce this threat, we used some of the most commonly used and critical Rust projects currently available [23], [24]. Furthermore, we selected subjects that vary in terms of program and test suite size, mutations, and functionality.

Potential faults in our modified version of mutest-rs, our experiment runner program, and our output processing and formatting scripts are a threat to construct validity. We mitigated this threat by extensive testing; we extended mutest-rs's automated test suite to cover our modifications, and manually verified the logs and outputs of our experiment runner program and data processing scripts. Furthermore, we make our extensions to the mutest-rs tool, experimental subjects, experiment runner program, experimental setup configurations, experiment logs and data files, and output processing and formatting scripts available in our replication package [15].

V. Results and Discussion

***RQ1**: How much is overall thread utilization increased with Dynamic Mutation Scheduling?:* The efficiency of parallel work scheduling techniques can be quantified through the use of a utilization metric. Overall thread utilization ρ is the mean of all thread utilization percentages, where each thread's utilization is determined by how much of the overall evaluation time it had test case execution work allocated to it. While overall thread utilization is not comparable across mutation evaluations because of differences in runtimes, we can directly compare the resulting wasted time T_I from each approach. Table II shows mean wasted time T_I in seconds for each evaluated approach across all subjects (Section IV), with the exception of process-based mutation evaluation ($\mathbb{S}_p$).

Comparing our Dynamic Mutation Scheduling techniques to traditional Static Mutation Batching ($\mathbb{B}$), we see a substantial decrease in wasted time in the case of 12 out of the 25 subjects. We only see an increase in wasted time in the case of one subject: `bevy_math`. This can be attributed to its combination of an especially high mutation count and instantaneous test cases, which results in the outlier result of the scheduling time exceeding the test execution time. Amongst all remaining subjects, the amount of wasted time remains similar to traditional Static Mutation Batching ($\mathbb{B}$), with the majority of these being

TABLE II

MEAN WASTED TIME (T_I), IN SECONDS, DURING MUTATION EVALUATION. Symbols represent techniques, as defined in Section IV-B. Amongst Dynamic Mutation Scheduling configurations, cells are color-coded according to the relative difference compared to $\mathbb{B}$, with darker blues showing more improved and darker reds showing more increased.

Subject	$\mathbb{S}_t$	$\mathbb{B}$	$\mathbb{D}_1$	$\mathbb{D}_\sim$	$\mathbb{D}_<$	$\mathbb{D}_>$	$\mathbb{D}_\ll$	$\mathbb{D}_\gg$	$\mathbb{D}_\times$
aho-corasick	5.5	6.9	4.9	5.2	4.5	5.0	4.7	5.1	5.1
alacritty	6.8	10.8	3.3	4.3	3.8	3.3	3.4	4.6	3.5
alacritty_terminal	175	204	155	157	156	155	156	155	156
base64	7.2	8.8	5.9	6.1	5.9	5.6	5.9	5.8	5.7
bevy_input	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bevy_math	0.6	0.5	0.7	2.3	0.7	0.7	1.6	1.5	9.6
bytes	2.2	2.9	1.1	1.1	1.1	1.1	3.1	3.1	1.1
chrono	22.3	22.1	21.6	21.7	21.8	20.8	23.8	23.6	23.7
clap_builder	0.2	0.3	0.3	0.3	0.3	0.3	0.4	0.4	0.4
gleam-core	12.1	79.6	4.6	4.0	5.1	4.3	*t/o*	*t/o*	5.9
hashbrown	0.2	1.3	0.2	0.6	0.5	0.4	0.7	0.4	0.3
itertools	0.1	0.0	0.0	0.1	0.0	0.0	0.1	0.1	0.1
log	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
polars-core	1.1	1.0	0.9	1.3	0.9	1.0	1.3	1.3	2.4
pyo3	0.1	0.1	0.0	0.1	0.0	0.0	0.2	0.2	0.1
rand	21.5	23.1	3.6	3.1	3.8	1.2	3.5	3.0	4.0
rand_core	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
regex	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
regex-automata	2.2	0.7	2.3	2.4	2.4	0.6	2.6	0.7	2.6
regex-syntax	22.2	22.4	13.7	14.7	15.0	13.8	15.0	13.9	13.8
rustls	15.8	17.5	7.7	8.0	8.3	8.2	9.5	20.5	8.6
ryu	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
serde_json	0.2	0.8	0.2	0.2	0.2	0.2	0.3	0.3	0.2
tokio	31.5	31.3	30.5	17.0	22.8	21.0	17.9	27.2	27.3
url	2.6	2.6	2.6	2.7	2.7	2.7	2.8	2.8	2.8

small, $\sim$ 1 second mutation evaluations. The subjects where Dynamic Mutation Scheduling decreases relative wasted time the most are `rand` (-97.3%), `gleam-core` (-95.4%), `hashbrown` (-86.2%), `serde_json` (-79.1%), and `pyo3` (-76.8%). Comparing the various selection heuristics of Dynamic Mutation Scheduling, selecting the mutation with the most tests ($\mathbb{D}_>$) achieves the highest median reduction in wasted time across all subjects, by -30.2%, followed by selecting the "first" mutation ($\mathbb{D}_1$) at -23.5%.

It is worth noting that neither overall thread utilization ρ reaches 100%, nor does wasted time T_I reach zero seconds. While these are theoretically possible, all mutation parallelization approaches are limited by the mutation compatibility constraints of the subject program, which is determined by how tightly coupled parts of the subject program are, and how large the scope of individual test cases in the test suite are.

In conclusion for RQ1, Dynamic Mutation Scheduling effectively increases thread utilization, and decreases wasted time over both state-of-the-art Static Mutation Batching and thread-based sequential evaluation of mutations in the majority of subjects, by up to 97.3%. Dynamic Mutation Scheduling with selecting the mutation with the most test cases ($\mathbb{D}_>$) shows the highest reductions in wasted time overall.

***RQ2**: How much is mutation evaluation runtime reduced with Dynamic Mutation Scheduling?:* Because Dynamic Mutation Scheduling focuses on mutation evaluation, and because meta-mutant compilation times are unaffected, we compare total mutation evaluation runtimes directly, inclusive of the baseline test suite evaluation required to determine test timeouts. First, we performed statistical analysis on the distributions of measured runtimes. For each subject, we applied the Wilcoxon rank-sum test [26], [27] individually to the underlying distributions of measured runtimes of each configuration, making pairwise comparisons between the baseline Static Mutation Batching ($\mathbb{B}$) and the various Dynamic Mutation Scheduling configurations. The resulting p-values are < 0.01 in the case of 20 out of 25 subjects, showing that the distributions are highly distinct from each other. The remaining 5 cases are comparisons where the runtime was not improved. We also computed the Vargha-Delaney A_{12} [28] effect size in the same manner, and found that 19 out of the 25 subjects' effect sizes were extremes of either > 0.92 or < 0.02, suggesting that there is a clear delineation between runtimes from the distributions of distinct configurations. We found rare outliers in the same cases as above. From these statistical analyses, we can determine that the underlying distributions of measured runtimes are highly separated between groups and highly concentrated within groups, and therefore comparing the means of the distributions for the rest of this discussion is adequate. Table III shows mean time taken, in seconds, to evaluate mutations with each evaluated approach across all subjects (Section IV). Table III also shows "*substantial*" relative changes in runtimes compared to the baseline traditional Static Mutation Batching ($\mathbb{B}$). We consider a difference in runtime "substantial" if the change is at least 0.1 seconds.

Comparing our Dynamic Mutation Scheduling technique to traditional Static Mutation Batching ($\mathbb{B}$), amongst the 25 subjects, 15 subjects have mutation evaluation runtimes over one second. We see substantial relative reductions in runtime across these subjects, with the most substantial being in the case of `rand` (-66.4%), `bytes` (-58.5%), `alacritty` (-54.4%), `rustls` (-46.5%), and `hashbrown` (-40.1%). Amongst the 11 subjects with runtimes less than two seconds, Dynamic Mutation Scheduling generally performs comparably to the baseline traditional Static Mutation Batching ($\mathbb{B}$).

The outlier results for all Dynamic Mutation Scheduling configurations of `aho-corasick`, and `rand` with $\mathbb{D}_<$are caused by variances in the exact test cases evaluated by different mutation evaluation configurations, and the resulting changes in how much test timeouts occupy the test thread pool. By calculating the total time impact of the test timeouts across the thread pool as $\sum T_{t/o} \div N$, where $N = 16$ is the thread pool size, we find that `aho-corasick` with $\mathbb{D}_1$has a test timeout impact of 45.9 seconds out of the 63.4 seconds total evaluation time, compared to $\mathbb{B}$'s test timeout impact of 22.8 seconds out of the 43.7 seconds total evaluation time, with other Dynamic Mutation Scheduling configurations following a similar trend. Similarly, `rand` with $\mathbb{D}_<$has a test timeout impact of an outlier 166.8 seconds compared to every other configuration, such as $\mathbb{B}$'s 8.5 seconds. Some other subjects are also affected by test timeouts, namely `tokio`, and `alacritty_terminal`. It is worth noting that variance in the behavior of timed out tests may affect any thread-based technique, including traditional Static Mutation Batching, because terminating threaded work safely has no general solution, and instead requires a cooperative approach that works with any generic test [21], [22].

TABLE III

MEAN TIME TAKEN, IN SECONDS, TO EVALUATE MUTATIONS WITH VARIOUS APPROACHES.

Subjects ordered by absolute difference of $\mathbb{D}_1$ compared to $\mathbb{B}$, starting from most improved. The $\pm\%$ columns show the relative difference compared to $\mathbb{B}$, as a percentage, wherever substantial. Symbols represent techniques, as defined in Section IV-B. Amongst Dynamic Mutation Scheduling configurations, cells are color-coded according to the relative difference compared to $\mathbb{B}$, with darker blues showing more improved and darker reds showing more increased.

Subject	$\mathbb{S}_p$	$\mathbb{S}_t$	$\mathbb{B}$	$\mathbb{D}_1$		$\mathbb{D}_\sim$		$\mathbb{D}_<$		$\mathbb{D}_>$		$\mathbb{D}_\ll$		$\mathbb{D}_\gg$		$\mathbb{D}_\times$	
	s	s	s	s	±%	s	±%	s	±%	s	±%	s	±%	s	±%	s	±%
`gleam-core`	*t/o*	154.2	195.2	141.0	-27.8%	140.4	-28.1%	142.9	-26.8%	141.9	-27.3%	*t/o*		*t/o*		143.6	-26.5%
`alacritty_terminal`[1]	352.7	204.0	228.2	181.6	-20.4%	181.1	-20.7%	182.3	-20.1%	182.7	-19.9%	186.2	-18.4%	182.7	-20.0%	183.0	-19.8%
`regex-syntax`	192.7	29.3	30.9	20.7	-32.9%	21.2	-31.4%	22.2	-28.2%	20.7	-33.1%	22.1	-28.5%	21.0	-32.3%	20.9	-32.4%
`rustls`	171.5	19.0	20.6	11.0	-46.5%	11.6	-43.6%	11.8	-42.6%	11.5	-44.0%	13.4	-34.8%	25.7	+25.0%	11.9	-42.1%
`rand`[1]	50.9	30.7	32.3	23.5	-27.2%	12.7	-60.7%	167.4	+418.8%	11.0	-66.1%	11.6	-64.0%	11.4	-64.6%	10.8	-66.4%
`alacritty`	64.1	11.8	13.4	6.1	-54.4%	6.1	-54.4%	6.1	-54.4%	6.1	-54.3%	6.2	-53.7%	6.2	-53.9%	6.2	-54.0%
`tokio`[1]	11.3	51.2	52.9	49.9	-5.8%	50.4	-4.8%	48.9	-7.6%	49.1	-7.2%	51.7	-2.4%	47.4	-10.4%	49.4	-6.7%
`base64`	20.8	9.4	10.8	7.8	-27.8%	7.8	-27.1%	7.7	-28.3%	7.6	-29.7%	7.7	-28.2%	7.6	-29.4%	7.6	-29.6%
`bytes`	15.2	2.4	3.1	1.3	-57.4%	1.3	-57.9%	1.3	-58.2%	1.3	-58.5%	3.8	+20.8%	3.8	+20.5%	1.4	-56.3%
`regex-automata`	156.5	2.9	3.9	3.0	-23.8%	3.2	-17.2%	2.8	-28.3%	2.9	-24.3%	3.0	-23.0%	3.0	-21.8%	4.5	+15.5%
`chrono`	235.0	27.3	27.1	26.4	-2.7%	26.3	-3.0%	26.6	-1.9%	25.6	-5.5%	29.0	+7.2%	28.8	+6.2%	28.6	+5.6%
`serde_json`	17.9	0.2	0.8	0.2	-74.0%	0.2	-73.4%	0.2	-74.7%	0.2	-73.5%	0.4	-55.7%	0.4	-56.5%	0.2	-72.1%
`hashbrown`	7.5	1.1	1.4	1.1	-20.8%	0.8	-40.1%	1.3		1.1	-18.1%	0.9	-36.8%	1.4		1.1	-21.0%
`polars-core`	145.4	1.3	1.2	1.2		1.5	+27.2%	1.1		1.2		1.6	+29.6%	1.6	+31.1%	2.6	+117.9%
`pyo3`	16.5	0.1	0.1	0.1		0.1		0.1		0.1		0.5	+286.1%	0.5	+290.8%	0.1	
`itertools`	10.4	0.1	0.1	0.0		0.1		0.0		0.0		0.1		0.1		0.1	
`bevy_input`	6.0	0.0	0.0	0.0		0.0		0.0		0.0		0.0		0.0		0.0	
`ryu`	1.1	0.0	0.0	0.0		0.0		0.0		0.0		0.0		0.0		0.0	
`regex`	0.2	0.0	0.0	0.0		0.0		0.0		0.0		0.0		0.0		0.0	
`log`	1.2	0.0	0.0	0.0		0.0		0.0		0.0		0.0		0.0		0.0	
`rand_core`	3.3	0.0	0.0	0.0		0.0		0.0		0.0		0.0		0.0		0.0	
`clap_builder`	61.0	0.2	0.3	0.3		0.4		0.4		0.3		0.4		0.4		0.4	
`url`	102.1	3.7	3.7	3.7		3.9	+5.5%	3.7		3.7		4.0	+7.8%	4.0	+7.7%	3.9	+5.9%
`bevy_math`	247.1	0.6	0.6	0.7		2.4	+327.3%	0.7		0.7		1.7	+199.3%	1.6	+191.1%	9.7	+1639.5%
`aho-corasick`[1]	48.1	67.3	43.7	63.4	+45.0%	52.3	+19.8%	71.1	+62.6%	57.0	+30.3%	67.4	+54.2%	47.4	+8.5%	63.6	+45.5%

[1] Runtimes include substantial, varying test timeouts that may negatively impact the performance of thread-based mutation evaluations.

The rest of the increased runtimes with some of the Dynamic Mutation Scheduling configurations are amongst subjects with very low, under two second long mutation evaluation runtimes. In these cases, certain, more expensive to compute selection heuristics for Dynamic Mutation Scheduling ($\mathbb{D}_\ll$, $\mathbb{D}_\gg$, $\mathbb{D}_\times$) take longer than the tests themselves, thus increasing mutation evaluation runtimes. The case of `bevy_math` is the most extreme example of this, and exemplifies why selection heuristics that are fast to compute should be preferred, at least for subjects with already short mutation evaluation runtimes.

In the *worst case scenario* of only `aho-corasick`, Dynamic Mutation Scheduling may evaluate additional timed out tests, depending on the exact schedule, and thus extend mutation evaluation runtimes by a largely variable amount. The same risk exists with other thread-based mutation evaluation approaches. In the *average case scenario* of smaller subjects with ~ 1 second long mutation evaluation runtimes, such as `serde_json` or `hashbrown`, Dynamic Mutation Scheduling can reduce mutation evaluation runtimes over Static Mutation Batching by 0.29 to 0.63 seconds, or by 20.8% to 74.0%. Across a very conservative hundred checks — which could be realistic for a test developer in a workday, this improvement accumulates to 29 to 63 seconds. In the *best case scenario*, such as `gleam-core`, Dynamic Mutation Scheduling can reduce mutation evaluation runtimes over Static Mutation Batching by 54.2 seconds, or by 27.8%. Across the same conservative hundred checks in a test developer's workday, this improvement accumulates to 542 seconds (or nine minutes), and to 45.2 minutes over a five day workweek.

Comparing our Dynamic Mutation Scheduling technique to process-based sequential evaluation of mutations ($\mathbb{S}_p$), we generally see mutation evaluation runtime reductions in the orders of magnitude, with the exceptions of `aho-corasick` due to the aforementioned test timeouts, and `tokio` — an asynchronous runtime that spawns its own threads during test execution. The additional overhead of processes is especially apparent in the case of `bevy_math`, which has 5,109 mutations and 276 short-running, millisecond-long test cases.

Across all subjects, Dynamic Mutation Scheduling using the heuristic of selecting the mutation with the most tests ($\mathbb{D}_>$), and Dynamic Mutation Scheduling using the heuristic of selecting the first mutation ($\mathbb{D}_1$) produce the fastest mutation evaluations, with a median of -19.0% and a maximum of -75.6%, and a median of -15.7% and a maximum of -74.7% in reductions respectively. The generally worse improvements in runtimes with more complex heuristics ($\mathbb{D}_\ll$, $\mathbb{D}_\gg$, $\mathbb{D}_\times$) indicates that selection functions for Dynamic Mutation Scheduling must remain very fast to compute, especially if the subject has a large number of mutations with fast-running test cases.

In conclusion for RQ2, mutation evaluation with Dynamic Mutation Scheduling performs substantially faster amongst larger subjects, by up to 75.6%, and comparably on very small subjects. Dynamic Mutation Scheduling with selecting the mutation with the most test cases ($\mathbb{D}_>$) shows the highest reductions in runtimes overall, and because it improves runtimes over Static Mutation Batching in all except one case, we recommend its use for all cases, regardless of subject size.

TABLE IV

MEAN TIME, IN SECONDS, TO EXHAUSTIVELY EVALUATE MUTATIONS. The ±% columns show the relative difference compared to $\mathbb{B}$, as a percentage. Symbols represent techniques, as defined in Section IV-B. Amongst Dynamic Mutation Scheduling configurations, cells are color-coded according to the relative difference compared to $\mathbb{B}$, with darker blues showing more improved.

Subject	$\mathbb{S}_p$	$\mathbb{S}_t$	$\mathbb{B}$	$\mathbb{D}_1$		$\mathbb{D}_>$	
	s	s	s	s	±%	s	±%
`aho-corasick`	128.7	75.8	75.1	74.1	-1.3%	74.2	-1.1%
`bytes`	55.6	17.4	13.8	12.0	-12.6%	11.8	-14.5%
`chrono`	341.7	86.8	85.9	84.4	-1.8%	84.9	-1.1%
`regex-syntax`	323.5	92.7	92.0	80.7	-12.2%	81.2	-11.8%
`rustls`	480.7	47.7	44.6	33.3	-25.2%	33.4	-25.0%

***RQ3**: How much is exhaustive mutation evaluation runtime reduced with Dynamic Mutation Scheduling?:* While the detections of individual mutations can be determined from the mutation evaluation in **RQ1** and **RQ2**, it is sometimes necessary or more informative to perform exhaustive mutation evaluation; evaluating *all* test–mutation pairs that have possible traces between them. For this evaluation, we select the five subjects with the lowest proportion of test–mutation pairs evaluated — excluding `gleam-core` due to its prohibitively long runtime for this experiment, and perform exhaustive mutation analysis with the baseline approaches ($\mathbb{S}_p$, $\mathbb{S}_t$, $\mathbb{B}$), and the top two best performing Dynamic Mutation Scheduling variants in **RQ2** ($\mathbb{D}_>$, $\mathbb{D}_1$). Table IV shows mean time taken, in seconds, to exhaustively evaluate mutations with each approach across the selected subjects, alongside relative changes in runtimes compared to baseline traditional Static Mutation Batching ($\mathbb{B}$).

We can see that, overall, exhaustive mutation evaluation runtime reductions compared to Static Mutation Batching are smaller than with non-exhaustive evaluation. For example, `rustls`'s -44.0% non-exhaustive improvement levels out to -25.0% after exhaustive evaluation with Dynamic Mutation Scheduling. We see similar trends with the substantially improved `bytes` and `regex-syntax`, and the slightly improved `chrono` and `aho-corasick`. Subjects with configurations that performed similarly in **RQ2** retain similar runtimes to each other after exhaustive mutation evaluation. It is worth noting that the potential impact of test timeouts occupying the test thread pool should be minimized with exhaustive evaluation, as all timed out tests are evaluated in all cases, although in different orders. The two evaluated Dynamic Mutation Scheduling variants perform almost identically to each other, similar to how they performed with non-exhaustive mutation analysis in **RQ2**. The overall runtimes increase more than the "exhaustiveness" of non-exhaustive mutation evaluations in Table I would suggest, with e.g., `bytes`' 40.3% exhaustive mutation analysis baseline runs in **RQ2** only taking 3.1 seconds, and its fully exhaustive mutation analysis baseline runs taking 13.8 seconds; much larger than the approximate ($\frac{1}{40.3\%} \approx$) 2.5 times increase that might be expected.

In conclusion for RQ3, exhaustive mutation evaluation times are substantially longer across all configurations. The relative rate of improvement between approaches remains almost identical with exhaustive evaluation. Dynamic Mutation Scheduling can decrease the time taken by up to 25.2%.

VI. RELATED WORK

A number of techniques exist in research for reducing the cost of mutation analysis. Such techniques are covered in the survey by Pizzoleto et al. [20]. Works on cost reduction include techniques that seek to reduce the cost of evaluating mutations, rather than their numbers, similar to this paper's approach.

The work of Zhang et al. [29] used code coverage and execution history to prioritize test cases during mutation evaluation. In comparison, our approach prioritizes test cases based on their original execution times. The work of Schuler and Zeller [30], and of Mateo and Usaola [31] applied statement coverage analysis to determine which test cases reached which mutations. Just et al. [32] built on these ideas using state infection. Unlike these techniques, our approach uses the static call graph mutest-rs already builds for this, filtering tests without any additional overhead or analysis required.

There exists multiple approaches to process-based parallelism of mutation evaluation, the first of which was the "split-stream" execution method proposed by King and Offutt [33]. They sought to exploit the common code paths between mutations and the original program, only executing common code once. Tokumoto et al. [34] implemented split-stream execution of bytecode mutants for C programs. Gopinath et al. [35], Sun et al. [36], and Vercammen et al. [37] combined mutations close together, forking a meta-mutant process at each point of divergence. All of these approaches rely on the costly forking of an instrumented program. Instead, our approach tackles the problem fundamentally differently, by using a single process, lightweight threads within, a fast call graph technique to determine compatibility, and our novel dynamic scheduling of mutations throughout the evaluation of mutations.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we present our novel Dynamic Mutation Scheduling approach for dynamically scheduling multiple mutations in parallel throughout mutation evaluation. Our algorithm activates new mutations as soon as capacity for new test cases is available, thereby reducing wasted time and increasing the utilization of modern, parallel computer systems. We perform an empirical evaluation, comparing seven different heuristic functions for Dynamic Mutation Scheduling. The results show that our approach, using a heuristic that selects the mutation with the most relevant test cases ($\mathbb{D}_>$), is overall the most efficient, achieving up to a 75.6% reduction in mutation evaluation runtimes. Because our approach, for all except one subject, either improves runtimes substantially, or achieves performance comparable to traditional Static Mutation Batching, we recommend its use on all subjects, both for exhaustive and non-exhaustive mutation evaluations.

The concepts behind our approach are universal across programming languages, and we consider exploring its application to other languages. To further investigate the effectiveness of our technique, future work could examine how it scales with the number of parallel threads, and to explore less conservative criteria for determining mutation compatibility in parallel evaluation, e.g., by using dataflow analysis.

References

[1] R. J. Lipton, "Fault diagnosis of computer programs," Carnegie Mellon Univ., Tech. Rep, Student, 1971.

[2] R. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1702444

[3] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1646911

[4] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation Analysis." Defense Technical Information Center, Fort Belvoir, VA, USA, Tech. Rep., Sep. 1979. [Online]. Available: http://www.dtic.mil/docs/citations/ADA076575

[5] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5487526

[6] G. Petrović, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just, "An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2018, pp. 47–53. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8411730

[7] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 163–171. [Online]. Available: https://dl.acm.org/doi/10.1145/3183519.3183521

[8] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical Mutation Testing at Scale: A view from Google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, Oct. 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9524503

[9] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2854146

[10] Z. Lévai and P. McMinn, "Batching Non-Conflicting Mutations for Efficient, Safe, Parallel Mutation Analysis in Rust," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Dublin, Ireland: IEEE, Apr. 2023, pp. 49–59. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10132214

[11] Z. Lévai, D. Shin, and P. McMinn, "A Comprehensive Empirical and Theoretical Analysis of Batching Algorithms for Efficient, Safe, Parallel Mutation Analysis in Rust," *ACM Transactions on Software Engineering and Methodology*, Jan. 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3787851

[12] Z. Lévai, "mutest-rs," 2026. [Online]. Available: https://github.com/zalanlevai/mutest-rs

[13] The Rust Project Developers, "The Rust programming language," 2026. [Online]. Available: https://www.rust-lang.org

[14] ——, "Rust: Production users," 2024. [Online]. Available: https://www.rust-lang.org/production/users

[15] Z. Lévai, "Replication package," Mar. 2026. [Online]. Available: https://zenodo.org/records/18893405

[16] R. H. Untch, "Mutation-based software testing using program schemata," in *Proceedings of the 30th Annual Southeast Regional Conference*, ser. ACM-SE 30. New York, NY, USA: Association for Computing Machinery, Apr. 1992, pp. 285–291. [Online]. Available: https://dl.acm.org/doi/10.1145/503720.503749

[17] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '93. New York, NY, USA: Association for Computing Machinery, Jul. 1993, pp. 139–148. [Online]. Available: https://dl.acm.org/doi/10.1145/154183.154265

[18] R. H. Untch, "Schema-based mutation analysis: A new test data adequacy assessment method," Ph.D. dissertation, Clemson University, Clemson, SC, USA, Dec. 1995. [Online]. Available: https://www.proquest.com/docview/304173339/abstract/F975B52F62924781PQ/1

[19] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 50–56. [Online]. Available: https://dl.acm.org/doi/10.1145/1982595.1982606

[20] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, Nov. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301554

[21] A. Kolesnichenko, S. Nanz, and B. Meyer, "How to Cancel a Task," in *Multicore Software Engineering, Performance, and Tools*, J. M. Lourenço and E. Farchi, Eds. St. Petersburg, Russia: Springer, Aug. 2013, pp. 61–72.

[22] U. Sethi, H. Pan, S. Lu, M. Musuvathi, and S. Nath, "Cancellation in Systems: An Empirical Study of Task Cancellation Patterns and Failures," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA, USA: USENIX Association, Jul. 2022, pp. 127–141. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/sethi

[23] The Open Source Security Foundation, "Criticality Score," Jun. 2022. [Online]. Available: https://commondatastorage.googleapis.com/ossf-criticality-score/index.html

[24] A. Arya, C. Brown, R. Pike, and The Open Source Security Foundation, "Open Source Project Criticality Score," The Open Source Security Foundation, Mar. 2023. [Online]. Available: https://github.com/ossf/criticality_score

[25] A. Ivo and R. Jacques, *Queueing Theory*. Eindhoven, Netherlands: Eindhoven University of Technology, Feb. 2002. [Online]. Available: https://iadan.win.tue.nl/blockq/queueing.pdf

[26] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: https://www.jstor.org/stable/3001968

[27] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. [Online]. Available: https://www.jstor.org/stable/2236101

[28] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, Jun. 2000. [Online]. Available: https://doi.org/10.3102/10769986025002101

[29] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, Jul. 2013, pp. 235–245. [Online]. Available: https://dl.acm.org/doi/10.1145/2483760.2483782

[30] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for Java," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, Aug. 2009, pp. 297–298. [Online]. Available: https://dl.acm.org/doi/10.1145/1595696.1595750

[31] P. R. Mateo and M. P. Usaola, "Reducing mutation costs through uncovered mutants," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 464–489, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1534

[32] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 315–326. [Online]. Available: https://dl.acm.org/doi/10.1145/2610384.2610388

[33] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.4380210704

[34] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden, "MuVM: Higher Order Mutation Analysis Virtual Machine for C," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Chicago, IL, USA: IEEE, Apr. 2016, pp. 320–

329. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7515483

[35] R. Gopinath, C. Jensen, and A. Groce, "Topsy-Turvy: A smarter and faster parallelization of mutation analysis," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 740–743. [Online]. Available: https://dl.acm.org/doi/10.1145/2889160.2892655

[36] C.-a. Sun, J. Jia, H. Liu, and X. Zhang, "A Lightweight Program Dependence Based Approach to Concurrent Mutation Analysis," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01. Tokyo, Japan: IEEE, Jul. 2018, pp. 116–125. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8377647

[37] S. Vercammen, S. Demeyer, M. Borg, N. Pettersson, and G. Hedin, "Mutation testing optimisations using the Clang front-end," *Software Testing, Verification and Reliability*, vol. 34, no. 1, p. e1865, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1865