2018 IEEE 11th International Conference on Software Testing, Verification and Validation

Accelerating Search-based Program Repair

Ben Mehne
University of California, Berkeley
Email: bmehne@cs.berkeley.edu

Hiroaki Yoshida, Mukul R. Prasad
Fujitsu Laboratories of America, Inc.
{hyoshida, mukul}@us.fujitsu.com

Koushik Sen
University of California, Berkeley
Email: ksen@cs.berkeley.edu

Divya Gopinath
CMU Silicon Valley
Email: divyag1@andrew.cmu.edu

Sarfraz Khurshid*
The University of Texas at Austin
Email: khurshid@ece.utexas.edu

***Abstract*—Automatic program repair techniques offer the possibility of reducing, or even eliminating, the substantial manual effort that currently goes into the patching of software defects. However, current repair techniques take minutes or hours, to generate rather simple repairs, severely limiting their practical applicability. Search-based program repair represents a popular class of automatic repair techniques. Patch compilation and test case execution are the dominant contributors to runtime in this class of repair techniques. In this work we propose two complementary techniques, namely *Location Selection* and *Test-Case Pruning*, to improve the efficiency of search-based repair techniques. *Location Selection* reduces the number of repair candidates examined in arriving at a repair, thereby reducing the number of patch compilations as well as the overall number of test case evaluations during the repair process. *Test-Case Pruning*, on the other hand, optimizes the number of test cases executed per examined candidate. We implement the proposed techniques in the context of SPR, a state-of-the-art search-based repair tool, evaluate them on the GenProg benchmarks and observe that the proposed techniques provide a 3.9X speed-up, on average, without any degradation in repair quality.**

I. Introduction

Debugging and patching of software defects is a laborious, largely manual activity, consuming a disproportionately large fraction of software development resources [1]. The relatively nascent body of research on automatic program repair (APR) [2–4] offers the promise of reducing the manual burden associated with patching bugs. However, current repair techniques take on the order of tens of minutes if not hours to find a repair and in several cases cannot completely search the space of fairly simple, one-line repairs even in 12 hours of compute time [3–5]. This factor alone limits expanding the scope of APR techniques to richer repair spaces and indeed the viability of APR techniques in real-world debugging scenarios. Therefore, we believe that in order for APR techniques to have practical impact we need to dramatically improve the efficiency of automatic program repair. This paper makes a contribution in that direction. Specifically, we propose techniques in the context of SPR [6], a state-of-the-art search-based repair tool. Our proposed techniques provide a 3.9X speed-up, on average, without any degradation in repair quality.

Search-based program repair techniques, or *generate-and-validate* (G&V) repair techniques as they are sometimes called [6], represent a popular class of automatic repair techniques. Given a buggy program P, failing at least one test in a test suite T, the repair tool searches the space of mutations to P, defined by a given set of repair templates (*generate* phase) for one that allows the mutated program to pass all the tests in T (*validate* phase). We present our approach in the context of this class of repair techniques.

Several researchers have raised the issue of the efficiency of current APR techniques [5–7], in particular, the fact that test case execution dominates the runtime of G&V repair techniques [8, 9]. However, in order to establish our own empirical basis for our research we performed the following study. We did the study using the SPR repair tool [6], a reasonably recent representative of G&V repair techniques for C programs, on a sample of 7 bugs from the GenProg benchmarks [2]. We chose one bug each from 7 of the 8 subject systems in the benchmarks (except fbc, which does not work on 64-bit systems), randomly chosen from the set of instances for each subject system, where SPR produced a patch. We profiled the SPR runs for the fraction of the runtime consumed by various facets of repair (excluding fault localization), such as test case execution, compilation of patched program instances, *etc*. The data shows that the runtime is dominated by the time for patch compilations and test executions, which consume, respectively, 4.4—49.8% and 20.9—91.9% of the runtime. Furthermore, the relative distribution varies significantly with subject systems (*e.g., lighttpd-1948-1949* used 4.4% and 91.9% for compilations and test executions, respectively, while *libtiff-d13be7c-ccadf48a* used 49.8% and 39.8% for compilations and test executions, respectively). Thus, an effective strategy for accelerating repair should address the compilation time as well as the time for test executions.

In this work we propose two complementary techniques, namely *Location Selection* and *Test-Case Pruning*, to improve the efficiency of G&V program repair techniques. *Location Selection* reduces the number of repair candidates examined in arriving at a repair, thereby reducing the number of patch compilations as well as the overall number of test case evaluations during the repair process. *Test-Case Pruning*, on the other hand, optimizes the number of test cases executed per examined candidate. Of course, much of the previous work in search-based APR directly or indirectly improves the underlying efficiency of repair as well. However, with a few exceptions [8], the focus of previous work has been to maximize the *fix-rate* of the technique, *i.e.,* the number of instances in which patches are produced, by using different search algorithms [2, 10], specialized schemas [11, 12], better orchestrations of those schemas [6], or by re-organizing the search space [3, 5]. More recent work addresses patch quality [13, 14]. By contrast, repair efficiency is the singular, driving concern of this work. In fact, the optimizations we propose are largely complementary to previous work on G&V program repair and arguably generic enough to work with any search-based APR technique.

Our *Location Selection* optimization is based on a heuristic state comparison. The intuitive reasoning behind our technique is as follows. Assuming a failing test case trace exhibits partially incorrect behavior (*i.e.,* it is incorrect at some of the program locations it executes), the locations at which the behavior is actually *correct*, may *not* be good candidate locations for affecting a repair. Further, if we assume that passing test case traces represent substantially correct behavior, program locations at which failing test cases and passing test-cases, in aggregate, exhibit substantially *similar* behavior, are the locations that are poor candidates for repair. Our proposed technique

* This author's research was partially supported by the US National Science Foundation under Grant Nos. CCF-1704790 and CCF-1718903.

uses a heuristic comparison of program states at a given location, under failing and passing tests, to identify such poor repair locations, and de-prioritize them for repair. Our *Location Selection* optimization is loosely inspired by the work on Delta Debugging [15–17], which views a bug as an "infection" or corruption in the program state. By iteratively and systematically mutating the program state under a failing test case and comparing it with that of a passing test at the same location, delta debugging attempts to precisely localize a defect as well as the input or state variables responsible for it. Our optimization employs state comparisons too but in a different manner, and for the purpose of accelerating program repair.

Our *Test-Case Pruning* technique is motivated by classical *regression test selection* (RTS) [18–22], a somewhat natural idea to pursue, given that G&V repair entails repeated executions of a test suite. However, current APR techniques typically evaluate several hundreds of repair candidates in a single repair run, each of which is typically a single-statement modification. Standard RTS techniques, which are designed to work with arbitrary program changes, would either be too coarse—it may track changes at source file or method level—or too expensive, to be useful in the current context. Our technique of *Test-Case Pruning* tracks test-case-to-source dependencies at a source line level which allows it to effectively prune test cases. Further, it derives this dependency information, with minimal overhead, from the fault localization step that typically precedes the repair run.

The main contributions of this paper are as follows:

- **Observation:** We make an empirical-study driven observation that patch compilation and test executions together dominate the runtime of modern G&V repair tools, partially corroborating the claims of previous research [8, 9].
- **Optimizations:** We propose two complementary techniques, *Location Selection* and *Test-Case Pruning*, to accelerate G&V program repair techniques, by reducing the time spent on compilation and test-case executions.
- **Implementation:** We implement the proposed techniques in the context of SPR [6], a state-of-the-art G&V repair tool.
- **Evaluation:** We evaluate the proposed techniques on 43 bugs from the GenProg benchmarks [2] and observe that the proposed techniques provide a 3.9X speed-up, on average, without any degradation in repair quality.

II. Motivating Example

To describe the *Location Selection* and *Test-Case Pruning* techniques, we first describe the behavior of generate and validate (G&V) program repair, and then the specific ways in which our techniques improve this behavior. To make the descriptions of this behavior concrete, we use a motivating example, described next.

A. Benchmark

During the development of our techniques, we focused on the runtime devoted to compilation and the runtime devoted to test case evaluation over several benchmarks. We use the benchmark *php-307846-307853* as a motivating example to illustrate the proposed techniques. Figure 1 presents the source code where the correct patch is applied. The bug is in the function `date_isodate_set`. In this function, a date object is constructed incorrectly: the implementation does not appropriately initialize some of the fields. The bug fix is to initialize all fields to zero with a memset. The code for the buggy `date_isodate_set` function is provided with the bug fix in a comment on the appropriate line. The implementation of our techniques produces a correct patch, as does the unmodified G&V program repair system we are implemented on top of.

```
PHP_FUNCTION(date_isodate_set)
{
  zval          *object;
  php_date_obj *dateobj;
  long           y, w, d = 1;

  if (zend_parse_method_parameters(/* ellided */,
      &object, date_ce_date, &y, &w, &d) == FAILURE) {
    RETURN_FALSE;
  }
  dateobj = (php_date_obj *)
      zend_object_store_get_object(object TSRMLS_CC);
  DATE_CHECK_INITIALIZED(dateobj->time, DateTime);
  dateobj->time->y = y;
  dateobj->time->m = 1;
  dateobj->time->d = 1;
  // Bug fix: memset(&dateobj->time->relative, 0,
      sizeof(dateobj->time->relative));
  dateobj->time->relative.d =
      timelib_daynr_from_weeknr(y, w, d);
  dateobj->time->have_relative = 1;

  timelib_update_ts(dateobj->time, NULL);

  RETURN_ZVAL(object, 1, 0);
}
```

Fig. 1. Buggy function from *php-307846-307853*

B. Generate & Validate Program Repair

To understand how our techniques accelerate the repair process, we must first discuss a generic G&V program repair system. G&V program repair systems operate in three phases: fault localization, generation, and validation.

The first stage of G&V repair systems is fault localization: the process of determining locations where the bug may reside. Generally this is performed by profiling specific characteristics of locations during test case evaluation. Spectrum-based fault localization techniques [23, 24] are widely used to rank such a series of locations based on their suspiciousness. In the motivating example, all locations within the function in Figure 1 are selected during the fault localization process, among approximately 100 more.

The second phase of GV program repair is to generate patches for one or more of the locations selected by the fault localization phase. The generation phase can evaluate the test cases or use additional information (such as whether a patch seems likely to be written by a developer) to prioritize or de-prioritize locations and patches. For *php-307846-307853*, the correct patch is generated after nearly two thousand others.

The third phase of GV program repair is the validation phase. The validation phase uses all test cases to show that the patch is valid—if all test cases pass with the patch, then the patch is considered validated. If no patch is valid, then G&V program repair can either return to the generation phase or, if there are no more patches to be generated, the process can complete with failure. *php-307846-307853* provides a test suite of nearly 7000 tests. The vast majority of these test cases do not detect the failing behavior.

Our work focuses on accelerating both the generation phase and the validation phase. The generation phase is accelerated by providing prioritization for locations that are particularly favorable for patching. For *php-307846-307853*, we specifically prioritize the line of the patch and not lines earlier in the function. The validation phase is accelerated by eliminating unnecessary test case evaluations. For *php-307846-307853*, this leads to a 6X reduction in test cases evaluated.

C. Test-Case Pruning

Here we introduce the *Test-Case Pruning* technique, which is inspired by regression test selection. The purpose of *Test-Case Pruning* is to minimize the cost of validating a patch. To do this, we identify a subset of test cases that cover only the changed locations in a program. In particular, for any location where a patch can be applied, we use profiling information from the localizer to identify the test cases that cover that location. For any location that is changed, that change can only effect test cases that cover that location. The set of all test cases that exercise at least one of the changed locations will contain all the test cases effected by the patch.

For the *php-307846-307853* benchmark, exactly 15 of the 6951 test cases cover some part of the function in Figure 1, and only 12 cover the location where the patch is applied. This information is obtained by profiling the coverage of the test cases. Using the exact test case coverage for a patch location, allows us to safely not run the vast majority of test cases during patching. For comparison, in the ideal case for generic G&V program repair, the "correct" patch would be generated on the first attempt and then validated. The validation, in this case, would use all test cases. This would mean that the generic G&V program repair would use over 400x more test case evaluations than are necessary to show that the patch is valid and over 2X what our technique uses to validate *all* patches (as discussed in Section IV).

In practice, G&V program repair systems produce multiple invalid patches which each require the use of one test case to invalidate the patch. While more patches will always require more test case evaluations and will decrease the effective acceleration of this technique, this technique will prevent unnecessary test case evaluations in the case that (1) a "correct" patch is found and (2) a test case is used to invalidate a patch that is not affected by that patch.

D. Location Selection

Here we describe the *Location Selection* technique. This technique aims to accelerate the generation phase of G&V program repair. To do this, we prioritize locations for patching by analyzing the values of the variables that are "live" at that location. We define *live* to be variables that have been written to and are otherwise in-scope. In particular, we analyze the range of values that occur during the positive test cases and compare that to the range of values that occur during the negative test cases. If the range of values during negative test cases is a subset of the range of values for positive test cases, then we consider the location to be a worse location for a patch. If the range of values during the negative test cases is not a subset of the ranges from the positive test cases, then we consider this location to be a better location for a patch. We consider all locations that we deem better for patching before any location we deem worse. We will now use Figure 1 to describe two cases: one in which a location is prioritized and one in which a location is de-prioritized.

To understand how this works for prioritizing locations, let us consider line 15 in Figure 1. This line is where the patch should be applied. In order to determine whether this location is suitable for patching by the *Location Selection* technique, we must log the values of the *live* variables at this location. As discussed in Section III-E, we use a subset of '*live* variables, focusing on variables that are used nearby. For purposes of exposition, let us assume that we have selected the variable `dataobj` and let us discuss the fields in `dateobj->time->relative` that are not correctly initialized. In particular, the `dateobj->time->relative.weekday_behavior` field, which is an integer field, causes this location to be prioritized. During the negative test case[1], this function produces an incorrectly initialized date object: the field of `dateobj->time->relative.weekday_behavior`, among others, is not set correctly. The value of `dateobj->time->relative.weekday_behavior` is non-zero for the negative test case. The incorrect output of the negative test case is an incorrect week number (the week number in the year, *e.g.,* the 2nd week of the year). This value is calculated via the `timelib_update_ts` function on line 19. This function uses different fields of `dateobj->time->relative`, including `weekday_behavior` to complete this calculation. If `weekday_behavior` is non-zero, then the calculation will fail and the results of the function call will be incorrect. During passing test cases for PHP, the value can range between zero and two. Because the value of this field is uninitialized during the negative test case, the values for `dateobj->time->relative.weekday_behavior` during the negative test case are pointer values that happen to exist at that memory location[2]. No value in this range is in the range of values from the positive test cases. We, therefore, consider this location as a better location for a patch. Note that if this field was zero during the negative test case, the range would be a subset of the range of values for the positive test cases, and the program would not have produced the wrong output. This technique works especially well for uninitialized values.

To continue the discussion, let us consider a location that was de-prioritized by our algorithm—line 9, after the curly brace. This is the last line before `dateobj` becomes "live". At this location, there are a number of variables in scope: global variables, `object`, `dateobj`, and the `long`-type variables `y,w,` and `d`. Our technique selects the variables that are "live"—they have been written to—and are otherwise used in the function (we don't use global variables that are only used elsewhere). This selection includes the `long`-type variables and the `object` variable. For purposes of discussion, let us consider only the `long`-type variable `y`—the remaining variables behave similarly. In the negative test case, the value for `y` is `2005`. For the positive test cases for *php-307846-307853*, the values for `y` are `-12345, -10, 0, 1, 10, 1963, 2006, 2008, 2009, 2010,` and `12345`. First, we should note that the value `2005` is not in this set, but it is still a value that can be used without causing a bug to occur. We use a *summary* of values in order to enable our technique to solve this issue—to overcome under-specification due to the sparseness of test cases. We compare the value `2005` to the range of values obtained in the positive test cases: `-12345` to `12345`. Since it is in that range, we conclude that this variable does not make this location suspicious. The remaining variables at this location exhibit similar behavior (the remaining `long`-type variables have the exact same range, as `12345` was used as a sentinel in testing). Since no variable makes this location suspicious, we de-prioritize it. Logically, we are asserting that there are no variables at this line that are immediately causing the bug. This technique works especially well with variables that can store any value from a continuous-range and have test cases that cause many values in that range to occur.

III. Techniques

In this section we describe both techniques more formally. While both of the techniques are distinct, we use one algorithm to perform both in tandem. First, we describe a high-level view of G&V repair, and then we describe the algorithm in detail.

[1]This benchmark only has a single negative test case.

[2]No pointers between zero and two are valid in our experimental setup.

Algorithm 1 Algorithm for validating patches

```
1: function RepairAlgorithm($L$, $\mathcal{G}$, $T_P$. $T_N$, $P$)
2:     $S \leftarrow$ PreProcess($L$, $T_P \cup T_N$, $P$)
3:     for all $l \in$ GetOrder($L$, $S$, $T_P$, $T_N$) do
4:         for all $I \in \mathcal{G}(P, L')$ do
5:             for all $t \in$ GetTestSetForPatch($I$, $S$) do
6:                 if $Fails(RunProgram(P, I, t))$ then
7:                     go to 4
8:             return $I$
9:     return $\perp$
```

Algorithm 2 Algorithm for filtering test cases and candidate locations

```
1: function GetOrder($L$, $S$, $T_P$, $T_N$)
2:     $\hat{S}_N \leftarrow$SummarizeForTestSet($T_N$, $L$, $S$)
3:     $\hat{S}_P \leftarrow$SummarizeForTestSet($T_P$, $L$, $S$)
4:     $L_S \leftarrow$ empty list, $L_O \leftarrow$ empty list
5:     for all $l \in L$ do
6:         if $\hat{S}_N[l] \not\leq \hat{S}_P[l]$ then
7:             $L_S \leftarrow L_S || l$
8:         else
9:             $L_O \leftarrow L_O || l$
10: function SummarizeForTestSet($T$, $L$, $S$)
11:     $\hat{S} \leftarrow$ empty map
12:     for all $t \in T$ do
13:         for all $l \in L$ do
14:             $\hat{S}[l] \leftarrow \hat{S}[l] \sqcup S[t, l]$
15:     return $\hat{S}$
16: function GetTestSetForPatch($I$, $S$)
17:     $testSet \leftarrow$ empty set
18:     for all $l \in ChangedLocations(I)$ do
19:         $testSet \leftarrow testSet \cup \{t | S[t, l] \neq \perp\}$
20:     return $testSet$
21: function PreProcess($P, L, T$)
22:     $I \leftarrow$GetInstrumentationPatch($P$, $L$)
23:     $S \leftarrow$ map from all values $t \in T, l \in L$ to $\perp$
24:     for all $t \in T$ do
25:         for all $\sigma, l \in LoggedStates(RunProgram(P, I, t))$ do
26:             $S[t, l] \leftarrow S[t, l] \sqcup \sigma$
27:     return $S$
28: function GetInstrumentationPatch($P$,$L$)
29:     $p, n \leftarrow P$
30:     $p' \leftarrow$ empty map, $n' \leftarrow$ empty map
31:     for all $l \in L$ do
32:         $l_t \leftarrow$ a new location
33:         $n'[l_t] \leftarrow n[l]$
34:         $p'[l_t] \leftarrow p[l]$
35:         $varsToLog \leftarrow$LoggingVars(l, $P$)
36:         $s' \leftarrow createLogStmt(l, varsToLog)$
37:         $p'[l] \leftarrow s'$
38:         $n'[l] \leftarrow \{l_t\}$
39:     return $p', n'$
```

A. Generate & Validate Repair

The problem of automated repair is to generate patches to programs such that all test cases pass. Our techniques work to increase the efficiency of existing repair systems that use a generic generate and validate (G&V) mechanism. To describe our techniques, we first must define a high-level view of G&V automated repair.

For our purposes, the following definitions suffice:

- A program P is a pair of two mappings: n and p, where p maps a location l to a statement s, and n maps a location to a set of the next locations that can be executed. This is inspired by [6].
- A patch I is a pair of two mappings n and p, which are a subset of their respective mappings in a program. This subset overrides the mappings in the program. For instance, if a patch maps $l \rightarrow s'$ and a program maps $l \rightarrow s$, then when the patch is applied l maps to s' instead of s.
- A patch generator $\mathcal{G}$ takes a program P, and a location l and returns a list of patches I for that location. The patch generator may take additional information, but our techniques do not require that it does so. For instance, the patch generator may use the program test cases to decide on the prioritization of possible patches for a location in the list.
- A test case set T is a set of test cases. We require two test case sets: the set of all test cases that the program passes originally (the positive test case set T_P) and the set of all test cases that the program fails to pass originally (the negative test case set T_N).

Algorithm 1 describes a simple G&V repair algorithm. Here, the repair algorithm is broken up into a pre-processing step and three nested loops. Our techniques require a pre-processing step, while in a *generic* G&V algorithm, the pre-processing step does nothing (*e.g.,* PreProcess simply returns $\perp$ on line 2). The outermost of the three nested loops iterates over the locations to generate patches at. In a generic G&V repair system, the order is unchanged from the order inputted (*e.g.,* GetOrder returns the L unmodified), while our techniques provide a more-optimal order. The middle nested loop iterates over the patches for a location. In the inner-most loop, the patch is tested by running the program with the patch for a set of test cases. In a generic G&V repair algorithm, this set is the set of all test cases (*e.g.,* GetTestSetForPatch returns a set of all test cases). If the patch fails a test case, then a new patch is generated and tested. If the patch successfully passes all test cases, then the patch is returned. Should no patch successfully pass all test cases, then a bottom value is returned, signifying a failure to produce a valid patch.

To describe our techniques, we provide algorithms for the GetOrder and GetTestSetForPatch functions that improve the overall run time of the repair system, using information gained through a PreProcess function call. These functions augment the repair algorithm by adding three different phases: a pre-processing phase that logs *state* information from the evaluation of test cases, a processing phase that summarizes the state information and uses it to prioritize patch locations, and a test case selection phase. These phases are discussed below, and the full algorithms is given are Algorithm 2.

B. Pre-Processing Phase

The pre-processing phase collects information to be used by the later stages by running the program under test with instrumentation. In particular, the pre-processing phase collects traces of partial *states*. A *state* σ is a mapping from variables to values during the execution of a specific program location. In order to provide the necessary instrumentation, each location where a patch may be made has a logging statement inserted in front. The logging statement logs the value of a heuristically-determined set of variables as a state. The heuristic is described in Section Section III-C1. This logic is described on lines 28 through 39 of Algorithm 2.

The instrumentation patch is then used to log states by running the program, with the patch, for all test cases. During the execution of the program with the instrumentation, the state & location pairs are recorded. This logic occurs on lines 21 through 27 in Algorithm 2. Ideally, we desire the state to give us information about its corresponding location, but a location can produce multiple states due to loops in the program. Instead of keeping a list of states for each location, we summarize the states by keeping a *summary* of values for each variable instead of a set of values. A *summary* is an abstraction of a set of values—any value that is in a set must be in the summary of that set, though values that are not in the set may also be in the summary of the set. Abstracting sets into summaries decreases the memory requirements of our algorithm, but it is not sound as the summary is necessarily an over-approximation of the values that occur in the program for that location. We require that summaries form a lattice: the join is well-defined, that the $\bot$ value represents an empty set of values, and $\top$ represents the set of all possible values. While there are a variety of summarization methods we can use, we elected to use *ranges* of values (*e.g.,* the values `1` and `3` would be summarized as the range from `1` to `3`). We call the mapping from variables to summaries a *state summary*. A *state summary* is a mapping from a variable to the summary of values it has during the execution(s) of a specific program location. In order to create state summary from the recorded states, we extend (if necessary) each summary for each variable with the corresponding value from the state. This computation is identical to the least upper bound of a lattice over state summaries. We use state summaries to implement *Location Selection*.

C. Location Selection

During the first phase, we collect the summaries of values that variables can obtain (state summaries) for each location for each test case. During this phase, we use this information to determine whether the state summaries for a location is "different" during positive test cases than during negative test cases. To do this, we first summarize the state summaries for each location for all positive test cases and construct a separate summary for each location for all negative test cases. This results in two separate mappings—each mapping is from a location to a state summary. This is performed on line 10 through line 15 in Algorithm 2. This code produces the least upper bound of all state summaries for each location for all test cases in the specified set.

Now, we have a mapping from each location to a single state summary for the positive test cases and a single state summary for the negative test cases. Let $\hat{\sigma}_P$ be the state summary for a location l for the positive test cases and let $\hat{\sigma}_N$ be the state summary for the same location for negative test cases. We then prioritize *suspicious* locations before *ordinary* locations for the purposes of patch generation. A *suspicious* location is one such that the $\hat{\sigma}_N \not\leq \hat{\sigma}_P$, which means that at least one value during the negative test cases did not occur during the positive test cases at that location. Any location that is not suspicious is *ordinary*. This operation is performed on lines 5-9 of Algorithm 2.

During patch generation, all locations are considered, but suspicious locations are prioritized above ordinary locations. Ordinary locations are still evaluated to maintain soundness of the technique: because we can discard variables via our heuristic (see Section Section III-C1) and we consider summaries instead of sets of values, discarding ordinary locations is unsound. To understand the former reason, consider a location that produces different state summaries between positive and negative runs only because of a single variable, which our heuristic happens to discard. For the latter case, consider a location that has the state summary (a range) of 1 to 5 for positive test cases but in the negative test cases the value 3 causes those test cases to fail and the value 3 does not occur during the positive test cases. In this case our *Location Selection* technique would incorrectly de-prioritize the location because the state summary is an over approximation of the values that can occur at that location. State summaries can trivially be overly broad: consider a variable that either contains the minimum value or the maximum value of its type. The summary of this variable will have a range containing all values for its type. We now discuss the selection of variables to create state summaries.

1) Variable Selection Heuristic: In the *Location Selection* technique, we construct summaries of values for variables at a location. Naively, we could construct these summaries for all variables at a location, but this is ineffective and inefficient in practice. Instead, we use the *variable selection heuristic* to select specific variables for which to construct summaries.

First, we only consider variables that are in-scope at that location and are "live". A "live" variable, for our purposes, is a variable that has already been written to (this prevents using uninitialized variables) or read from (we assume that variable is correctly initialized if it is read from). From this set of considered variables, we apply the heuristic. The heuristic is designed to eliminate variables that cannot be used by any patch and to eliminate variables that have values that cannot effectively be summarized.

The first set of variables to eliminate—variables that cannot be used by any patch—is effective because any variable that cannot be used in the patch (the patch cannot read from or write to it) is a variable that the patch is *agnostic* to. A patch is *agnostic* to the variable if the behavior of the patch is not effected by the value of the variable. A variable that cannot effect the behavior of the patch and, hence, its correctness, is not necessary to be evaluated for the *Location Selection* technique. Our heuristic removes variables that SPR's patch generation technique cannot use; *e.g.,* floating point variables.

The second set of variables to eliminate—variables that have values that cannot be effectively summarized—is necessary to prevent our technique from behaving erratically and is independent of the summarization method. To explain this concept, consider a pointer variable that obtains its value from a call to `malloc`. The range of values this pointer may obtain is any pointer value in the heap, with the correct byte alignment. A summary of this range would be, ideally, the minimum and maximum values of the portion of the heap used for memory allocation at that time in the program. Knowing that a pointer points to some specific area in the heap rarely determines whether the pointer is valid. This is both because multiple locations in the same range are invalid and code is rarely designed to require specific heap ranges for execution. Our heuristic removes these variables that cannot be effectively summarized; *e.g.,* pointer variables.

D. Test-Case Pruning

The remaining technique to implement is the *Test-Case Pruning* technique. In this technique, we construct a test case set of all test cases that covered a location that was changed in the patch. In practice, the coverage of each test case is commonly available for each location that a patch may be applied. For purposes of exposition, we will assume that no such information is available for each location and will construct the necessary information from the mappings obtained in the first phase. At the end of the first phase, we have a map from test case & location pairs to the summary of values that variables can obtain (state summaries). If the state summary is an empty range (the variables there cannot obtain any value), then we know that the location is never encountered during execution of the program for that test case. The set of all tests cases where a specific location has a non-empty range is the set of all test cases that cover that location. This set is constructed on line 17 through line 19. When a patch for that location is being validated, we skip any test case that is not in this set. This is sound because any test case that exercises the locations in the patch are in the test case set—any test case that exercises the location must produce some non-empty state summary for that location.

E. Implementation

We implement both of our techniques on top of SPR [6], modifying the source code available in its replication package. SPR is a recent generate and validate automated repair tool for C programs. SPR uses several heuristics to decrease its runtime and improve its accuracy. We now describe SPR's architecture and how it's internal heuristics interact with our techniques.

The execution of SPR can be broken into multiple phases: a fault localization phase, patch schema selection, and staged patch validation. The first phase—fault localizing—runs as a pre-processing step for SPR, using the coverage information for all test cases to determine a list of locations for patching. The resulting list is ordered based on how likely the location is the source of the bug. Our *Test-Case Pruning* technique can use the coverage information, with zero overhead, from this phase to determine the test cases for each patch location. Our *Location Selection* technique requires a separate pre-processing stage with measurable overhead (as discussed in Section IV-E).

The second phase—patch schema selection—chooses a *schema* for each of these locations. A *schema* is a generic plan for patching, like introducing an if-statement or an assignment statement. A schema does not necessarily specify a complete patch: for instance, it may specify that an if-statement is introduced, but not the condition. Once a schema is selected for each location, the location-schema pairs are prioritized based on both the position in the list from the fault localization and the schema selected. Our *Location Selection* prioritization supersedes this ordering.

The third phase—staged patch validation—creates a concrete patch from the schema in stages and validates the patch. The creation of a concrete patch from a schema sometimes involves the running of test cases to collect further information. Here, our *Test-Case Pruning* technique eliminates unnecessary test case executions. Once a concrete patch is created, the patch is added to a *batch*. A *batch* is a set of patches that are compiled at once. Patches are added to a batch until certain heuristics are met, one of which is the time since a batch was last compiled. The patch that is run during validation is selected via a run-time environment variable. This is achieved by separating the patches by a switch-case statement. Once the batch is compiled and a specific patch is selected, it is validated by running all test cases, prioritizing the test cases that last caused a patch to fail earlier. Our *Test-Case Pruning* technique alters this technique to only be test cases that could be affected by the patch. The very design of staged patch validation, decreases the number of concrete patches and thereby decreases the number of compilations and test cases evaluated.

Two of the heuristics in the third phase negatively effect the utility of our techniques: batching and the existing test-case prioritization. The batching behavior prevents *Location Selection* from effectively minimizing compilation time, as the difference of time of compilation between a batch of size one and a batch of a larger size is negligible. The existing test-case prioritization essentially simulates a simpler form of the *Test-Case Pruning* technique, because the test case that last failed is often the test case that has the most coverage.

IV. Evaluation

A. Experimental Setup

All experiments for this paper were performed on virtual machines with 2 Intel Xeon E5-2695 CPUs and 8Gb of memory allocated. The experiments used modified source code and support scripts from the SPR replication package [25] with a time limit of 12 hours. We used 43 bugs from the GenProg benchmarks [2] used by SPR.

RQ1: Can *Test-Case Pruning* and *Location Selection* working in tandem improve the efficiency of generate-and-validate program repair without degrading patch correctness? (Section IV-B)

RQ2: What is the contribution to the speedup from each of the two techniques? (Section IV-C)

RQ3: Does *Test-Case Pruning* reduce the number of test cases evaluated for each repair candidate and *Location Selection* the number of repair candidates examined to derive a repair? (Section IV-D)

RQ4: What is the overhead of applying the proposed techniques? (Section IV-E)

B. RQ1: Overall Speedup

Table I shows the patch generation results for the execution of four techniques: the original SPR (columns SPR), SPR enhanced with *Test-Case Pruning* (columns +TP), SPR enhanced with *Location Selection* (columns +LS), and SPR enhanced with both techniques (+TP+LS). The *patch correctness* columns record the correctness of the patch categorized as CR (correct patch), PL (plausible patch), or TO (timeout). As in previous work [3, 4, 6] we categorize a patch that is semantically equivalent to the developer-provided patch (established through manual comparison) as *correct*, and one that is not equivalent but nevertheless passes the test suite, as *plausible*.

The upper half of the table (termed *comparable instances*) corresponds to the 27 bugs where all four techniques produced identical patches. These include all of the 12 correct patches produced by the original SPR (bugs # $1 - 12$) and a further 15 instances with plausible patches. In these instances all four techniques are effectively searching comparable search-spaces, and can be meaningfully compared in terms of efficiency. The bottom half of the table (termed *incomparable instances*) includes a further 16 instances where all techniques ran but produced different results, *i.e.,* either different plausible patches or one or more of the techniques timing out. In these instances the techniques are effectively searching different search spaces and it is less meaningful to compare (or even define) their relative efficiency. Nevertheless, we report this data for completeness.

The columns labeled *Runtime* report the repair time for various techniques, recorded in minutes (720 minutes if a timeout occured). Note that, consistent with previous work [3, 4, 6], this time *does not* include the bug localization time. Further, it does not include our pre-processing time, which we discuss separately under RQ4. The columns labelled *Speed-up* are calculated as a ratio to the runtime of the original SPR. Again, speed-ups although calculated for the bottom half-of the table have only nominal value.

Discussion: Overall, the combined technique (+TP+LS) provides a net speed-up for almost all of the 27 comparable instances (top half of Table I), and in particular for *each* of the 12 correct instances

TABLE I
PATCH GENERATION RESULTS.

Bug		Patch Correctness				Runtime [min.]				Speed-up [X]			Pre-processing
#	ID	SPR	+TP	+LS	+TP+LS	SPR	+TP	+LS	+TP+LS	+TP	+LS	+TP+LS	Time [min.]
1	gmp-13420-13421	CR	CR	CR	CR	179.9	179.3	103.8	100.8	1.0	1.7	1.8	3.2
2	libtiff-ee2ce5b7-b5691a5a	CR	CR	CR	CR	13.1	11.3	12.5	9.6	1.2	1.1	1.4	1.4
3	php-307562-307561	CR	CR	CR	CR	251.4	184.3	300.7	420.3	1.4	0.8	0.6	5.7
4	php-307846-307853	CR	CR	CR	CR	78.2	52.3	23.5	9.6	1.5	3.3	8.2	5.3
5	php-307914-307915	CR	CR	CR	CR	45.5	45.2	46.8	40.5	1.0	1.0	1.1	4.9
6	php-308734-308761	CR	CR	CR	CR	339.3	123.0	343.5	121.2	2.8	1.0	2.8	4.7
7	php-309516-309535	CR	CR	CR	CR	92.8	51.5	60.9	18.6	1.8	1.5	5.0	4.3
8	php-309579-309580	CR	CR	CR	CR	66.5	23.1	69.9	22.5	2.9	1.0	3.0	4.6
9	php-309892-309910	CR	CR	CR	CR	84.4	36.6	118.7	10.2	2.3	0.7	8.3	5.5
10	php-310991-310999	CR	CR	CR	CR	143.1	166.2	165.2	142.5	0.9	0.9	1.0	4.3
11	php-311346-311348	CR	CR	CR	CR	64.4	21.3	48.7	5.5	3.0	1.3	11.8	4.7
12	python-69783-69784	CR	CR	CR	CR	74.0	50.8	61.8	38.4	1.5	1.2	1.9	3.6
13	gmp-14166-14167	PL	PL	PL	PL	19.0	9.3	15.8	4.7	2.1	1.2	4.1	3.0
14	gzip-a1d3d4019d-f17cbd13a1	PL	PL	PL	PL	6.1	6.6	18.0	18.0	0.9	0.3	0.3	1.9
15	libtiff-0860361d-1ba75257	PL	PL	PL	PL	24.4	20.0	15.6	14.8	1.2	1.6	1.6	2.0
16	libtiff-90d136e4-4c66680f	PL	PL	PL	PL	13.3	11.7	12.7	9.8	1.1	1.0	1.4	1.3
17	lighttpd-1948-1949	PL	PL	PL	PL	55.1	57.9	59.0	54.2	1.0	0.9	1.0	2.6
18	lighttpd-2330-2331	PL	PL	PL	PL	43.2	24.5	35.1	16.2	1.8	1.2	2.7	3.0
19	php-308525-308529	PL	PL	PL	PL	370.9	58.1	258.5	64.8	6.4	1.4	5.7	5.1
20	php-309688-309716	PL	PL	PL	PL	42.3	30.6	43.5	43.2	1.4	1.0	1.0	4.8
21	php-310011-310050	PL	PL	PL	PL	462.0	228.6	470.9	220.1	2.0	1.0	2.1	5.3
22	php-310370-310389	PL	PL	PL	PL	165.2	109.8	160.1	96.0	1.5	1.0	1.7	5.7
23	php-311323-311300	PL	PL	PL	PL	96.5	112.0	147.2	96.2	0.9	0.7	1.0	22.0
24	python-69368-69372	PL	PL	PL	PL	35.8	35.9	68.4	64.5	1.0	0.5	0.6	7.4
25	python-69709-69710	PL	PL	PL	PL	48.0	44.6	47.9	41.0	1.1	1.0	1.2	3.8
26	python-70019-70023	PL	PL	PL	PL	306.2	280.1	323.9	285.3	1.1	0.9	1.1	8.2
27	wireshark-37112-37111	PL	PL	PL	PL	42.0	30.0	32.6	25.4	1.4	1.3	1.7	12.3
Average										1.7	1.1	2.7	
Average (correct)										1.8	1.3	3.9	
28	libtiff-5b02179-3dfb33b	PL	PL	PL*	PL*	6.3	2.9	11.2	11.0	2.2	0.6	0.6	1.6
29	lighttpd-1913-1914	PL	PL	PL*	PL*	134.4	148.7	21.1	12.0	0.9	6.4	11.2	2.3
30	php-309111-309159	PL	PL	PL*	PL*	92.1	46.3	22.2	6.1	2.0	4.1	15.2	5.0
31	php-309986-310009	PL	PL	PL*	PL*	560.7	360.4	103.2	3.5	1.6	5.4	159.4	5.6
32	php-310673-310681	PL	PL	PL*	PL*	42.1	41.2	108.7	24.3	1.0	0.4	1.7	5.1
33	python-69223-69224	PL	PL	TO	TO	225.7	216.9	720.0	720.0	1.0	0.3	0.3	3.9
34	lighttpd-2661-2662	PL	PL*	PL*	PL*	162.0	165.2	12.4	7.7	1.0	13.1	21.0	1.5
35	python-70098-70101	PL	PL*	PL	PL*	63.8	34.4	60.6	35.3	1.9	1.1	1.8	2.7
36	wireshark-37172-37171	PL	PL*	PL*	PL*	39.4	40.0	24.2	17.1	1.0	1.6	2.3	13.4
37	wireshark-37172-37173	PL	PL*	PL*	PL*	36.0	34.6	19.6	29.8	1.0	1.8	1.2	13.2
38	wireshark-37284-37285	PL	PL*	PL*	PL*	35.9	40.2	21.9	29.7	0.9	1.6	1.2	13.2
39	libtiff-d13be72c-ccadf48a	TO	PL*	TO	PL*	720.0	19.4	720.0	4.1	37.1	1.0	173.5	2.4
40	php-308323-308327	TO	PL*	TO	PL*	720.0	121.7	720.0	12.7	5.9	1.0	56.8	5.0
41	lighttpd-1806-1807	TO	TO	TO	TO	720.0	720.0	720.0	720.0	1.0	1.0	1.0	2.2
42	php-307931-307934	TO	TO	TO	TO	720.0	720.0	720.0	720.0	1.0	1.0	1.0	5.0
43	php-308262-308315	TO	TO	PL*	PL*	720.0	720.0	79.0	19.5	1.0	9.1	36.9	12.4
Average										3.8	3.1	30.3	

SPR: original SPR, +TP: SPR with *Test-Case Pruning* technique, +LS: SPR with *Location Selection* technique, +TP+LS: SPR with both techniques
CR: a correct patch, PL: a plausible but incorrect patch, TO: no patch generated due to time-out, *: the patch is different from the SPR patch.

(bugs # $1-12$), faithfully re-generating each of the correct patches. For these instances, the average speed-up is $2.7X$, and 3.9X on only the correct instances. It can be as high as $11X$ on the correct or plausible instances (bug #11).

There are three instances, bug #3, 14, 24, where there is a modest slowdown. In these instances, *Location Selection* incorrectly de-prioritizes the correct location and the repair search takes significantly longer to re-discover the same patch, later. There are some remarkable speed-ups among the incomparable instances, specifically bugs #31, 40, and 41. These are discussed under RQ2.

The combined technique provides a 3.9X speed-up of repair, on average, without any degradation in correct patch quality.

C. RQ2: Speedup Contribution by Each Technique

The columns +TP and +LS under the speed-up section of Table I report the speed-ups provided (individually) by the *Test-Case Pruning* and *Location Selection* techniques respectively.

Discussion: *Test-Case Pruning* provides a speed-up of $1.7X$ on average ($1.8X$ on only the correct patches), producing modest slow-downs in some of the instances, *e.g.,* bug #10 where the speed-up is $0.9X$. In this case, and most others, the pruning of some (redundant) test-cases by *Test-Case Pruning* slightly perturbs the candidate batching order that was being produced by SPR's heuristics, actually increasing runtime by a modest amount. *Location Selection* provides a speed-up of $1.1X$ on average ($1.3X$ on only the correct patches). It also produces slow-down in several cases, sometimes as

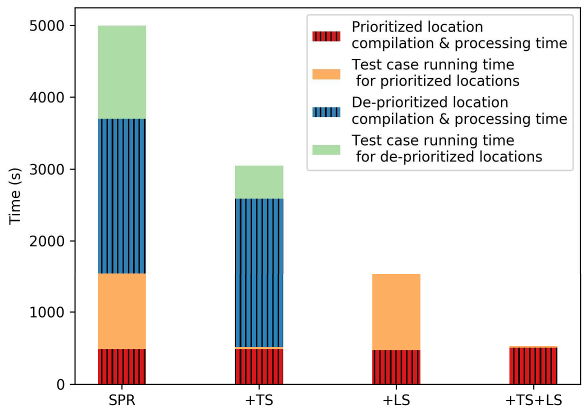

Fig. 2. The runtime of *php-307846-307853*, broken into compilation and processing time (darker colors, with vertical lines) and test case runtime sections (lighter colors). The test case runtime sections correspond to the compilation and runtime sections immediately below them on a bar.

much as $0.3X$ speedup (*i.e.,* $3.3X$ slow-down), *e.g.,* in bug #14. In these cases, one of two behaviors occur: the patch location is de-prioritized or the batching system is made less optimal.

The former case—the patch location is de-prioritized—occurs when either the positive test case set is unable to explore a good representation of the range of valid values for a variable or the failing value exists inside this range (*e.g.,* an integer value causes the test case to fail, but the values greater than or less than it do not cause failing behavior). This is the case for bug #14. The latter case—the batching system is made less optimal—occurs due to an interaction between the batching evaluation behavior. A batch attempts to compile many patches to nearby locations at once. If the locations are prioritized in a different order, then the size of the batch may decrease, causing more compilations.

Interestingly, the two techniques complement each other quite well, producing an overall speed-up that exceeds the product of the two individual speed-ups. An example of this behavior is in the motivating example *php-307846-307853*, which is bug #4. The runtime of this benchmark is given in Figure 2. In the figure, there are four bars corresponding to the four configurations tested: unmodified SPR, SPR with the *Test-Case Pruning* technique, SPR with the *Location Selection* technique, and SPR with both techniques enabled. Each bar is broken into two or more sections. We describe the sections by focusing on the first bar, which contains all sections. On this bar, the lowest section (dark red with vertical lines) is the compilation and processing runtime for patch locations that would have been prioritized; the second lowest section on the bar (light orange) is the test case execution time for those patch locations; the third section (dark blue with vertical lines) is the runtime for the compilation and processing time for patch locations that would be de-prioritized; and the fourth section (light green) is the runtime for the test case execution time for the de-prioritized patch locations. In the remaining bars, one or more of these sections have been eliminated or minimized by our techniques. In this figure, the speedup is clearly shown: the reason a better-than-linear speedup is possible is that the *Location Selection* removes both test cases and compilation runtime, and the *Test-Case Pruning* optimization does not affect both sets of patch locations (those prioritized or de-prioritized) equally.

Test-Case Pruning and *Location Selection* provide speed-ups of $1.8X$ and $1.3X$ respectively, on average. They work even better together, each compounding the other's gains.

D. RQ3: Reduction in Executed Test-cases & Repair Candidates

Table II shows data for the same 43 bugs as Table I, organized in the same two sets of bugs. Columns $3-5$, show statistics for the *Test-Case Pruning* technique, in particular, the number of test-cases executed per candidate patch evaluated, for the original SPR (column 3), SPR+TP (column 4) and the ratio of columns 3 and 4 as the reduction ratio (column 5). Columns $6-8$, show statistics for the *Location Selection* technique, in particular, the number of candidate patches evaluated till the generation of a successful patch (or timeout), for the original SPR (column 6), SPR+LS(column 7) and the ratio of columns 6 and 7 as the reduction ratio (column 8).

Discussion: Considering only the comparable instances (*i.e.,* the set of first 27 bugs) *Test-Case Pruning* produces, on average, an $9.6X$ reduction in the number of test-cases executed per candidate. For bug # 19 the reduction is quite remarkable ($126.5X$) but this is definitely an extreme case. For most instances TP produces a $1-2X$ reduction, and in some cases leaves it unchanged. Overall, this is still a substantial reduction, which ultimately contributes to the runtime speed-up (RQ1 and RQ2).

Overall, *Location Selection* produces an average $3.6X$ reduction in the number of candidates evaluated per repair run, compared to the original SPR. However, the gains are not uniform across all instances, ranging from a $7.7X$ reduction for bug #7 to a $10X$ increase for bug #24. This is because a single location can produce a different number of candidate patches depending on the statements at that location. For instance, SPR has different schemas for generating conditionals than for generating statements—should a location with many schemas available be de-prioritized, the number of candidate patches will drop more than if a location with few schemas is de-prioritized.

Test-Case Pruning produces an average $9.6X$ reduction in the number of test-cases executed per candidate. *Location Selection* produces an average $1.8X$ reduction in the number of candidates evaluated in a repair run.

E. RQ4: Overhead of the Techniques

As discussed earlier, *Test-Case Pruning* has no overhead, since the information needed to perform it is already provided by spectrum-based fault localization. The time for the pre-processing phase needed to implement *Location Selection*, is shown in the last column of Table I, for each bug. When taken as a fraction of the original SPR repair time for the corresponding bug, and averaged out, this comes to, an overhead of 5.6% for only correct patches, 9.0% when averaged on the first 27 bugs, *i.e.,* the comparable, instances, and 9.5% when considering all 43 bugs.

Discussion: Overall, the $10-20\%$ pre-processing overhead, while not negligible, is more than compensated by the speed-up in the repair time provided by the techniques (typically $2.7X$ speed-up on average for comparable cases), making the techniques a net-positive proposition. Further, with the exception of a single instance (where the pre-processing time is about 22 minutes) the pre-processing is of the order of $5-15$ mins., where the original SPR time was several tens of minutes.

Finally, we note that there is significant scope for optimizing the current, first-cut implementation of the pre-processing phase. One direction is to gather the necessary state information, not through a pre-processing phase (as is currently the case) but in a incremental,

TABLE II
QUANTITATIVE REDUCTIONS BY THE PROPOSED TECHNIQUES.

Bug		# of Test Cases Executed Per Repair Candidate			# of Repair Candidates Evaluated		
#	ID	SPR	+TP	Reduction Ratio [X]	SPR	+LS	Reduction Ratio [X]
1	gmp-13420-13421	1.6	1.5	1.1	6475.0	2976.0	2.2
2	libtiff-ee2ce5b7-b5691a5a	3.7	2.6	1.4	499.0	499.0	1.0
3	php-307562-307561	5.2	5.4	1.0	1675.0	5128.0	0.3
4	php-307846-307853	7.7	1.2	6.4	1999.0	125.0	16.0
5	php-307914-307915	21.8	22.1	1.0	387.0	377.0	1.0
6	php-308734-308761	45.3	2.1	21.6	2252.0	2252.0	1.0
7	php-309516-309535	11.0	1.5	7.3	1455.0	188.0	7.7
8	php-309579-309580	48.9	1.2	40.8	499.0	499.0	1.0
9	php-309892-309910	49.7	1.9	26.2	438.0	352.0	1.2
10	php-310991-310999	558.4	558.4	1.0	101.0	101.0	1.0
11	php-311346-311348	284.2	72.2	3.9	99.0	127.0	0.8
12	python-69783-69784	15.5	7.6	2.0	232.0	158.0	1.5
13	gmp-14166-14167	6.2	3.9	1.6	499.0	258.0	1.9
14	gzip-a1d3d4019d-f17cbd13a1	1.7	1.7	1.0	579.0	1050.0	0.6
15	libtiff-0860361d-1ba75257	7.7	8.0	1.0	397.0	296.0	1.3
16	libtiff-90d136e4-4c66680f	3.9	2.8	1.4	499.0	499.0	1.0
17	lighttpd-1948-1949	22.4	22.4	1.0	159.0	159.0	1.0
18	lighttpd-2330-2331	6.2	3.2	1.9	409.0	339.0	1.2
19	php-308525-308529	455.3	3.6	126.5	289.0	998.0	0.3
20	php-309688-309716	20.6	20.6	1.0	366.0	366.0	1.0
21	php-310011-310050	355.7	170.3	2.1	499.0	499.0	1.0
22	php-310370-310389	116.3	72.0	1.6	499.0	499.0	1.0
23	php-311323-311300	42.2	42.2	1.0	499.0	499.0	1.0
24	python-69368-69372	5.2	5.2	1.0	499.0	4999.0	0.1
25	python-69709-69710	9.0	9.0	1.0	320.0	320.0	1.0
26	python-70019-70023	3.2	3.2	1.0	2654.0	2654.0	1.0
27	wireshark-37112-37111	16.1	8.4	1.9	249.0	219.0	1.1
Average				9.6			1.8
28	libtiff-5b02179-3dfb33b	11.0	3.9	2.8	72.0	499.0	0.1
29	lighttpd-1913-1914	45.1	43.8	1.0	145.0	82.0	1.8
30	php-309111-309159	44.1	3.8	11.6	499.0	51.0	9.8
31	php-309986-310009	1991.8	1557.8	1.3	122.0	11.0	11.1
32	php-310673-310681	16.9	16.9	1.0	492.0	589.0	0.8
33	python-69223-69224	12.7	12.8	1.0	188.0	TO	-
34	lighttpd-2661-2662	3.2	2.6	1.2	999.0	155.0	6.4
35	python-70098-70101	2.2	1.4	1.6	1494.0	1494.0	1.0
36	wireshark-37172-37171	2.1	2.0	1.1	359.0	331.0	1.1
37	wireshark-37172-37173	2.1	1.8	1.2	369.0	266.0	1.4
38	wireshark-37284-37285	1.9	1.6	1.2	430.0	188.0	2.3
39	libtiff-d13be72c-ccadf48a	TO	6.6	-	TO	TO	-
40	php-308323-308327	TO	1.9	-	TO	TO	-
41	lighttpd-1806-1807	TO	TO	-	TO	TO	-
42	php-307931-307934	TO	TO	-	TO	TO	-
43	php-308262-308315	TO	TO	-	TO	24.0	-
Average				2.4			3.6

on-demand fashion during the repair run. We are currently experimenting with this idea among others.

Test-Case Pruning has no additional overhead. *Location Selection* has a $10 - 20\%$ pre-processing overhead, which is more than compensated by the $2 - 4X$ reduction in repair time provided by the two techniques.

V. THREATS TO VALIDITY

Internal validity. Our implementation of the two techniques in the context of SPR could have bugs, which could impact our internal validity. One technique that was used to validate the absence of bugs in the state recording mechanism was to use a slower implementation for verifying correctness and a faster mechanism for the actual evaluation runs, and verifying that they produced the same results.

External validity. Our techniques have currently only been implemented in the SPR tool [6] for C program repair and evaluated only on the GenProg benchmarks [2]. As such our conclusions may not apply to other G&V repair tools or other benchmarks or to repair for other languages, such as Java. Intuitively, our techniques do not directly exploit any special features of SPR, the GenProg benchmarks, or the C programming language, and should apply more generally. However, to rigorously mitigate this threat, replication studies on other benchmarks and G&V repair tools, including Java repair tools, should be conducted.

Construct validity. Our criterion for classifying patches as *correct* or *plausible* is based on manual analysis, which is not scientifically rigorous, even though it is accepted practice in previous work [4, 6, 7, 26]. For each correct patch that was produced, we verified that it had the same MD5 hash as the patch produced by the local installation of the unmodified SPR. Any patches that produced different MD5 hashes were manually compared to see if the patches were otherwise semantically identical and none of the patches with different hashes were considered semantically equivalent.

Other metrics, such as number of repair candidates evaluated and the number of test-cases evaluated per candidate, while not common, are simple modifications of metrics used by previous work on program repair [8, 27].

VI. Related Work

Search-based repair. The objective of research in this area, at least initially, was to maximize the fix-rate of the repair technique, *i.e.,* the fraction of instances for which a patch (passing all tests) is produced. The proposed techniques use innovations in the underlying search technique, the repair search space defined by the repair templates, or the search order to achieve this. GenProg [2] uses a genetic programming based search and produced patches for 55 out of 110 bugs evaluated. RSRepair [10] uses a random search instead, with similar success. PAR [11] proposes a set of repair schemas manually derived from human-written patches, Relifix [12] proposes a set of repair schemas specialized for regression errors, while SPR [6] prioritizes repairs related to conditional statements. Prophet [3] and history-driven repair [5] both use knowledge from a large corpus of previous successful patches to order the search space of potential repairs; Prophet relies on a machine-learned model while [5] abstracts patches into canonical graph representations. Machine learning has also been applied to repair selection conditions in database statements in ABAP programs [28] and more recently conditional statements in Java programs [29]. SketchRep [30] reduces the problem of program repair to program sketching [31] and uses the SAT-based Sketch system as an off-the-shelf synthesis backend. More recent work [32] performs a similar reduction but employs execution-driven sketching [33] to solve the ensuing search problem and perform program repair.

Recent studies have highlighted the concern around patch quality [26, 34]. The Kali tool [26] demonstrated that many of the earlier G&V repair tools tend to produce patches that effectively deleted (valid) functionality. Smith et al. [34] further showed that G&V repair tools tend to overfit repairs to the weak specifications (*i.e.,* test suites) they work with, leading to poor quality repairs. Tan et al. [13] use this understanding to propose a set of generic, forbidden repair transformations, which they refer to as anti-patterns, to block nonsensical repairs that might otherwise be produced. In very recent work, ACS [14] proposes a method for precise condition synthesis by instantiating heuristically ranked variables in frequently occurring predicates, mined from a given corpus of code.

While most of the above innovations have indirectly also contributed to the efficiency of G&V repair, only a handful of techniques have directly targeted or even evaluated the efficiency of repair [6, 8]. AE [8] uses deterministic search coupled with light-weight program analysis to prune equivalent patches. SPR [6] uses abstract conditions to evaluate and prune candidates for condition-related repairs, before concretizing the repairs, typically providing an order of magnitude speed-up over vanilla G&V repair. In any case, our proposed techniques are orthogonal to the above techniques, since they are independent of the search algorithm or the repair schemas used (*i.e.,* the repair space). Thus, arguably they can be integrated into any G&V repair technique.

Oracle-based repair. These techniques use some analysis, typically symbolic execution, to generate an oraclular representation of the repair. They differ in how they create a concrete repair from the repair oracle. SemFix [35] uses program synthesis for this purpose, while MintHint [36] uses statistical analysis to search for a concrete repair. DirectFix [37] attempts to generate comprehensible patches by generating minimal repairs; it casts repair concretization as a partial maximum satisfiability problem over satisfiability modulo theories (SMT) formulas. Angelix [4] retains this notion of minimality but enhances scalability by using a light-weight repair constraint. SearchRepair [38] searches for repairs in a corpus of human-written correct patches, indexed on the basis of their input-output behavior, encoded as SMT constraints. NOPOL [7], a predecessor of SPR, also uses abstract conditions as repair oracles of condition-related repairs, but concretizes the repair by encoding it as an SMT formula. In principle, our proposed techniques, especially *Location Selection* should also be applicable to oracle-based repair techniques, but would require further investigation to establish feasibility and impact.

Regression test selection. Regression test selection (RTS) is a well-research body of work, spanning over three decades, summarized in several excellent surveys [18–22]. The main distinction between different RTS techniques is the granularity at which they collect dependency information. While it seems natural, even obvious, to apply RTS techniques to optimize G&V repair, the challenge is that classical RTS techniques would be either too-grained and/or too expensive to be useful in the context of repair. Our contribution is to design a precise, but relatively cheap RTS technique that effectively exploits the mechanics of G&V repair.

Delta Debugging. The work on Delta Debugging and its extensions [15–17, 39, 40] attempts to precisely localize the location of a defect, as well as a minimal subset of input or state variables responsible for it. This is done by iteratively and systematically mutating the program state under a single failing test case and comparing it with that of a single passing test, at the same location, under the applied mutations. However, while this work provided the initial motivation for our *Location Selection* technique our approach is fairly different in both its objective and its mechanics. Our aim is to identify program locations where it would be unproductive to attempt repairs and de-prioritize such locations during a G&V repair search. This is done through a heuristic function which compares, *in aggregate*, the program states produced by *all* failing and passing tests (in the complete test suite), to determine if failing and passing tests exhibit substantially similar behavior at that location. This involves no mutations or repeated execution of tests.

VII. Conclusions & Future Work

Repair efficiency of program repair, *i.e.,* the time it takes for a repair tool to generate a successful patch, is currently one of the key impediments to the practical adoption of such tools. In this work we proposed two complementary optimization techniques, namely *Location Selection* and *Test-Case Pruning*, to substantially improve the efficiency of search-based repair techniques. We implemented them in the context of the SPR search-based repair tool, and evaluated them on the GenProg benchmarks. As our experiments demonstrate, *Location Selection* successfully reduces the number of candidates that need to be examined before generating a repair, by $1.8X$ on average, while *Test-Case Pruning* reduces the number of test cases executed per examined candidate, by $9.6X$ on average. Together they accelerate the repair runs, by a factor of $3.9X$, on average. Interestingly, the techniques perfectly complement each other, with the speed-up of the combined technique surpassing even the product of the individual speed-ups. Further, the techniques can be integrated into current G&V repair flows fairly easily, through an additional pre-processing step, which consumes $10 - 20\%$ of the repair time on average. There is significant scope for further reducing this overhead. We are actively exploring this, as well as other research ideas to enhance repair efficiency.

References

[1] Cambridge University, "Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year," http://www.prweb.com/releases/2013/1/prweb10298185.htm, 2013.

[2] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13.

[3] F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 298–312.

[4] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 691–701.

[5] X. B. D. Le, D. Lo, and C. L. Goues, "History Driven Program Repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. Piscataway, NJ, USA: IEEE Press, March 2016, pp. 213–224.

[6] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 166–178.

[7] J. Xuan, M. Martinez, F. DeMarco, M. Clment, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, Jan 2017.

[8] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Piscataway, NJ, USA: IEEE Press, Nov 2013, pp. 356–366.

[9] C. Goues, S. Forrest, and W. Weimer, "Current Challenges in Automatic Software Repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, Sep. 2013.

[10] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The Strength of Random Search on Automated Program Repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265.

[11] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[12] S. H. Tan and A. Roychoudhury, "Relifix: Automated Repair of Software Regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 471–482.

[13] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in Search-based Program Repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 727–738.

[14] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 416–426.

[15] A. Zeller, "Isolating Cause-effect Chains from Computer Programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10. New York, NY, USA: ACM, 2002, pp. 1–10.

[16] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[17] H. Cleve and A. Zeller, "Locating Causes of Program Failures," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 342–351.

[18] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

[19] E. Engström and P. Runeson, "A Qualitative Survey of Regression Testing Practices," in *Proceedings of the 11th International Conference on Product-Focused Software Process Improvement*, ser. PROFES'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 3–16.

[20] E. Engström, P. Runeson, and M. Skoglund, "A Systematic Review on Regression Test Selection Techniques," *Inf. Softw. Technol.*, vol. 52, no. 1, pp. 14–30, Jan. 2010.

[21] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression Test Selection Techniques: A Survey." *Informatica (Slovenia)*, vol. 35, no. 3, pp. 289–321, 2011.

[22] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[23] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[24] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: A Spectrum-based Fault Localization Tool," in *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime*, ser. SINTER '09. New York, NY, USA: ACM, 2009, pp. 23–30.

[25] "Index of /spr-rep," http://rhino.csail.mit.edu/spr-rep/, 2016, accessed:2016-06-11.

[26] Z. Qi, F. Long, S. Achour, and M. Rinard, "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 24–36.

[27] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 191–201.

[28] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra, "Data-Guided Repair of Selection Statements," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 243–253.

[29] D. Gopinath, K. Wang, J. Hua, and S. Khurshid, "Repairing Intricate Faults in Code Using Machine Learning and Path Exploration," in *IEEE International Conference on Software*

Maintenance and Evolution (ICSME), 2016, pp. 453–457.

[30] J. Hua and S. Khurshid, “A Sketching-Based Approach for Debugging Using Test Cases,” in *14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2016, pp. 463–478.

[31] A. Solar-Lezama, “Program sketching,” *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013.

[32] J. Hua, M. Zhang, K. Wang, and S. Khurshid, “Towards practical program repair with on-demand candidate generation,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18, 2018, to appear.

[33] J. Hua and S. Khurshid, “EdSketch: Execution-driven sketching for Java,” in *24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, 2017, pp. 162–171.

[34] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 532–543.

[35] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “SemFix: Program Repair via Semantic Analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781.

[36] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, “MintHint: Automated Synthesis of Repair Hints,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 266–276.

[37] S. Mechtaev, J. Yi, and A. Roychoudhury, “DirectFix: Looking for Simple Program Repairs,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 448–458.

[38] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, “Repairing Programs with Semantic Code Search (T),” in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 295–306.

[39] D. Jeffrey, N. Gupta, and R. Gupta, “Fault localization using value replacement,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA ’08. New York, NY, USA: ACM, 2008, pp. 167–178.

[40] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, “Bugfix: A learning-based tool to assist developers in fixing bugs,” in *2009 IEEE 17th International Conference on Program Comprehension*, May 2009, pp. 70–79.