Parallelization in System-level Testing: Novel Approaches to Manage Test Suite Dependencies

Pasquale Polverino, Fabio Di Lauro, Matteo Biagiola, Paolo Tonella *Member, IEEE Computer Society*, and Antonio Carzaniga,

Abstract—System-level testing is fundamental to ensure the reliability of software systems. However, the execution time for system tests can be quite long, sometimes prohibitively long, especially in a regimen of continuous integration and deployment. One way to speed things up is to run the tests in parallel, provided that the execution schedule respects any dependency between tests. We present two novel approaches to detect dependencies in system-level tests, namely PFAST and MEM-FAST, which are highly parallelizable and optimistically run test schedules to exclude many dependencies when there are no failures. We evaluated our approaches both asymptotically and practically, on six Web applications and their system-level test suites, as well as on MySQL system-level tests. Our results show that, in general, PFAST is significantly faster than the state-of-the-art PRADET dependency detection algorithm, while producing parallelizable schedules that achieve a significant reduction in the overall test suite execution time.

Index Terms—System-level testing, Test dependencies

1 INTRODUCTION

System-level testing is fundamentally important but also time-consuming. System-level test suites can run for hours or even days [1], and these long execution times can become a critical limitation especially within a regimen of continuous integration and deployment. For instance, Dobslaw et al. [1] studied test selection using industrial test suites developed at Huawei Cloud Computing Technologies whose sizes range from 3k to 11k test cases. Fortunately, it is possible to allocate virtually unlimited computing resources at a relatively low cost to run the tests in parallel. However, the parallel scheduling of the tests, or the selection of a subset of the tests, must be consistent with whatever semantic dependencies might exist between tests. In principle, one could rely on explicitly declared dependencies. In practice, such specifications are rare; even when they exist, they might be inconsistent with the tests. Our goal, then, is to automate and optimize the parallel execution of system-level tests by detecting dependencies and therefore deriving consistent test schedules from the test suite itself. This is not a new problem. Other researchers have developed ways to detect dependencies, primarily in unit tests for Java [2], [3], [4], [5]. In particular, the tools and techniques developed so far, which include DTDETECTOR [2], ELECTRICTEST [3], and PRADET [4], combine static and dynamic analysis to extract an approximate set of dependencies from read-after-write operations on Java objects that are shared among tests.

With this work, we consider a different context. We focus on *system tests* rather than unit tests. We also focus on Web applications and database management systems (e.g., the MySQL server), which are business-critical domains where dependability must be ensured with thorough testing, and that are representative of a broad class of distributed applications. In this context, there are still dependencies between tests, and the dependencies are still due to shared state and therefore data flow between tests. However, this shared state exists in less identifiable forms in various components at different levels (e.g., front end, application tier, database), which renders existing white-box dependence analysis not applicable. Biagiola et al. [5] also considered this context and developed a technique that infers likely read-after-write dependencies based on test names. Our general approach is different.

We treat the application and its tests purely as black boxes within a simple model of dependencies. This model is a special form of manifest test dependence as defined by Zhang et al. [2]. In essence, given a test suite $T = t_1, t_2, \ldots, t_n$ of n tests, which we take as a valid sequential schedule, we consider dependencies $t_i \longrightarrow t_j$, when $j > i$, that indicate that t_j depends on a prior execution of t_i. We further assume that the only way to detect such a dependency is to observe that t_j would fail in a sequence of tests $\ldots t_j$ that does not contain t_i (we discuss our model of dependencies in greater detail in Section 2).

With these basic assumptions, we develop two novel dependency detection algorithms. Like the state-of-the-art PRADET algorithm, both our algorithms essentially assemble a test dependency graph and then use it to produce a set of valid test schedules of minimal maximal length that cover

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3572388

- *Pasquale Polverino, Fabio Di Lauro, Matteo Biagiola, Paolo Tonella, and Antonio Carzaniga are with the Università della Svizzera italiana, Lugano, Switzerland. Street: Via Buffi 13. Postcode: 6900. E-mail: {pasquale.polverino, fabio.di.lauro, matteo.biagiola, paolo.tonella, antonio.carzaniga}@usi.ch.*

© 2025 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

all tests. The main intuition behind these two algorithms is in the way they confirm or exclude dependencies. In particular, the idea is to optimistically run test schedules that would trigger a series of dependencies, and therefore can confirm at least one dependency in case of failure, or exclude many dependencies when there are no failures. Also, unlike PRADET, these algorithms are designed to be themselves highly parallelizable. Indeed, PRADET was originally developed for and evaluated with Java unit tests, which typically run in the order of milliseconds. By contrast, the set-up and execution times for Web and MySQL system tests are in the order of seconds. Furthermore, MySQL support the execution of system tests with the Valgrind [6] memory analysis tool. The use of Valgrind is very valuable. On the other hand, Valgrind significantly increases the execution time of the test suite. This is where the efficiency, both in the analysis phase and in the parallelization of tests, is particularly desirable.

The first algorithm, which we call Parallel Framework for Automated System Testing (PFAST), iteratively excludes each individual test t_i from the initial schedule with the intent to optimistically exclude all dependencies on t_i. If a failure occurs, the algorithm can certainly include a dependency on t_i, and then it refines the schedule to detect other dependencies on t_i. In the worst case, PFAST requires the execution of $O(n^2)$ test schedules. The second algorithm, called MEMory-intensive FASt web Testing (MEM-FAST), incrementally builds and runs passing schedules of increasing length. In the worst case, MEM-FAST requires the execution of an exponential number of test schedules. However, the algorithm is very efficient and practical—outperforming both PFAST and PRADET—when the underlying test dependency graph is very sparse, that is, when tests are almost always independent.

We evaluate our algorithms on six real Web applications and their system-level tests, as well as on eight MySQL system-level test suites. These experiments demonstrate that our algorithms serve their primary purpose of automating the parallelization of test suites, obtaining a significant reduction in overall execution time. This reduction is the same achieved by PRADET, which makes sense, as the reduction is anyway a property of each test suite. However, the experiments also demonstrate that our algorithms run orders of magnitude faster than PRADET. We also conduct a Monte Carlo analysis of the asymptotic behavior of our algorithms as well as of the PRADET algorithm in the presence of various classes of test dependencies.

In summary, we contribute: (1) A general-purpose test-dependency detection algorithm called PFAST; (2) A special-purpose test-dependency detection algorithm called MEM-FAST that is particularly suitable for very sparse dependency graphs; (3) Concrete implementations of both algorithms within an automation and optimization framework for system-level testing; (4) An experimental study in which we analyze the benefits and performance of our system-level testing framework with real applications as well as with a battery of synthetic cases.

2 DEPENDENCY MODEL

Let $T = \langle t_1, t_2, \ldots, t_n \rangle$ be a given test suite consisting of n tests. A *schedule* for T is a set $\{S_1, S_2, \ldots\}$ of sequences $S_i \subseteq T$ of subsets of T. A sequence S_i is *valid* if the sequential execution of all the tests in S_i completes with no failures. We define a test failure as a violation of an oracle, either explicit (i.e., assertions written by developers in the test case), or implicit (e.g., uncaught exception). A schedule is valid if all its sequences are valid. We assume that the given, sequential scheduling of the test suite, $t_1, t_2, \ldots, t_n$ is valid. We therefore call the schedule consisting of the single sequence $S = T$ the *reference schedule*. In the rest of the paper we also sometimes refer to schedules consisting of a single sequence. In those cases, where there is no ambiguity, we use the term *schedule* to refer directly to their test sequence.

In our dependency model, there is no use in scheduling tests in any order other than the one consistent with the reference schedule. This is because, with a sequence containing tests t_i and t_j, either t_i must precede t_j, or vice-versa, or both relative orders are valid. Thus, since we assume that the reference schedule is valid, a schedule in the same order as the reference schedule is just as good as the latter. For example, consider a test suite t_i,t_j,t_k in which t_k depends on both t_i and t_j, and no other dependencies exist. Here, both t_i,t_j,t_k and t_j,t_i,t_k are valid (minimal) schedules, but there is no reason to prefer t_j,t_i,t_k over t_i,t_j,t_k. We therefore only consider schedules where every sequence S_i is a *subsequence* of T. In other words, for our purposes, a schedule of T consists of sequences of tests S_i obtained by removing tests from T.

A test *dependency* $t_j \rightarrow t_i$ is a binary, transitive relation between tests t_j and t_i, which we read as "t_j depends on t_i" and that we define as follows: $t_j \rightarrow t_i$, with $j > i$, if and only if every valid sequence S that contains t_j also contains t_i (before t_j).

Given a test suite $T = \langle t_1, t_2, \ldots, t_n \rangle$, the *test dependency graph (TDG)* is a directed graph $G = (V, A)$, with $V = \{t_1, t_2, \ldots, t_n\}$ and $A \subseteq V \times V$, where each arc $(t_j, t_i) \in A$ represents a *direct* dependency $t_j \rightarrow t_i$, meaning that there is a dependency $t_j \rightarrow t_i$, and there is no other test t_k such that $t_j \rightarrow t_k$ and $t_k \rightarrow t_i$.

The test dependency graph embodies the dependency relations and therefore also the main scheduling problem we want to solve. What we want is a schedule that is complete and also minimal. A schedule is *complete* if every test t_i is in at least one subsequence. A schedule $\{S_1, S_2, \ldots\}$ is *minimal* when (1) there are no two sequences S_i, S_j such that $S_i \subseteq S_j$, and (2) the length $\ell = \max\{|S_i|\}$ is minimal. A complete and minimal schedule can be computed from the test dependency graph as follows: for each vertex t_i that has no incoming arcs, add a sequence S_i defined by the transitive closure of the dependency relation starting from t_i, which can be computed with a breadth-first search, $S_i = BFS(G, t_i)$. Specifically, we iterate backward on the original test suite; for each test not included in any previously computed sequence, we use BFS to discover all its direct and indirect dependencies. We add those as a new sequence, and continue until we have covered all tests. And as specified above, the order of the tests within S_i is the same as within the reference schedule T.

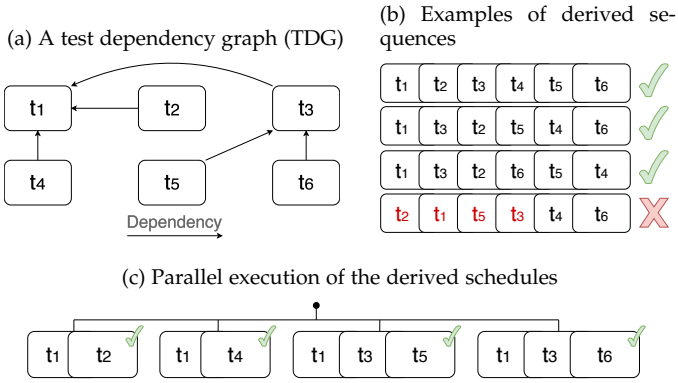

Figure 1: A test dependency graph (TDG), including derived sequences and their parallel execution.

Figure 1b presents examples of both valid and non-valid test sequences derived from the test dependency graph illustrated in Figure 1a. Additionally, Figure 1c illustrates the parallel execution of schedules that can be derived from the TDG.

Notice that our choice of dependency model is a special case of, and therefore a less general model than the model of manifest test dependence defined by Zhang et al. [2]. In other words, there are concrete cases of test dependencies, even resulting from relatively simple data flows on shared state, that can not be completely characterized within our model of dependencies (e.g., *alternative* dependencies, i.e., when a test t_j depends on either t_i or t_k). Still, we believe that our model is adequate in most practical cases, and our experimental evaluation supports this position. Notice also that this choice of model is a fundamental ingredient of the parallelization framework we propose. In fact, within the more general model, the problem of detecting dependencies is NP-hard, whereas under our model, our PFAST algorithm builds the test dependency graph efficiently, with $O(n^2)$ test suite executions in the worst case, and much closer to $O(n)$ in practice.

The question, then, is how accurate is the result of PFAST with respect to the actual dependencies, or even more concretely with respect to the correctness of the parallel schedule produced by PFAST. As detailed in Section 3.2.2, the dependency detection algorithm may fail to detect some dependencies that do not fit our model. We therefore add a final check—that the schedules are indeed error-free—and, if necessary, a repair phase to account for such additional dependencies. The key point, however, is that in practice the repair phase is rarely required, and the expected overall cost of the detection algorithm remains significantly lower than an algorithm that would account for those dependencies from the start.

3 ALGORITHMS

We now present the test dependency detection algorithms that are essential to our test automation and parallelization framework. However, before detailing our own algorithms PFAST and MEM-FAST, we describe an extension of the PRADET algorithm [4] that we implemented to deal with system-level test suites, and that we then use in our comparative evaluation.

3.1 PRADET Algorithm

PRADET was originally proposed by Gambi et al. [4] to detect dependencies for Java unit test suites. PRADET first creates an initial set of dependencies between the tests of a given test suite by running a dynamic data-flow analysis during the execution of the test suite. Such dependencies are organized in a test dependency graph. Then, PRADET refines the test dependency graph by testing each dependency individually, thereby removing the spurious ones. In the context of system-level testing, the initial data-flow analysis is most likely not feasible, since it would require a complete control over all the application components. The starting point of the algorithm is therefore the most conservative TDG, meaning that each test t_j is assumed to depend on every test t_i with $i < j$. The algorithm then continues with the same refinement procedure. See Algorithm 1 for details.

Algorithm 1: The PRADET algorithm for system-level test suites

Input : $T = \langle t_1, \ldots, t_n \rangle$, a test suite in the original order
Output: Test Dependency Graph $G = (V, A)$

1 $V \leftarrow \{t_1, \ldots, t_n\}$
2 $A \leftarrow \{(t_j, t_i) | t_i, t_j \in V \wedge 1 \leq i < j \leq n\}$
3 $S \leftarrow \emptyset$
4 **while** $\exists (t_j, t_i) \in A, (t_j, t_i) \notin S \wedge (V, A \setminus \{(t_j, t_i)\} \cup \{(t_i, t_j)\})$ *is acyclic* **do**
5 $S \leftarrow S \cup \{(t_j, t_i)\}$ /* *select arc* (t_j, t_i) */
6 $TDG' \leftarrow (V, A \setminus \{(t_j, t_i)\} \cup \{(t_i, t_j)\})$ /* *invert arc* */
7 $schedule \leftarrow$ COMPUTESCHEDULE(TDG', t_i)
/* *schedule based on the original order of the tests in* T *that satisfies the dependencies of* t_i *in* TDG'. */
8 $results \leftarrow$ EXECUTESCHDEDULE$(schedule)$
9 **if** $results[t_j] \neq FAIL$ **then**
10 $A \leftarrow A \setminus \{(t_j, t_i)\}$
11 **end**
12 **end**
13 $(V, A) \leftarrow$ COMPUTETRANSITIVEREDUCTION(V, A)
14 **return** (V, A)

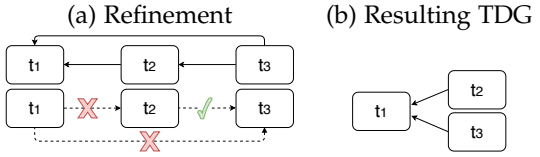

Figure 2: PRADET applied on the test suite $T = \langle t_1, t_2, t_3 \rangle$

The algorithm starts by building the vertex and the arcs sets of the initial, full test dependency graph (lines 1–2) where each node t_i is connected to all the preceding nodes (i.e., $t_{i-1}, \ldots, t_1$). The algorithm then checks that each arc in the dependency graph indeed represents a real dependency. The set S accumulates all the tested arcs in the main loop (lines 4–12) that runs as long as there is a yet untested arc in A. PRADET checks whether an arc (t_j, t_i) is indeed necessary by executing a sequence of tests

consistent with a test dependency graph (line 6) in which the candidate arc is inverted, and therefore in which t_j is executed before its supposed dependent test t_i. This inversion is checked only if it does not create a cycle in the TDG. This would happen if the candidate arc represents an *indirect* dependency implied by other arcs. For example, suppose $t_3 \rightarrow t_2$ and $t_2 \rightarrow t_1$, and consider now the implied (indirect) dependency $t_3 \rightarrow t_1$. Inverting $t_3 \rightarrow t_1$ would create the dependency cycle $t_3 \rightarrow t_2 \rightarrow t_1 \rightarrow t_3$.

In line 7, PRADET builds and runs a schedule that tests the selected dependency. The schedule is built through topological sorting, taking into account the original order of the tests in the test suite, and respecting all the other dependencies of t_j. If the schedule is successful, the arc is removed from the set A (line 10), meaning that test t_j does not need t_i to execute successfully, thus the dependency $t_j \rightarrow t_i$ is spurious. Finally, at line 13, PRADET computes the transitive reduction of the TDG to remove indirect dependencies, and returns the resulting graph.

Let us consider the execution of PRADET on the test suite $\langle t_1, t_2, t_3 \rangle$. Let us suppose that the underlying test dependency graph of such test suite has two dependencies, i.e., $t_2 \rightarrow t_1$ and $t_3 \rightarrow t_1$ (see Figure 2b). The initial test dependency graph (lines 1–2) built by PRADET contains three arcs, i.e., $t_2 \rightarrow t_1$, $t_3 \rightarrow t_2$, and $t_3 \rightarrow t_1$. Let us suppose that the first arc being inverted is $t_2 \rightarrow t_1$ (see Figure 2a). As t_2 has no other dependency, the schedule that PRADET computes (line 10) is simply $\langle t_2 \rangle$. As the execution of $\langle t_2 \rangle$ fails, the arc cannot be removed, and $t_2 \rightarrow t_1$ is marked as a *manifest* dependency. Let us suppose that the next arc being inverted is $t_3 \rightarrow t_2$. A schedule for t_3 that respects all dependencies except the inverted one is $\langle t_1, t_3 \rangle$. Since this schedule is successful, we can safely remove the arc $t_3 \rightarrow t_2$. Following the same procedure, the last arc to analyze is $t_3 \rightarrow t_1$, which cannot be removed because the resulting schedule (i.e., $\langle t_3 \rangle$) fails. The dependency refinement algorithm outputs the TDG shown in Figure 2b.

3.2 PFAST Algorithm

PFAST builds on the idea that removing a single test t_i from within a test suite would cause all the tests that depend on t_i to fail. Conversely, should there be no failures, then PFAST can safely exclude any dependency on t_i. Then, by iteratively repeating this process for all the tests, PFAST can identify all the dependencies in a test suite. The only complication arises in the failure case. If t_j is the first test that fails upon the removal of t_i, then we can safely conclude that t_j depends on t_i. However, we can not draw any conclusions regarding any other failure t_k after t_j, since that might indicate a dependency with t_j or just a spurious effect of the first failure of t_j. Therefore, in the case of failures, PFAST progressively removes the failing tests to determine the exact dependencies for all subsequent tests. Algorithm 2 shows the PFAST routine.

The algorithm starts by building the vertex and the arcs set of a disconnected test dependency graph in lines 1–2. The vertex set contains one vertex for each test in the test suite, while the arcs set is initially empty. In each iteration of the main loop (lines 3–16) PFAST attempts to remove one test from the original test suite to determine

Algorithm 2: PFAST

```
Input  : T = ⟨t_1, ..., t_n⟩, a test suite in its original order.
Output: Test Dependency Graph G = (V, A)
1  V ← {t_1, ..., t_n}
2  A ← ∅
3  for i = 1 to n − 1 do
4      S ← T \ ⟨t_i⟩                          /* remove t_i from T */
5      results ← EXECUTESCHEDULE(S)
6      while GETFAILEDTESTS(results) ≠ ∅ do
7          for j = i + 1 to n do
8              if results[t_j] = FAIL then
9                  A ← A ∪ {(t_j, t_i)}
10                 S ← S \ ⟨t_j⟩               /* remove t_i from S */
11                 break
12             end
13         end
14         results ← EXECUTESCHEDULE(S)
15     end
16 end
17 (V, A) ← COMPUTETRANSITIVEREDUCTION(V, A)
18 return (V, A)
```

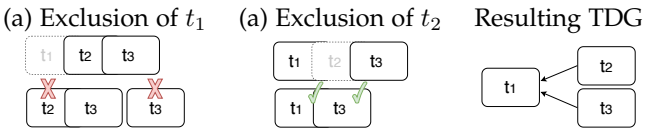

Figure 3: PFAST applied on the test suite $T = \langle t_1, t_2, t_3 \rangle$

if subsequent tests depend on it. In case of failures, then, the inner loop (lines 6–15) identifies the exact dependencies with t_i by progressively excluding failing tests. In particular, a first failure of t_j indicates that the dependency $t_j \rightarrow t_i$ exists. Then, PFAST removes t_j and reruns the remaining sequence to detect other dependencies $t_k \rightarrow t_i$, with $k > j$. When the main loop ends, PFAST computes the transitive reduction of the obtained TDG in line 17, to remove indirect dependencies.

Figure 3 shows an example of the execution of PFAST on the test suite $\langle t_1, t_2, t_3 \rangle$. The first test being excluded is t_1 (Figure 3a). PFAST runs the sequence $\langle t_2, t_3 \rangle$, where t_2 fails, PFAST therefore also removes t_2 and executes the sequence $\langle t_3 \rangle$, where t_3 also fails (Figure 3a). Correspondingly, PFAST adds two arcs to the TDG: $t_2 \rightarrow t_1$ and $t_3 \rightarrow t_1$. In the next main iteration, PFAST excludes test t_2 (Figure 3b). The resulting sequence $\langle t_1, t_3 \rangle$ executes without failures, and therefore no other arc is added to the TDG. The resulting TDG is shown in Figure 3c.

Notice that the worst-case complexity is quadratic for both PRADET and PFAST, since both algorithms require the executions of $O(n^2)$ test sequences when the test dependency graph is complete. However, PRADET *always* iterates over a quadratic number of arcs (see line 4 in Algorithm 1), while PFAST is linear in the best case, in the absence of test dependencies. As we show experimentally, the cost of PFAST is substantially lower than the cost of PRADET.

3.2.1 *Correctness of* PFAST

We now prove that PFAST finds all and only the existing dependencies that conform with our model of dependencies. The correctness derives directly from the following

loop invariant: at the start of each iteration of the main loop (lines 3–16), the set of arcs A contains all and only the arcs that represent dependencies from tests in $\{t_i, \ldots, t_n\}$ to test t_{i-1}.

Initialization.

The invariant is trivially true before the first iteration: when $i = 1$, the set of arcs A is empty. Since there is no test before t_1, there is no test on which the tests $\{t_i, \ldots, t_n\}$ depend on.

Maintenance.

The body of the main loop works by creating a sequence that removes t_i from the reference schedule. Then, we run the resulting sequence. For the tests $\{t_1, \ldots, t_{i-1}\}$, there will not be any failures since this is a prefix of the original test suite, which is assumed to be passing. Therefore, we will not enter the while-loop of lines 6–15, resulting in no arc added to A, for such tests. For a test $t_j \in \{t_{i+1}, \ldots, t_n\}$, there are two cases that need to be considered, whether t_j passes or fails. As per our dependency model, if t_j does not depend on t_i, then the exclusion of t_i will not cause the failure of t_j. In other words, t_j will pass, and no arc (t_j, t_i) will be added to A in line 9. If on the other hand, t_j does depend on t_i, then according to our dependency model, the exclusion of t_i will cause the failure of t_j, which will trigger the execution of the while-loop of lines 6–15 at some point, resulting in arc (t_j, t_i) being added to A in line 9. Therefore, incrementing i to the next iteration of the main loop (lines 3–16) will preserve the loop invariant.

Termination.

The main loop terminates when $i = n$. Because each loop iteration increases i by one, we must have $i = n$ at the time of termination. Substituting n for i in the loop invariant, we have that the set of arcs A will contain all and only the arcs that represent the dependencies from tests in t_n to test t_{n-1}. Since t_n is the last test of the test suite, we can conclude that we have found all the dependencies between the tests in the test suite. Hence, the algorithm is correct. □

3.2.2 *Repair of* PFAST

PFAST is correct provided that the test dependencies are accurately modeled by the test dependency graph as defined in Section 2. However, this might not be the case. For example, a test t_k might depend on one of two other tests, t_i and t_j, but not necessarily on any one of them in particular. In such a case, PFAST successfully removes, individually, both t_i and t_j (line 4 of Algorithm 2) with no error, resulting in a false negative, meaning failing to detect any dependency $t_k \to t_i$ or $t_k \to t_j$. As a consequence, one or more of the parallel schedules obtained from the TDG will fail, requiring a schedule repair operation.

We therefore complement PFAST with a validity check and with a conservative repair algorithm that restores the missing dependencies by reintroducing segments of test sequences taken from the original order, until a passing schedule is found. Specifically, the repair adds an arc to the TDG (a possible dependency) between each failing test and all the preceding tests (in the reference schedule). With this TDG, we attempt to prune the newly added arcs using a strategy similar to PRADET. In terms of complexity, the repair has a cost of $O(n^2)$, so the overall worst-case performance is not affected.

3.2.3 *Parallelization of* PFAST

One of the points of strength of PFAST is the possibility of parallelizing the PFAST itself. In our experimentation, we use an implementation based on a parallelized version of the algorithm. The main idea is that the generation and execution of a single sequence of tests can happen independently from other sequences. Hence, each sequence can run in a separate thread. PFAST is therefore "trivially" parallelizable on its main loop, as shown in Algorithm 3 (the only changes w.r.t. Algorithm 2 are highlighted in red).

Algorithm 3: Parallelized PFAST

```
   Input  : T = ⟨t_1, ..., t_n⟩, a test suite in its original
            order.
   Output: Test Dependency Graph
 1 V ← {t_1, ..., t_n}
 2 A ← ∅
 3 mtx ← NEWMUTEX()
 4 parallel for i = 1 to n − 1 do
 5 |   S ← T \ ⟨t_i⟩
 6 |   results ← EXECUTESCHEDULE(S)
 7 |   while GETFAILEDTESTS(results) ≠ ∅ do
 8 |   |   for j = i + 1 to n do
 9 |   |   |   if results[t_j] = FAIL then
10 |   |   |   |   ACQUIRE(mtx)
11 |   |   |   |   A ← A ∪ {(t_j, t_i)}
12 |   |   |   |   RELEASE(mtx)
13 |   |   |   |   S ← S \ ⟨t_j⟩
14 |   |   |   |   break
15 |   |   |   end
16 |   |   end
17 |   |   results ← EXECUTESCHEDULE(S)
18 |   end
19 end
20 (V, A) ← COMPUTETRANSITIVEREDUCTION(V, A)
21 return (V, A)
```

3.3 MEM-FAST Algorithm

The Memory-intensive Fast Web Testing (MEM-FAST) algorithm builds on the idea that an optimal parallelizable schedule consists of valid (passing) sequences of tests of minimal length that cover all tests. The algorithm therefore builds such sequences incrementally, by extending previously checked (passing) sequences with tests that have not yet been covered. MEM-FAST starts with sequences of length one by running each test in isolation and keeping the passing ones. Then, sequences of length two are valid (passing) sequences obtained by concatenating valid sequences of length one with tests that failed in the previous executions. This process continues as long as there are failing tests.

This sequence-extension process tries to cover new tests but does not in fact enumerates all possible sequences. In fact, the extension of a sequence to incorporate test t_i is considered only if it immediately includes t_i right after a known prefix sequence. But what if t_i depends on more tests that have already been covered? In that case, the extension algorithm would fail to cover t_i. That is when MEM-FAST

Algorithm 4: MEM-FAST

```
   Input  : T = ⟨t_1, ..., t_n⟩: test suite in its original order
   Output: A test schedule {S_1, S_2, ...}
   Output: Test Dependency Graph G = (V, A)
1  V ← {t_1, ..., t_n}
2  A ← ∅
3  P ← ∅
4  F ← ∅
5  for i = 1 to n do
6      results ← EXECUTESCHEDULE(⟨t_i⟩)
7      if results[t_i] = PASS then
8          P ← P ∪ {⟨t_i⟩}
9      end
10     else
11         F ← F ∪ {t_i}
12     end
13 end
14 r ← 1
15 while F ≠ ∅ do
16     passed ← false
17     foreach sequence S ∈ P such that |S| = r ∧ k < i,
        with S = ⟨s_j, ..., s_k⟩ do
18         foreach t_i ∈ F do
19             results ← EXECUTESCHEDULE(S · ⟨t_i⟩)
20             if results[t_i] = PASS then
21                 P ← P ∪ {S}
22                 F ← F \ {t_i}
23                 A ← A ∪ {(t_i, s_k)}
24                 passed ← true
25             end
26         end
27     end
28     if passed ∧ t_r ∉ F then
29         r ← r + 1
30         continue;
31     end
32     foreach sequence S = ⟨s_j, ..., s_k⟩ such that
        s_l ∈ {t_1, ..., t_{i-1}} with j ≤ l ≤ k and S has a prefix
        in P increasingly ordered by length do
33         results ← EXECUTESCHEDULE(S · ⟨t_r⟩)
34         if results[t_r] = PASS then
35             P ← P ∪ {S, S · ⟨t_r⟩}
36             F ← F \ {t_r}
37             A ← A ∪ {(t_r, s_j), ..., (t_r, s_k)}
38             r ← |S| + 1
39             break
40         end
41         if results[s_k] = PASS then
42             P ← P ∪ {S}
43         end
44     end
45 end
46 (V, E) ← COMPUTETRANSITIVEREDUCTION(V, E)
47 return (V, E)
```

switches to an exhaustive search. Algorithm 4 shows the pseudo-code of MEM-FAST.

The algorithm starts by building four sets (lines 1–13): The set P of all the passing sequences derived from the reference schedule T, the set F of all the currently failing tests from T, the set of vertices V and arcs A of a disconnected test dependency graph. Initially, the algorithm considers individual tests and therefore sequences of length 1 (lines 5–13). Then, starting from those initial sets P and F, the algorithm tries to find progressively longer sequences that cover more and more tests (lines 15–45). In the first for-loop (lines 17–27), MEM-FAST tries to extend the passing sequences by appending each test $t_i \in F$ to each sequence S of length r in P. The value of r initialized at line 14, represents the length of the schedule we try to append tests to, at a given point. If the resulting schedule passes, we can conclude that there is a direct dependency between t_i and the last test in S, therefore we add an arc in the TDG (line 23). However, this first loop does not cover the case in which a test depends on more than one test. We solve this issue by introducing an exhaustive search over all the combinations of passing schedules in lines 32–44. By ordering the schedules by length during the exhaustive search, we guarantee the minimality in the length of the resulting parallel schedules.

The exhaustive search in lines 32–44 is guaranteed to find a valid sequence that covers t_i, but it is also the part of the algorithm that requires exponential time and space. MEM-FAST exhibits this exponential behavior in the presence of many dependencies, while being very efficient when the dependency graph is very sparse.

Figure 4 shows an example of execution of MEM-FAST on the test suite $\langle t_1, t_2, t_3 \rangle$. After the execution of sequences of length one (see lines 5–13 in Algorithm 4), the two sets P and F are $P = \{\langle t_1 \rangle\}$ and $F = \{t_2, t_3\}$ (see Figure 4.a). With $r = 1$, MEM-FAST builds two candidate prefixes $\langle t_1, t_2 \rangle$ and $\langle t_1, t_3 \rangle$ in lines 17–27. As both are successful (see Figure 4.b), MEM-FAST adds the arcs (t_2, t_1) and (t_3, t_1) to the TDG, skipping the exhaustive search (see lines 32–44 in Algorithm 4) and terminates the loop at line 15, returning the resulting test dependency graph shown in Figure 4.c.

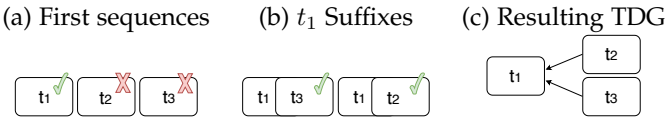

Figure 4: MEM-FAST Applied on the Test Suite $T = \langle t_1, t_2, t_3 \rangle$

4 EXPERIMENTAL EVALUATION

We now present the results of the experiments that we conducted to evaluate the algorithms and the overall test optimization framework we propose. We conducted experiments with existing real systems (i.e., Web applications and database management systems) and their system-level tests to assess the practical value of our proposed algorithms. We also experimented with synthetic dependency graphs to assess the asymptotic behaviors of the algorithms in the presence of various types of dependency relations. At a high-level, we set out to answer the following research questions:

RQ1 (Dependency Detection) What is the execution cost of each dependency detection technique?

RQ2 (Test Execution) What is the expected execution cost saving brought by the parallelized schedules?

We analyze the execution cost of each method for determining the dependencies between tests (RQ1) using both synthetic and real system-level test suites.

Metrics (RQ1). With synthetic test suites, we measure the execution cost by counting the number of schedules that each dependency detection algorithm has to run to produce its final output, as well as the number of tests in such schedules. Indeed, with synthetic tests, each test has the same execution cost (i.e., no execution cost), hence the number of schedules (and test runs) gives a reliable indication of the computation requirements of each dependency detection algorithm. We also vary the number of tests in each test suite, to analyze how the execution cost of each algorithm increases with the test suite size. With real system-level test suites, beyond measuring the number of schedules and their tests, we measure the actual clock time each algorithm takes to complete the dependency detection task, because each test has its own runtime (typically in the order of seconds) and each schedule execution includes the cost of setting up the system (e.g., a Web application or a database management system like the MySQL server).

With the second research question (RQ2), we study the benefit of the parallelization framework for testing. In particular, we analyze the speed-up of the parallelizable schedules generated by each dependency detection technique, as compared with the sequential execution of a given test suite in its original order.

Metrics (RQ2). In the case of synthetic tests, we simply measure the number of test cases in the longest schedule extracted/generated by each dependency detection algorithm, and compare it against the given test suite size. With real system-level test suites, we measure the execution time of the extracted/generated schedule with the highest runtime, assuming that we can parallelize all the schedules. We then compare such runtime with the sequential runtime of the original test suite.

4.1 Subjects

We conducted our experiments using two different families of subjects, namely synthetic tests, and real system tests. Synthetic tests allow us to control for important factors, such as the test suite size and the test dependency graph connectivity and shape, and therefore give us a more general and theoretical perspective. Real systems with their system tests give us indications on the expected performance in practical cases. For completeness, we have also evaluated PFAST and MEM-FAST on the unit test subjects used to evaluate the original version of PRADET [4]. We report the results of the comparison in the appendix.

4.1.1 Synthetic Tests

We created random test dependency graphs using three distinct algorithms employed in random graph generation, namely Barabási–Albert [7], Erdős–Rényi [8], and Out-Degree-3-3. "Synthetic" tests are nodes of such test dependency graphs, with unitary execution cost (they are actually not executed at all).

The Barabási–Albert [7] strategy creates graphs where a few nodes have a much higher number of connections than other nodes. In particular, the probability of a new node t_j being connected to an existing node t_i, is proportional to the degree of node t_i (i.e., the sum of its incoming and outgoing edges), hence nodes with a higher degree are more likely to attract connections. This model is also called preferential attachment. For instance, the probability p_i that a node is connected to node t_i is given by $p_i = d_i / \sum_{j=1}^{i-1} d_j$ where d_i is the degree of node t_i, and the sum is made over the degrees of all the nodes generated before t_i.

The Erdős–Rényi [8] strategy establishes that each node t_i is connected to any other node t_j with a certain, fixed probability that is independent of t_i or t_j. This model is intended to generate graphs with no preferential attachment, and therefore no particular structure. The last category of graph generation algorithm is Out-Degree-3-3, which we designed to generate graphs where each node is connected with exactly three nodes at random. Such strategy ensures that each node in the test dependency graph has the same average degree.

The three categories of graph generation algorithms cover a diverse range of graph structures, representing different kinds of test dependency graphs that might characterize real system-level test suites: respectively, (1) TDGs with a few central nodes on which several other nodes depend; (2) unstructured TDGs where nodes have the same average connectivity; (3) TDGs where the number of dependencies is assumed to be limited, to 3 in our case.

Table 1: Overview of Web applications and their test suites. The first macro-column (WEB APP) show the version of the corresponding Web application and the respective Lines of Code (LOC). The second macro-column (TEST SUITES) shows the number of tests in the test suite of the corresponding Web App (#), as well as the average number of LOC in each test divided by the total number of LOC.

APP	WEB APP		TESTS SUITES	
	Version	LOC	#	LOC (Avg/Tot)
addressbook [9]	8.0.0	16,383	27	48/1,355
claroline [10]	1.11.10	352,968	40	46/1,834
expresscart [11]	1.1.19	1,383,562	27	37/1,008
mantisbt [12]	1.1.8	211,265	41	43/1,748
mrbs [13]	1.4.9	35,902	22	51/1,121
ppma [14]	0.6.0	578,858	23	52/1,186
Total		2,578,938	201	46/8,252

Table 2: Overview of MySQL test suites by their respective functionalities. The column "# Tests" shows the number of tests for each functionality, while the column "Big Tests" shows the test suites that contain at least one computationally expensive test case (or "big" test).

Functionality	# Tests	Big Tests
audit_null	15	No
collations	32	No
jp	111	No
json	19	No
gcol	34	Yes
gis	70	No
innodb_zip	26	Yes
information_schema	8	No

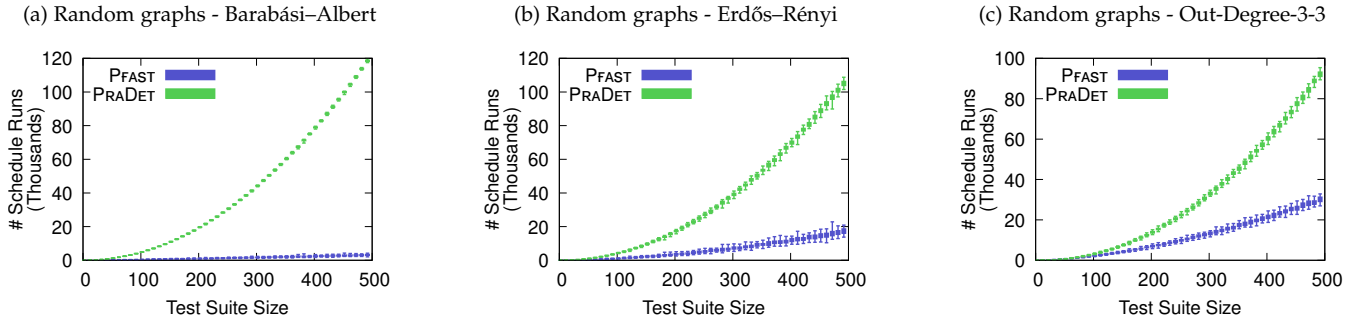

Figure 5: Asymptotic behavior of PFAST and PRADET in terms of number of schedule runs over the test suite size.

4.1.2 Real System-level Test Suites

Web applications. We selected six open-source Web applications used in previous Web testing research [15], [5], [16], [17]. Each subject is accompanied by a JUnit test suite, which includes 21—41 Selenium [18] test cases. Table 1 shows our subject systems, including their names, versions, lines of code (LOC) of both the Web applications and the associated test suites (measured using `cloc` [19]), as well as the number of tests in each test suite.

MySQL. We selected eight test suites for as many functionalities from the MySQL server test suite [20]. In the selection process we tried to cover a good range of test suite sizes. In particular, the smallest size is 8 (*information_schema*) and the largest size is 111 (*jp*). Additionally, we also included test suites containing tests marked as "big" [21]. In MySQL terminology, those tests either take a very long time to run or use considerable resources, making them unsuitable for a normal test suite run [21]. Table 2 shows our selected test suites, also including those that contain at least a "big" test.

4.2 Experimental Procedure

We now detail the procedure we used to run the experiments and collect the corresponding metrics for both synthetic and system-level test suites.

Table 3: Number of random graphs generated by each graph generation strategy for each dependency detection technique.

	# Graphs	Techniques
Erdõs–Rńyi	300	PRADET, PFAST, MEM-FAST
Erdõs–Rńyi	2500	PRADET, PFAST
Barabási–Albert	2500	PRADET, PFAST
Out-Degree-3-3	2500	PRADET, PFAST

4.2.1 Synthetic Tests

Regarding *dependency detection* (RQ1), for each random graph generation strategy, we varied the number of nodes n (i.e., the number of tests in the test suite) from 2 to 492 in steps of 10. For each value of n, we generated 50 random graphs, resulting in a total of 2500 random graphs for each technique. For the Erdős–Rényi strategy we employed a fixed probability set to $(\log n)/n$ so that the expected number of edges is on the order of $O(n \log n)$. Given that MEM-FAST exhibits exponential growth with the increasing number of schedules as n increases, we examined its behavior by limiting n to 50. We generated 50 random graphs for each of six different fixed probabilities (0.0001, 0.0005, 0.001, 0.005, 0.01, and 0.02) using the Erdős–Rényi strategy, resulting in a total of 300 random graphs. For graphs consisting of 50 nodes and higher probabilities, MEM-FAST manifests its (expected) exponential behavior, failing to complete within a 30-minute time limit. With this algorithm, we only used the Erdős–Rényi strategy, as we can control the number of edges in the random graph, as opposed to Barabási–Albert and Out-Degree-3-3, and we studied the dependency cost of each algorithm (i.e., MEM-FAST, PRADET and PFAST) when the number of connections in the graph varies. Table 3 summarizes the number of generated random graphs for each dependency detection technique.

Regarding *test execution* (RQ2), for the techniques that output a test dependency graph, namely PRADET and PFAST, we generated minimal parallel test schedules automatically by traversing the TDG to obtain the transitive closure of the dependency relations.

4.2.2 Real System-level Test Suites

Web Test Suites. Regarding Web application system-level test suites, for *dependency detection* (RQ1), we configured our execution environment to run both the Web application and the associated test suite in separate Docker containers. Each schedule's execution requires the setup of the initial state of the Web application; to reset the state across schedule executions, we simply delete and recreate the Web application containers. We used the parallelized version of PFAST in our experiments; we configured PFAST and MEM-FAST to run at most 12 instances of the Web application under test at the same time (i.e., at most 12 schedules can be executed in parallel) [1]. All the experiments have been executed on a virtual machine running on Ubuntu 22.04.3 LTS with 16 Intel(R) Xeon(R) Gold 5218 CPUs and 32GB of RAM. To make the comparison with PRADET fair, we spawned the same number of instances of the Web application under test as PFAST and MEM-FAST. Even though PRADET cannot

1. The 12 instances account for three quarters of the total number of CPU cores in the VM we used for the experiments. For *mrbs* we used 4 parallel instances, to reduce the impact of flakiness.

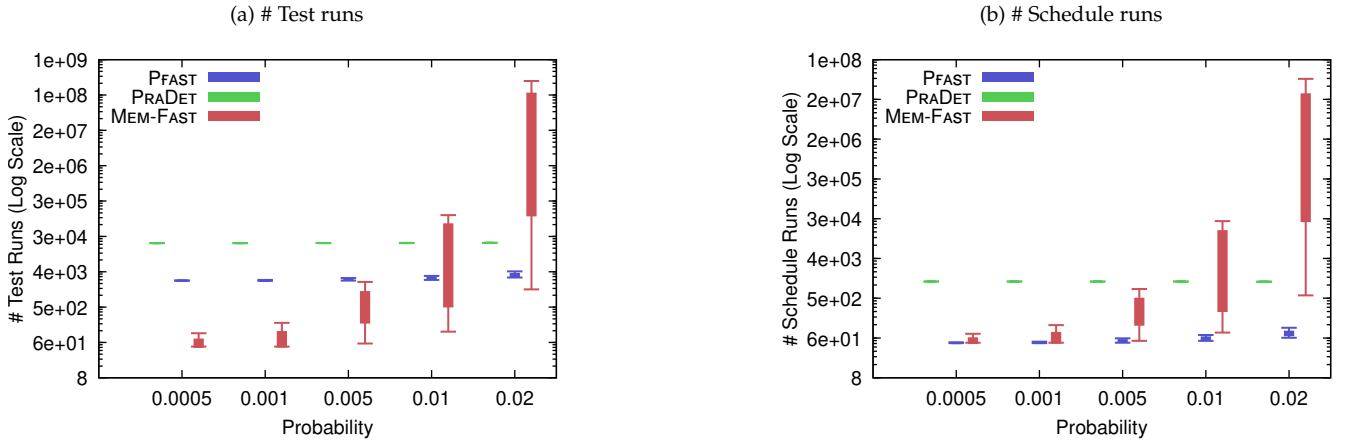

Figure 6: Dependency detection cost of PFAST, PRADET and MEM-FAST, for Erdős-Rényi randomly generated graphs.

execute schedules in parallel, it benefits from having multiple running instances of the application, to avoid waiting for the setup of the application each time it requests the execution of a schedule. This way, the initial setup cost for each schedule becomes negligible for PRADET.

To account and compensate for *flakiness*, i.e., the tendency of some tests to fail non-deterministically [22], we manually fixed each test suite before running the dependency detection algorithms. In particular, we added appropriate delays in synchronization points, as Web test cases are affected by timing issues (e.g., an element is clicked before it is fully loaded on the page, causing the test to break). This did not completely eliminate the problem of flaky tests, especially with our detection algorithms, i.e., PFAST and MEM-FAST, that run test suites in parallel. We therefore executed each algorithm ten times and analyzed results across multiple runs. For each run of each dependency detection technique, we set a maximum time budget of 24h.

Regarding *test execution* (RQ2), for PRADET, PFAST and MEM-FAST we extracted the parallelizable schedules from the resulting TDG. When executing the resulting schedules for each technique, we configured the tool to execute at most 6 schedules in parallel.

MySQL Test Suites. Regarding MySQL system-level test suites, for *dependency detection* (RQ1) we used the same experimental setup as the one used for the Web application test suites. In practice, we use the same hardware, we run the MySQL test suites within Docker containers, and we execute each dependency detection algorithm ten times with a timeout of 24h. MySQL test suites are more demanding in terms of resources as compared to the Web application test suites. We therefore configure the algorithms to use at most 8 instances of MySQL at the same time (instead of the 12 instances we used for Web applications).

Regarding *test execution* (RQ2), we extracted the parallelizable schedules from the resulting TDGs as we did for Web test suites. We then executed the sequential test suites and the parallelizable schedules of PRADET, PFAST and MEM-FAST while running the system under test with Valgrind [6], enabled for those test suites supporting it (namely, those with no "big" tag). MySQL allows to use Valgrind as a memory debugging and checker tool, to ensure that a release does not introduce any memory-related bugs or vulnerabilities. In particular, Valgrind increases the running time of a test case by a factor of 10 [21].

4.3 Results

4.3.1 RQ1 (Dependency Detection)

Synthetic Tests. Regarding synthetic tests, Figure 5 shows the asymptotic behavior of PRADET and PFAST in terms of number of schedule runs as the test suite size increases (we report the plots in terms of number of test runs, which show a similar trend, in our replication package). Each point in the figures is a box plot showing the variation in the number of schedule runs for the 50 generated random graphs of the given size. In every random graph generator category, the cost of PRADET surges with larger test suite sizes, whereas PFAST's growth is markedly slower. The disparity between PRADET and PFAST varies depending on the specific random graph generator used. Specifically, considering the Barabási–Albert strategy, PFAST and PRADET starts to diverge with a small test suite size (i.e., < 30), while with Erdős–Rényi and Out-Degree-3-3 , the divergence point progressively moves towards higher test suite sizes (i.e., ≈ 50 and ≈ 100 respectively). Even in the worst case scenario, i.e., Out-Degree-3-3, the number of schedule runs for PFAST is ≈ 3 times smaller than PRADET's.

Figure 6 shows how the cost of PFAST, PRADET, and MEM-FAST vary when the probability of the Erdős–Rényi random generator increases (i.e., when the generated graphs become progressively more connected). On the left-hand side, the cost is represented by the number of test runs, while on the right-hand side by the number of schedule runs. For each probability value, the figures show three box plots, i.e., one for each technique, showing the cost variation for 50 randomly generated graphs in a logarithmic scale. In terms of both test runs and schedule runs, both PRADET and PFAST have a constant cost, as they do not depend on how connected is the underlying test dependency graph. On the other hand, MEM-FAST is exponentially more expensive the more connected the test dependency graph becomes, both in terms of number of test runs and number of schedule runs. However, MEM-FAST is significantly more efficient

Table 4: Dependency Detection (RQ1) and Test Execution (RQ2) for real system-level Web test suites (first 8 rows) and MySQL test suites in all subjects (last 10 rows). The results are averaged across ten runs. The "*OOB*" symbol means *Out of Budget*, i.e., the technique exhausted its 24h budget without terminating the dependency detection process (when computing the median we did not consider the rows with the *OOB* symbol). We measured the execution time of MySQL test suites ("Execution Time" column) while running Valgrind [6] (except for *gcol* and *innodb_zip*).

		RQ1 (Dependency Detection)									RQ2 (Test Execution)			
		# Deps. Found			# Schedules Runs			Dependency Detection Time (s)			Execution Time (s)			
	Subjects	PRADET	PFAST	MEM-FAST	PRADET	PFAST	MEM-FAST	PRADET	PFAST	MEM-FAST	Sequential	PRADET	PFAST	MEM-FAST
Web Test Suites	addressbook [9]	31	30	27	320.0	90.0	3,363.0	7,241.0	701.7	11,191.9	62.1	58.7	59.5	44.5
	claroline [10]	42	42	42	751.0	138.0	8,143.1	26,801.5	1,436.6	36,071	104.9	60.4	59.4	58.8
	expresscart [11]	22	22	17	342.0	67.0	218.4	16,732.9	1,042.6	1,402.5	123.0	41.5	41.2	35.7
	mantisbt [12]	35	35	33	776.0	143.0	347.0	42,583.5	2,466.1	1,784.1	217.0	157.5	159.8	91.2
	mrbs [13]	21	21	21	198.0	98.0	968.0	6,743.5	1,153.6	5,226.1	75.1	45.0	46.1	36.8
	ppma [14]	22	22.4†	18	229.0	75.0	176.1	5,881.3	749.8	606.3	68.9	52.1	48.8	32.1
	Median	*26.5*	*26.2*	*24.0*	*331.0*	*94.0*	*657.5*	*11,987.0*	*1,098.1*	*3,505.1*	*90.0*	*55.4*	*54.1*	*40.6*
MySQL Test Suites	audit_null	0	0	0	105.0	29.0	15.0	6,315.3	265.7	95.5	1,400.4	463.4	464.4	466.0
	collations	*OOB*	0	0	*OOB*	63.0	32.0	*OOB*	2,102.6	253.2	17,889.5	*OOB*	872.1	874.4
	jp	*OOB*	0	0	*OOB*	221.0	111.0	*OOB*	6,396.0	682.0	12,052.7	*OOB*	1,077.9	1,066.3
	json	0	0	0	171.0	37.0	19.0	27,932.9	847.7	146.9	8,005.3	2,187.7	2,184.8	2,187.0
	gcol*	*OOB*	0	0	*OOB*	67.0	34.0	*OOB*	3,766.4	493.3	702.3	*OOB*	396.2	398.6
	gis	*OOB*	0	0	*OOB*	139.0	70.0	*OOB*	2,763.0	428.0	6,708.6	*OOB*	786.1	789.9
	innodb_zip*	*OOB*	0	0	*OOB*	51.9	26.0	*OOB*	4,755.6	372.2	1,063.7	*OOB*	291.1	289.9
	information_schema	0	0	0	28.0	15.0	8.0	1,490.5	127.0	54.7	1,287.7	473.1	458.1	461.6
	Median	*OOB*	*0*	*0*	*OOB*	*57.5*	*29.0*	*OOB*	*2,432.8*	*312.7*	*4,054.5*	*OOB*	*625.2*	*628.0*

* *Test suites with "big" tests (see Table 2); test execution (RQ2) with Valgrind not supported;*
† *In some runs flakiness triggers the recovery of a few false positive dependencies.*

than PFAST and PRADET when the test dependency graph is sparse (i.e., from probability 0.0005 to probability 0.005). This is evident in the number of test runs (left-hand side of Figure 6), where MEM-FAST executes $\approx 30\times$ fewer tests than PRADET and PFAST. In the number of schedules, PFAST and MEM-FAST are comparable for sparsely connected graphs, as MEM-FAST executes single-test schedules, while PFAST's schedule runs include many more tests.

Web Test Suites. Regarding Web test suites, the first 8 rows of Table 4 show the results in terms of RQ1 (Dependency Detection) and RQ2 (Test Execution), averaged across ten runs. Columns 5–7 (number of schedule runs) and Columns 8–10 (dependency detection time in seconds) show the results for RQ1 for all the six test suite subjects.

Columns 5–6 show that PFAST is significantly less costly than PRADET in terms of number of schedule runs. The improvements range from $2\times$ (*mrbs*) to $5\times$ (*claroline*). This reduction directly translates into a much lower dependency detection time (Columns 7–8) for PFAST. The least difference is with *ppma*, where PFAST takes 12 minutes on average to complete the task while PRADET takes 1 hour and 38 minutes. With *mantisbt* we instead observe the highest difference between PFAST and PRADET, from 41 minutes on average to almost 12 hours. On the other hand, MEM-FAST (Column 7) generates many more schedules as compared to PRADET, yet it is faster than PRADET in terms of dependency detection time in the majority of subjects (i.e., 4 out of 6). The reason is twofold: schedules generated by MEM-FAST are generally very short and MEM-FAST executes those schedules in parallel, while PRADET executes them sequentially. Overall, PFAST outperforms MEM-FAST by a large margin for 4 out of 6 test suites, both in terms of number of schedule runs and dependency detection time, since test dependency graphs for such test suites are not sparsely connected.

MySQL Test Suites. Regarding the MySQL system-level test suites, the last 10 rows of Table 4 show the results in terms of RQ1 and RQ2. In this case the number of dependencies found by each algorithm is the same (Columns 2–4). MEM-FAST executes the least amount of schedules (Columns 5–7) in all subjects, while PRADET, when it does not time out, executes from $2\times$ to $4\times$ more schedule than PFAST and from $3.5\times$ to $9\times$ more schedules than MEM-FAST. MEM-FAST is also the most efficient in terms of dependency detection time with a runtime of only 312 seconds based on the median across all subjects. On the contrary, PRADET runs out of budget in most of the subjects (i.e., 5 out of 8 subjects), while taking much longer than PFAST and MEM-FAST in the remaining cases. This result confirms that MEM-FAST is quite efficient when the dependency graph is sparse, while PFAST remains a competitive alternative even in this context.

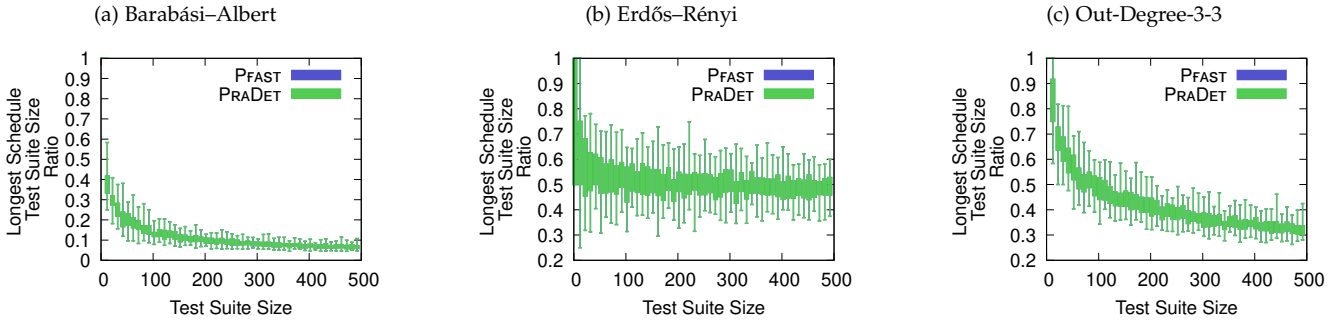

Figure 7: Asymptotic behavior of PFAST and PRADET, in terms of ratio between the longest schedule and the test suite size. As both techniques always output the optimal test dependency graph, the box-plots of PFAST and PRADET completely overlap.

RQ1 (Dependency Detection): Overall, in both synthetic and real test suites, PFAST *outperforms* PRADET in terms of dependency detection cost. Synthetic tests show that PFAST scales better than PRADET when the test suite size increases, and real system-level tests show that, in practice, PFAST takes from 6 to 19× less time than PRADET to detect dependencies on Web test suites, while it takes from 11 to 40× less on MySQL test suites. Moreover, in synthetic tests, MEM-FAST significantly *outperforms* both PRADET and PFAST when the underlying test dependency graph is sparse. This is confirmed by the MySQL test suites results, where there are no dependencies between tests, and MEM-FAST is the most efficient algorithm.

4.3.2 RQ2 (Test Execution)

Synthetic Tests. Concerning synthetic tests, both PRADET and PFAST output the same test dependency graph for all categories of random graph generators, which corresponds to the underlying (optimal) test dependency graph. This also applies to MEM-FAST, whose longest schedule corresponds to the longest schedule of the optimal test dependency graph. For PFAST and PRADET we report the ratio between the longest schedule and the test suite size, as the test suite size increases (Figure 7). For each test suite, figures show the box plots with the ratios given the 50 randomly generated graphs. As both techniques always output the optimal test dependency graph, the box-plots of PFAST and PRADET completely overlap. Figure 7 shows that for all categories of random graph generators, the ratio between longest schedule and test suite size, decreases with the test suite size. In particular, for the Barabási–Albert strategy (Figure 7a), we have the highest reduction, converging to 10% of the length of the test suite; for Out-Degree-3-3 (Figure 7b), the ratio decreases less quickly, asymptotically converging to a 30% reduction on average. On the other hand, for Erdős–Rényi , the reduction stays around 50%, as the graphs generated with such strategy have a totally random structure.

Web Test Suites. Regarding Web test suites, Columns 2–4 of Table 4 show the number of dependencies in the resulting test dependency graphs found by each technique for each subject. We observe that all techniques output a similar test dependency graph also with real Web test suites (i.e., the first 8 rows). In particular, in two out of six cases (*claroline* and *mrbs*), the number of dependencies is the same for each technique, across all the runs; in the remaining cases, MEM-FAST outputs a test dependency graph with less dependencies than PRADET and PFAST. This is because MEM-FAST tries to build valid schedules of minimal length, resulting in a test dependency graph with a smaller number of dependencies.

Columns 12–14 of Table 4, show the execution time (in seconds) of the parallelized schedules produced by PRADET, PFAST and MEM-FAST, while Column 11 shows the sequential execution of the test suite. We observe that, based on the median, PRADET and PFAST achieve comparable reduction w.r.t. the sequential execution for all the subjects (i.e., 1.5× saving). MEM-FAST, on the other hand, has an advantage w.r.t. PRADET and PFAST (i.e., a 2× saving), since MEM-FAST's parallelized schedules are shorter.

MySQL Test Suites. Considering MySQL system-level test suites, all techniques output the same test dependency graph in all runs, i.e., a test dependency graph with no dependencies (it should be noted that a-priori one cannot assume the absence of dependencies in any stateful, complex system such as MySQL).

When executing MySQL test suites with Valgrind, we observe a significant reduction of the execution time w.r.t. the sequential execution. Such reduction is equivalent for all dependency detection algorithms, i.e., PRADET, PFAST and MEM-FAST, as they all output the same test dependency graph (the average and the median of PRADET differ from those of PFAST and MEM-FAST because they are computed on five less subjects, while the average and the median of PFAST and MEM-FAST are approximately the same). Even when considering test suites executed without Valgrind, namely *gcol* and *innodb_zip*, the reduction is significant, ranging respectively from 1.7× to 3.6×. Moreover, the reduction is much more pronounced than with Web test suites, ranging from 2 (*gcol*) to 20× (*collations*). Practically, the execution of the test suite is reduced from 1 hour to 10 minutes, which allows for a significant time saving for each regression run. Moreover, when considering PFAST and MEM-FAST, the time saving justifies the cost of executing dependency detection, which respectively takes, based on the median across all subjects, 43 minutes and 5 minutes, respectively. In both cases, a single test suite execution is sufficient to offset and start gaining from the cost of de-

pendency detection. However, this is not true for PRADET, where dependency detection takes more than 24 hours for more than half of the subjects.

RQ2 (Test Execution): Overall, the schedules generated by PRADET, PFAST and MEM-FAST are comparable both in terms of length, for synthetic tests, and in terms of execution time, for real system-level test suites. The reduction w.r.t. the sequential execution for Web test suites ranges from 1.5× to 2×. However, PFAST achieves the same reduction by detecting dependencies up to orders of magnitude faster than PRADET. For MySQL test suites, the reduction w.r.t. the sequential execution is even more pronounced, reaching a 6× saving, lowering the execution time from 1 hour to 10 minutes. A single test execution offsets the dependency detection cost of PFAST and MEM-FAST, while PRADET requires many more executions, since it takes more than 24 hours in most of the subjects.

4.4 Threats to Validity

The limited number of subjects we used, poses an *external validity* threat. Although more subjects are needed to fully assess the generalizability of our results, we used both synthetic and real system-level test suites to evaluate our approach. Synthetic test cases allow studying how the dependency detection algorithms scale when varying the test suite size. In such context, we used three different categories of random graph generation algorithms to generate test dependency graphs with different structures. For system-level test suites, we selected two types of systems, i.e., Web applications and a database management system, namely MySQL. For Web applications, we considered six open-source test suites used in previous Web testing research, where Web applications spanning different domains were tested. For MySQL, we selected eight test suites with varying test suite sizes and types of tests (i.e., test suites with and without "big" tests (see Table 2)).

The flakiness of test cases poses an *internal validity* threat, especially in Web test suites. We addressed this issue by manually adding appropriate delays in Web test suites before running the dependency detection algorithms. Moreover, we executed each approach, both for Web and MySQL test suites ten times to account for the possible flakiness introduced by the parallelized execution of the schedules.

With respect to *reproducibility*, we make our replication package publicly available, ensuring our evaluation is repeatable and our results reproducible. In particular, we distribute in different repositories the tools to replicate our experiments with synthetic tests [23], and those to replicate the experiments on real system-level tests suites [24], [25].

5 RELATED WORK

The problem of test dependency has been extensively studied in the literature, especially within the context of the test flakiness problem [26], [27], [28], [29], [30], [31], [32], [22], [33], [34]. Indeed, a recent survey by Parry et al. [22] reports that order-dependent tests constitute up to 16% of flaky bug reports, while Luo et al. [32] empirically observed that test dependency is among the three most commonly observed categories of flaky tests.

Researchers have proposed different heuristics to automatically detect dependencies in a test suite [4], [35], [2], [36]. Bell and Kaiser [35] proposed the tool VMVM, that isolates unit tests by resetting the state of the program under test before each test execution. The output of their approach is just a flag, signalling the occurrence of a test dependency problem: no TDG is computed. Zhang et al. [2] developed DTDETECTOR, which detects dependencies by exhaustively executing all possible sequences of k tests, a parameter configurable by the user. The tool ELECTRICTEST by Bell et al. [36] extracts the set of dependencies between tests by conducting a data-flow analysis to identify read and write operations on the Java objects shared between tests. PRADET [4] uses ELECTRICTEST to extract an approximated set of dependencies, which are then validated individually with dynamic analysis. The work of Li et al. [37], and Wei et al. [38] is also related to what we present, as it concerns approaches to detect order-dependent tests.

Our approach follows the same principles of PRADET, i.e., validating dependencies by executing selected schedules. However, our approach is designed for system-level tests, which makes all existing approaches inapplicable, because exhaustive execution of all k-bounded sequences would be too expensive and data-flow analysis cannot be carried out on large Web applications. In fact, in system-level Web testing the state of the application is distributed across multiple tiers/components, and it is generally difficult to control and trace it. For instance, static data-flow analysis across multiple tiers is infeasible in practice. Hence, we resort to dynamic dependency detection algorithms, which either incrementally add the dependencies associated to each test (PFAST) or directly create a set of parallelizable schedules (MEM-FAST). Detecting order-dependent tests also differs from our ultimate goal in a fundamental way. Indeed, this line of work is intended to find *some* dependencies between tests (binary order relations). By contrast, our goal is to obtain a maximally parallel schedule for the tests, which must be consistent with *all* the dependencies.

In the field of system-level Web test suites, Biagiola et al. [5] proposed TEDD, which uses NPL and string analysis techniques to filter the likely false dependencies of an approximated test dependency graph. More recently STILE by Olianas et al. [16], [39] addressed the problem of parallelizing the schedules extracted from a test dependency graph, respecting the dependencies between tests and, at the same time, avoiding repeated executions of the same test sequence prefixes. Similarly to Biagiola et al. [5] and Olianas et al. [16], [39], we focus on end-to-end Web test suites, but contrary to TEDD [5], our approach does not make any assumption on the quality of the test names. Moreover, we address the problem of detecting dependencies in end-to-end Web test suites, while STILE [16], [39] starts from a validated test dependency graph.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced two novel test-dependency detection algorithms for system-level test suites, namely PFAST and MEM-FAST. We evaluated the practical benefits of such approaches using existing systems and their associated system-level test suites (e.g., Web applications

and a database management system like MySQL). We also assessed their asymptotic behaviors using synthetic test dependency graphs, covering a diverse range of graph structures. Our results show that PFAST, PRADET, and MEM-FAST achieve comparable degrees of parallelization for both synthetic and real end-to-end Web test suites. However, PFAST is significantly faster than PRADET in the dependency detection task both asymptotically, when executed on synthetic test dependency graphs, and practically, with real Web test suites, taking, on average, one order of magnitude less time. Moreover, MEM-FAST significantly *outperforms* both PRADET and PFAST when the underlying test dependency graph is extremely sparse, as in sparse synthetic graphs and in MySQL test suites.

In our future work, we plan to combine the benefits of MEM-FAST and PFAST, by implementing a hybrid algorithm that heuristically detects whether the test dependency graph is sparse and triggers the most efficient algorithm. For instance, in a continuous integration process, developers could run PFAST to determine the sparsity of the dependencies given the initial version of the test suite. If the dependency graph is sparse, developers should use MEM-FAST in subsequent releases to update the dependency graph when the test suite changes. We also plan to implement a distributed version of PFAST, to further improve its efficiency.

ACKNOWLEDGEMENTS

This work was supported in part by Swiss National Science Foundation with project Toposcope (grant n. 214989).

REFERENCES

[1] F. Dobslaw, R. Wan, and Y. Hao, "Generic and industrial scale many-criteria regression test selection," *J. Syst. Softw.*, vol. 205, p. 111802, 2023.
[2] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, (New York, NY, USA), pp. 385–396, ACM, 2014.
[3] J. Bell, G. E. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.
[4] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pp. 1–11, IEEE Computer Society, 2018.
[5] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web test dependency detection," pp. 154–164, ACM, 2019.
[6] "Valgrind.." https://valgrind.org/info/, 2025.
[7] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.
[8] P. L. Erdos and A. Rényi, "On random graphs. i.," *Publicationes Mathematicae Debrecen*, 2022.
[9] "Simple, web-based address and phone book ." https://sourceforge.net/projects/php-addressbook/, 2017.
[10] "Claroline. open source learning management system.." https://github.com/claroline/Claroline, 2019.
[11] "A fully functioning Node.js shopping cart with Stripe, PayPal, Authorize.net, PayWay, Blockonomics, Adyen, Zip and Instore payments.." https://github.com/mrvautin/expressCart, 2023.
[12] "Mantis Bug Tracker (MantisBT) ." https://github.com/mantisbt/mantisbt, 2014.
[13] "The Meeting Room Booking System (MRBS) is a PHP-based application for booking meeting rooms.." https://github.com/meeting-room-booking-system/mrbs-code, 2024.
[14] "Php password manager." https://github.com/pklink/ppma, 2017.
[15] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 272–281, 2013.
[16] D. Olianas, M. Leotta, F. Ricca, M. Biagiola, and P. Tonella, "STILE: a tool for parallel execution of E2E web test scripts," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 460–465, 2021.
[17] D. Olianas, "STILE test suite parallelizer replication package," 2023. GitHub repository.
[18] B. Garcia, *Hands-On Selenium WebDriver with Java*. O'Reilly Media, Inc., 2022.
[19] A. Danial, "cloc: Count Lines of Code." https://github.com/AlDanial/cloc, 2023. Accessed: 11-12-2023.
[20] "MySQL Server, the world's most popular open source database.." https://github.com/mysql/mysql-server, 2025.
[21] "mysql-test-run.pl — Run MySQL Test Suite.." https://dev.mysql.com/doc/dev/mysql-server/9.1.0/PAGE_MYSQL_TEST_RUN_PL.html, 2025.
[22] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.
[23] "End-to-End testing synthtetic experiments." https://github.com/pako-23/gtdd-synthetic-tests-simulator, 2024. Accessed: 18-03-2024.
[24] "GTDD." https://github.com/pako-23/gtdd, 2024. Accessed: 18-03-2024.
[25] "End-to-End web test suites experiments." https://github.com/pako-23/gtdd-benchmarks, 2024. Accessed: 18-03-2024.
[26] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, (Piscataway, NJ, USA), pp. 233–242, IEEE Press, 2017.
[27] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, (Piscataway, NJ, USA), pp. 39–48, IEEE Press, 2015.
[28] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, (New York, NY, USA), pp. 821–830, ACM, 2017.
[29] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, (New York, NY, USA), pp. 197–207, ACM, 2017.
[30] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 433–444, ACM, 2018.
[31] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation*, ICST '16, pp. 80–90, April 2016.
[32] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), pp. 643–653, ACM, 2014.
[33] S. Habchi, M. Cordy, M. Papadakis, and Y. L. Traon, "On the use of mutation in injecting test order-dependency," *CoRR*, vol. abs/2104.07441, 2021.
[34] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020* (S. Khurshid and C. S. Pasareanu, eds.), pp. 298–311, ACM, 2020.
[35] J. Bell and G. Kaiser, "Unit test virtualization with vmvm," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 550–561, ACM, 2014.
[36] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of*

APPENDIX

PFAST and MEM-FAST are agnostic w.r.t. the types of tests, hence they are also applicable to unit tests. However, the parallelization enabled by dependency detection has less impact than with system-level tests, as unit tests execute quickly. For this reason, the focus of this work is on system-level testing, where parallelized schedules result in a considerable time saving during regression runs.

For completeness, we also evaluated PFAST and MEM-FAST on JUnit tests used to evaluate the original version of PRADET [4]. The results are in Table 5. Specifically, when comparing PFAST, MEM-FAST, and PRADET on JUnit tests, we used our version of PRADET, which is the black-box variant that does not build a preliminary test dependency graph through dynamic data flow analysis during test execution. We chose to compare the algorithms in this way because PFAST and MEM-FAST do not use white-box information when detecting dependencies, thus for a fair comparison, PRADET should also be treated as a black-box method.

In practice, we took each subject listed in the replication package of PRADET [40], and we executed the JUnit test suite using the reference order (i.e., the order considered by `maven` when executing the command `mvn test`). We then excluded the subjects whose test suite failed when executed using the reference order, namely *crystalvc*, *dynoptic*, *togglz*, *joda-time* and *dstop* (i.e., 5 out of 18 subjects). Indeed, PFAST and MEM-FAST assume that the target test suite executed in the reference order passes, such that dependencies can be confirmed when there is a failure. For the remaining subjects, we executed PRADET, PFAST, and MEM-FAST once with a 48 hours budget, the same used by Gambi et al. [4].

Table 5 shows the results for RQ1 and RQ2 when considering the JUnit subjects. Columns 2–4, show the number of dependencies found by each dependency detection technique. Overall, the algorithms find approximately the same number of dependencies, and the dependency graphs of such JUnit subjects are very sparse, confirming the results of Gambi et al. [4]. In terms of number of schedules runs and dependency detection time (RQ1, columns 6–11), PFAST and MEM-FAST significantly outperform PRADET, which also runs out of budget for 5 subjects out of 13 (indicated with the "*OOB*" symbol). In particular, PFAST generates from 7.6 (*photoplatform-sdf*) to 70× (*jackson-core*) less schedules than PRADET (when PRADET does not time out); similarly, MEM-FAST generates from 9.4 (*xmlsecurity*) to 160× (*indextank-engine*) less schedules than PRADET. In terms of dependency detection time, PFAST takes 53× less time than PRADET (considering a dependency detection time of 48h in the 5 subjects in which PRADET times out) based on the median across all subjects, while MEM-FAST takes 127× less time.

As the dependency graphs are very sparse, MEM-FAST is very efficient in this context, although PFAST remains a competitive alternative. In terms of test execution (RQ2) Columns 12–15 show the execution time of the sequential execution, and of the parallelized execution of the schedules generated by PRADET, PFAST and MEM-FAST respectively. The parallelized schedules execute 3× faster, based on the median across all subjects, although the absolute difference is small (6 seconds vs 2 seconds) as JUnit tests are computationally inexpensive.

Table 5: Dependency Detection (RQ1) and Test Execution (RQ2) for JUnit subjects used in the PRADET paper [4]. The symbol "*OOB*" means that a specific technique exhausted its 48 hours budget as set by Gambi et al. [4] (when computing the median we did not consider the rows with the *OOB* symbol). We excluded the subjects *crystalvc*, *dynoptic*, *togglz*, *joda-time*, and *dstop*, as their respective test suites fail when executed using the reference order.

		RQ1 (Dependency Detection)									RQ2 (Test Execution)			
		# Deps. Found			# Schedules Runs			Dependency Detection Time (s)			Execution Time (s)			
Subject	# Tests	PRADET	PFAST	MEM-FAST	PRADET	PFAST	MEM-FAST	PRADET	PFAST	MEM-FAST	Sequential	PRADET	PFAST	MEM-FAST
xmlsecurity	92	4	4	4	4,186.0	191.0	444.0	8,305.0	262.0	421.0	10.7	4.9	4.2	4.8
photoplatform-sdf	31	2	0	0	465.0	61.0	31.0	5,754.0	238.0	111.0	14.5	10.6	10.3	11.0
DiskLruCache	61	0	0	0	1,830.0	121.0	61.0	5,923.0	109.0	49.0	3.7	2.2	2.1	2.2
Bateman	76	0	0	0	2,850.0	151.0	76.0	3,985.0	117.0	60.0	1.6	1.4	1.3	1.3
Bukkit	429	*OOB*	0	0	*OOB*	857.0	429.0	*OOB*	755.0	297.0	2.3	*OOB*	2.1	2.6
webbit	131	0	0	0	8,551.0	262.0	131.0	21,950.0	941.0	93.0	6.3	3.1	2.9	3.2
stream-lib	140	*OOB*	0	0	*OOB*	280.0	140.0	*OOB*	3,227.0	127.0	148.7	*OOB*	35.4	36.7
http-request	163	2	2	0	13,203.0	326.0	163.0	168,789.0	558.0	126.0	33.0	1.9	1.9	1.8
jackson-core	282	0	0	0	39,621.0	563.0	282.0	131,470.0	615.0	197.0	5.4	2.0	2.0	2.1
jsoup	526	*OOB*	1	1	*OOB*	1,053.0	798.0	*OOB*	961.0	535.0	2.1	*OOB*	2.4	2.2
dynjs	865	*OOB*	0	0	*OOB*	1,729.0	865.0	*OOB*	3,788.0	665.0	7.3	*OOB*	2.8	2.8
indextank-engine	61	0	0	0	9,868.0	268.0	61.0	40,190.0	371.0	51.0	6.0	3.7	3.7	3.6
okio	727	*OOB*	0	0	*OOB*	1,454.0	727.0	*OOB*	10,162.0	490.0	145.9	*OOB*	41.8	42.8
Median		*0*	*0*	*0*	*6,368.5*	*280.0*	*163.0*	*15,127.5*	*615.0*	*127.0*	*6.3*	*2.7*	*2.8*	*2.8*