2017 IEEE/ACM 39th International Conference on Software Engineering

Optimizing Test Placement for Module-Level Regression Testing

August Shi*, Suresh Thummalapenta†, Shuvendu K. Lahiri‡, Nikolaj Bjorner‡, Jacek Czerwonka†
*University of Illinois at Urbana-Champaign, USA
†Microsoft Corporation, USA
‡Microsoft Research, USA

***Abstract*—Modern build systems help increase developer productivity by performing incremental building and testing. These build systems view a software project as a group of inter-dependent modules and perform regression test selection at the module level. However, many large software projects have imprecise dependency graphs that lead to wasteful test executions. If a test belongs to a module that has more dependencies than the actual dependencies of the test, then it is executed unnecessarily whenever a code change impacts those additional dependencies.**

In this paper, we formulate the problem of wasteful test executions due to suboptimal placement of tests in modules. We propose a greedy algorithm to reduce the number of test executions by suggesting test movements while considering historical build information and actual dependencies of tests. We have implemented our technique, called TestOptimizer, on top of CloudBuild, the build system developed within Microsoft over the last few years. We have evaluated the technique on five large proprietary projects. Our results show that the suggested test movements can lead to a reduction of 21.66 million test executions (17.09%) across all our subject projects. We received encouraging feedback from the developers of these projects; they accepted and intend to implement ≈80% of our reported suggestions.

I. Introduction

Large-scale software development projects use *build systems* to manage the process of building source code, applying static analyzers, and executing tests. Any inefficiencies in the underlying build system directly impact developer productivity [5]. Given the significance of build systems, major companies, such as Microsoft and Google, have made huge investments in developing efficient, incremental, parallel, and distributed build systems. Example build systems include CloudBuild [8], [18], Bazel [1], and FASTBuild [2].

These build systems view a software project as a group of inter-dependent modules and inherently perform (safe) regression test selection [9], [19], [24] at the module level. Given a change, they leverage the build dependency graph to identify modules that are impacted by the change and perform activities (such as building or applying static analyzers) only on those modules. More specifically, whenever any dependency of a test module is impacted by the given change, all tests in that module are executed; otherwise, the module is skipped and no tests in the module are executed. The major advantage of module-level test selection is that it does not require any additional metadata (such as fine-grained dependencies like exercised statements for each test) beyond what is available in the build specification. Especially in the case of large software projects that execute millions of tests each day, storage and maintenance of this additional metadata adds non-trivial overhead. Therefore, module-level test selection is the most practical option for performing test selection.

Despite the increasing sophistication of build systems, large software projects with thousands of modules often have dependency graphs that make inefficient use of these build systems. Such dependency graphs not only increase the build-activity time, but also make module-level test selection less efficient by executing tests that are not affected by the given change. Vakilian et al. [22] studied the impact of monolithic build modules on the performance of distributed builds. Their work focuses on underutilized modules that include files not needed by some of its dependents and attempts to split them into smaller modules. Our work highlights another source of inefficiency in the form of wasteful test executions due to test placement in test modules that have more dependencies than the tests actually need. Therefore, many irrelevant tests that are not affected by a change often get executed due to changes in developer-specified dependencies of the module, and such changes cannot alter the behavior of these tests. Execution of irrelevant tests can not only waste machine resources but can also severely affect developer productivity if those irrelevant tests are flaky [13]. Flaky tests are tests that can pass or fail even without any changes to code. In practice, each test failure requires developers to manually triage the failures and identify the root cause. Therefore, when an irrelevant flaky test fails, developers end up spending unnecessary effort debugging the failure (which has nothing to do with their change) only to identify that the failure is due to a flaky test.

From our experience with CloudBuild, the reasons for such incorrect placement of tests include a lack of comprehensive knowledge of all test modules in the project and developers making large refactorings to the code base (more details in Section V-D). The goal of our work is to provide a practical solution to reduce the number of wasteful test executions under the constraints imposed by the underlying build systems. Our solution for finding better placement for tests is applicable beyond just the CloudBuild build system, as it can be applied to any build system that builds at the level of modules, such as the build system that is used at Google (Bazel) [1].

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.69

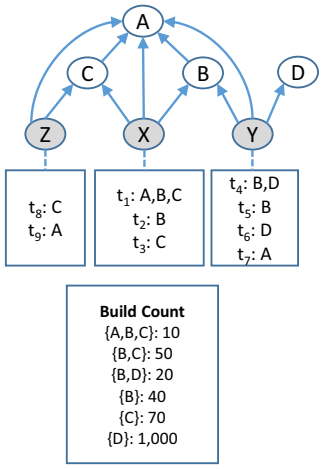

Fig. 1. An example dependency graph with build nodes `A`, `B`, `C`, and `D`; test nodes `X`, `Y`, and `Z`, actual dependencies for tests t_1 through t_9, and the build count (number of times built) for sets of build nodes

A. Running Example

To illustrate how a build system performs regression test selection at the module level and the problem we address in this paper, consider an example dependency graph shown in Figure 1 for a hypothetical project. The example project includes four application modules (`A`, `B`, `C`, and `D`), and three test modules (`X`, `Y`, and `Z`). In the rest of this paper, we use the notation *build node* to represent an application module and *test node* to represent a test module. When a change happens to the code in a build node, the build node is built, and all build nodes and test nodes transitively dependent on that build node are built as well. Whenever a test node is built, all the tests inside it are also executed. For example, when test node `Z` is built, tests t_8 and t_9 are executed. The directed edge from `C` to `A` indicates that `C` is dependent on `A`.

Given this background, we next explain how this dependency graph can lead to wasteful test executions. Suppose that a developer makes a change to code in build node `D`. As such, `D` is built, and test node `Y`, which depends on `D`, is built as well, and all the tests in `Y` are executed. However, as shown in Figure 1, tests t_5 and t_7 do not actually depend on `D`, and they need not be executed when `D` changes. These two tests are executed because they happen to be placed in `Y`. This inefficiency is even worse if the dependencies change more frequently. The table "Build Count" in Figure 1 shows the number of times a set of build nodes are built together, e.g., the set of build nodes {`A`,`B`,`C`} are built 10 times together. We see that, in this example, build node `D` by itself is built 1,000 times. As `Y` depends on `D`, it is built at least 1,000 times, hence all tests in test node `Y` are executed at least 1,000 times. To be more precise, the tests are executed 1,120 times, because these tests also need to be executed when `D` is built in combination with other build nodes and when `B` is built as well, resulting in an additional 120 times (when sets {`A`,`B`,`C`}, {`B`,`C`}, {`B`,`D`}, and {`B`} are built).

B. Overview of Our Technique

In this paper, we focus on reducing wasteful test executions due to incorrect placement of tests within test nodes. An ideal placement of tests to avoid wasteful test executions is to place each test in a test node that shares exactly the same set of dependencies that are exercised by the test during its execution. This placement ensures that only relevant tests get executed for a given change. However, in large projects, moving thousands of tests to achieve the ideal placement requires huge effort. Given that developers are always under pressure to meet customer requirements, they often have limited time to perform these activities. Therefore, it is not practical to invest huge effort (although one-time) in achieving the ideal placement. Furthermore, introducing a large number of modules can significantly increase the time to build all modules in the project. Our technique addresses these issues by suggesting movements that reduce the number of test executions while *minimizing the number of suggestions* to give to developers.

Our technique, called *TestOptimizer*, accepts the following inputs: dependency graph, actual dependencies for tests, and number of times build nodes have been built over a given range of time. TestOptimizer formulates the problem as a decision problem, asking if there is a way to split test nodes to reduce the overall number of wasteful test executions. TestOptimizer uses a heuristic, called *affinity*, which is the set of dependencies the tests in a test node should be testing (Section IV-B). Affinity helps determine the tests that need not be moved, i.e., tests that execute the affinity of its test node are not amenable to movement. TestOptimizer uses a greedy algorithm to iteratively suggest test movements that lead to reductions in number of test executions. These suggestions involve moving tests into a newly created test node or to an existing test node that shares the exact same dependencies as the tests. Furthermore, TestOptimizer suggests the test movements ranked in the order of highest reduction in the number of test executions first. We envision that developers can use TestOptimizer once to get the correct placement of all tests. Once the recommendations are incorporated, TestOptimizer need to be executed only on added or modified tests.

For the example shown in Figure 1, TestOptimizer produces two suggestions (moving tests t_5 into a new test node and moving tests t_7 and t_9 into a new test node). These suggestions reduce the number of test executions by 2,230 test executions, a 42% reduction (more details in Section III) for this example.

C. Contributions

This paper makes the following contributions:

- We formalize the problem of wasteful test executions due to incorrect placement of tests in test nodes.
- We define a cost metric that models the expected number of test executions in a given range of time based on historical build count.
- We propose a technique, called TestOptimizer, that uses a greedy algorithm to suggest test movements between test nodes. Our technique produces a ranked list of

suggestions, prioritizing the ones that give the highest reductions in the expected number of test executions.
- We implement TestOptimizer in a tool on top of CloudBuild, a widely used build system at Microsoft.
- To evaluate our tool, we apply it on five large proprietary projects. Our results show that the suggested movements can result in a reduction of 21.66 million test executions (17.09%) across all our subjects. We received positive feedback from the developers, who also accepted and intend to implement ≈80% of our suggestions.

II. Problem Statement

Let $\mathcal{B}$ be the set of all *build nodes* for the project. Let $\mathcal{N}$ be the set of all *test nodes* for the project. Let $\mathcal{T}$ be the set of all the *tests* for the project, contained in the set of test nodes. Let Π be a partitioning of $\mathcal{T}$, and for a test node n, let $\Pi(n) \subseteq \mathcal{T}$ be the set of tests contained in n. Given that Π is a partitioning of $\mathcal{T}$, $\Pi(n_1) \cap \Pi(n_2) = \emptyset$ for distinct test nodes n_1 and n_2. For each test t, let $TDeps(t) \subseteq \mathcal{B}$ be all the build nodes that t depends on – these are the build nodes t needs to both compile and to execute. For the purpose of this paper, we require that build nodes do not depend on test nodes. Further, whenever a test node n is built, all the tests in $\Pi(n)$ are executed.

Consider a sequence of builds during a given range of time R (say, over a six-month period). There are two factors that influence the total number of test executions for a project during R: (i) the count of the number of times test nodes are built over R (given the incremental nature of a build, not every test node in a project is built in a given build), and (ii) the number of tests executed when building a test node.

Our solution changes the partitioning Π of tests by moving tests from existing test nodes to other (possibly new) test nodes. To compute the reduction in the number of test executions, we first need to compute the number of times test nodes are built (the test nodes' build counts) after a change in partitioning Π without actually replaying the builds during R.

A. Computing Build Count for a Test Node

Let us assume we have collected the build count for all the build nodes and test nodes during a past range of time R. To determine the build count of a test node n after moving tests across test nodes, one possibility is to simply reuse the build count of n collected in that range of time R. However, simply reusing the build count is inaccurate for two main reasons. First, after moving tests between test nodes, there can be changes to the dependencies of each existing test node. Second, new test nodes can be added, which do not have any existing build count information. Recall the example in Figure 1. If test t_8 is moved out of `Z`, that movement removes the dependency that test node `Z` has on `C`. This modified test node cannot be expected to have the same build count as before. Similarly, if t_7 is moved out of `Y` and into a brand new test node that depends only on `A`, we would not have any collected build count for this new test node.

However, in most cases we can accurately compute the build count of a given test node entirely in terms of the build count of the build nodes in $\mathcal{B}$ and the dependencies of the tests within the given test node. We make the following assumptions during the range of time R:

1) The dependencies of individual tests do not change.
2) Tests are changed, added or removed to the test nodes only in conjunction with another change to a dependent build node, i.e., a change does not affect test nodes only.

From our experience with CloudBuild, we believe these assumptions hold for most test nodes. For the first assumption, given that the dependencies in question are modules in the project as opposed to finer-grained dependencies such as lines or files, it is unlikely that there are drastic changes that lead to changes in dependencies at the level of modules. For the second assumption, we find that it is very rare for developers to be only changing test code; they change code in build nodes much more frequently and changes made to test nodes are in response to those build node changes. As we explain below, given these assumptions, we can compute precisely when a test node n will be built, namely whenever at least one dependency in $TDeps(n)$ is built. Although these assumptions may not hold for a very small fraction of test nodes during the time frame R, their effect is negligible when R is sufficiently large (e.g., a few months).

We define the dependencies of a test node n as the union of the dependencies of the tests contained in n given Π.

Definition 1 ($NDeps(n, \Pi)$): For a given test node n, the set of build nodes that n depends on under Π is defined as:

$$NDeps(n, \Pi) \doteq \{b \in \mathcal{B} \mid \exists t \in \Pi(n) \text{ such that } b \in TDeps(t)\}$$

Given our assumptions about test nodes, a test node n is built *if and only if* a build node in $NDeps(n, \Pi)$ is built.

For a subset $\mathcal{B}' \subseteq \mathcal{B}$, let $BC_R(\mathcal{B}')$ denote the number of builds where only the build nodes in $\mathcal{B}'$ are built together. The box in Figure 1 titled "Build Count" shows the number of times each subset of build nodes is built, e.g., the exact subset {`A`, `B`, `C`} was built 10 times in the range of time. Given our assumptions of test nodes not changing, a test node n is built in a build iff any of its dependencies is built. We can thus compute the number of times a test node n is built by summing up the build counts of $BC_R(\mathcal{B}')$ where $\mathcal{B}'$ intersects with $NDeps(n, \Pi)$.

Definition 2 ($NodeCount_R(n, \Pi)$): For a given test node n, the computed number of times it is built during a range of time R is defined as:

$$NodeCount_R(n, \Pi) \doteq \sum_{\{\mathcal{B}' \subseteq \mathcal{B} \mid \mathcal{B}' \cap NDeps(n,\Pi) \neq \emptyset\}} BC_R(\mathcal{B}')$$

Since the set of all builds in R can be partitioned by using the distinct subsets $\mathcal{B}'$ as identifiers, we count each build exactly once in the above equation.

B. Number of Test Executions in a Project

Our metric for the cost of testing is the number of tests that are executed over R. As such, our definition of the testing cost for a test node is related to the number of tests in the test node and the number of times the test node would be built:

Definition 3 ($NodeCost_R(n, \Pi)$): The number of test executions for a test node n within a range of time R is the product of the number of tests in n and the build count of n within R:

$$NodeCost_R(n, \Pi) \doteq |\Pi(n)| \times NodeCount_R(n, \Pi)$$

The total number of test executions for testing the entire project would then be the sum of the number of test executions for each test node.

Definition 4 ($Cost_R(\mathcal{N}, \Pi)$): The number of test executions $Cost_R(\mathcal{N}, \Pi)$ needed for building all test nodes $\mathcal{N}$ in a project within a range of time R is defined as:

$$Cost_R(\mathcal{N}, \Pi) \doteq \sum_{n \in \mathcal{N}} NodeCost_R(n, \Pi)$$

C. Reducing Test Executions

Our goal is to reduce the number of test executions in a project. While we could formulate the problem to allow placement of tests to any test node irrespective of the initial placement Π, the resulting ideal placement of tests could be very different from the original placement of tests. Such a placement would then result in too many suggested test movements for the developers to implement, and it is not practical to invest huge effort (although one-time) in achieving this ideal placement of tests.

In this paper, we consider the option of *splitting* a test node n into two test nodes $\{n, n'\}$ ($n' \notin \mathcal{N}$) where a subset of tests from n are moved to n'. As such, we can constrain the number of suggested test movements to report to developers. We define Π^n (over $\mathcal{N} \cup \{n'\}$) to be identical to Π at $\mathcal{N} \setminus \{n, n'\}$, and the disjoint union $\Pi^n(n) \uplus \Pi^n(n') = \Pi(n)$, meaning that Π^n only moves a subset of tests (possibly empty) from n to n'.

Our test placement problem can now be restated as the following decision problem:

> Does there exist a $n \in \mathcal{N}$ such that $Cost_R(\mathcal{N}, \Pi) > Cost_R(\mathcal{N} \cup \{n'\}, \Pi^n) + c$?

where c is a constant value representing a threshold for reducing at least a certain number of test executions.

The threshold c acts as a knob for controlling suggestions, where a split is suggested only when the reduction in number of test executions is worth the overhead of the developer implementing the suggestion. Essentially, c represents the minimal return-on-investment that can be expected by a developer to implement the suggested split. With multiple such n that reduce the number of test executions by at least c, the one that provides the highest reduction can be chosen first.

III. TestOptimizer

Our technique, called TestOptimizer, provides a practical solution to reduce the number of wasteful test executions.

Algorithm 1 Splitting a test node

```
1:  procedure SplitTestNode(n, Π, R)
2:      groups ← NewMap()
3:      for each t ∈ Π(n) do
4:          deps ← TDeps(t)
5:          if ¬groups.ContainsKey(deps) then
6:              groups[deps] ← ∅
7:          end if
8:          groups[deps] ← groups[deps] ∪ {t}
9:      end for
10:     n' ← NewNode()
11:     Π^n ← NewPartition(Π, n')
12:     repeat
13:         tests ← ∅
14:         maxCost ← Cost_R({n, n'}, Π^n)
15:         for each deps ∈ groups.Keys do
16:             Π' ← Π^n
17:             RemoveTests(Π', n, groups[deps])
18:             AddTests(Π', n', groups[deps])
19:             newCost ← Cost_R({n, n'}, Π')
20:             if newCost < maxCost then
21:                 tests ← groups[deps]
22:                 maxCost ← newCost
23:             end if
24:         end for
25:         if tests ≠ ∅ then
26:             RemoveTests(Π^n, n, tests)
27:             AddTests(Π^n, n', tests)
28:         end if
29:     until tests = {}
30:     return Π^n, n'
31: end procedure
```

More specifically, TestOptimizer produces a ranked list of suggestions that help reduce a large number of wasteful test executions with minimal test movements. Our technique also allows developers to specify threshold c in terms of the number of test executions that should be reduced for a suggestion. The suggestions also include additional recommendations where tests can be moved into existing test nodes, in case such test nodes already exist in the dependency graph. We first present how TestOptimizer splits a single test node into two test nodes and then explain how to use the single-split algorithm to deal with all test nodes in a given project.

A. Splitting a Test Node

Given a test node, we seek to find a subset of tests that can be moved to a new test node, resulting in the highest reduction in number of test executions. We use an iterative greedy algorithm to find such subset of tests that can be moved from the given test node.

Algorithm 1 shows the individual steps. Given a test node n, the loop in lines 3 to 9 iterates over the individual tests in that test node. For each test t, our algorithm gets the build nodes the test depends on using $TDeps(t)$. If an existing

Iteration	Node Y	Node Y_1	# Test Execs
1	$g_1 : \{t_4\}$ $g_2 : \{t_5\}$ $g_3 : \{t_6\}$ $g_4 : \{t_7\}$		4,480
2	$g_1 : \{t_4\}$ $g_2 : \{t_5\}$ $g_3 : \{t_6\}$	$g_4 : \{t_7\}$	3,370
3	$g_1 : \{t_4\}$ $g_3 : \{t_6\}$	$g_4 : \{t_7\}$ $g_2 : \{t_5\}$	2,480

TABLE I
STEPS IN SPLITTING TEST NODE Y

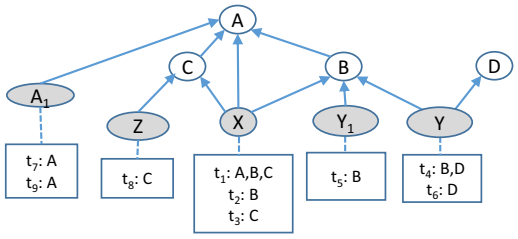

Fig. 2. Final dependency graph after applying our suggestions

group of tests already shares the same set of build nodes as dependencies, the test is added to that group; otherwise, a new group is created.

Next, the algorithm simulates moving groups of tests into another (initially empty) test node so as to identify the group that results in the highest reduction. At line 10, the algorithm makes a new test node n', and at line 11, the algorithm makes a new partitioning Π^n that includes n'. The loop 15 to 24 iterates through the groups and simulates moving each group of tests from n to n'. The simulation is done by using a temporary partitioning Π' that starts as Π^n but is modified by using $RemoveTests$ to remove a group of tests from n and then using $AddTests$ to add the same group of tests to n'. The algorithm chooses the group of tests whose movement results in the highest reduction in number of test executions. The outer loop (lines 12 to 29) greedily chooses to move that group of tests from n into n' (lines 25 to 28) by modifying that new partitioning Π^n. The outer loop iterates until there are no groups that help reduce the number of test executions. When the loop terminates, the algorithm returns the new partitioning Π^n and the new test node n'. In case n cannot be split, Π^n will map n' to an empty set. Our algorithm moves groups of tests instead of individual tests in each iteration due to two reasons. First, moving groups of tests instead of individual tests can make the search faster. Second, since the group of tests share the same dependencies, the tests are likely related to each other and should be placed in the same test node.

In the example dependency graph shown in Figure 1, consider the given test node n as Y. Table I shows the result after each iteration of the outer loop (lines 12 to 29). Initially, Loop 3 to 9 identifies four groups of tests, shown as g_1 to g_4. The initial number of test executions computed is $4,480$. At the end of the first iteration of the outer loop, group g_4 is selected as the group that results in the highest reduction in the number of test executions, resulting in $3,370$ test executions. Note that g_4 includes the test t_7 that depends only on A, which is the root node of the dependency graph. Therefore, it is clearly evident that a large number of wasteful test executions is due to the test t_7. After one more iteration, the algorithm returns the partitioning that maps test node n to tests in groups g_1 and g_3 and maps new test node n' to tests in groups g_2 and g_4; this results in a final $2,480$ test executions.

B. Handling all Test Nodes

We next explain how we use the previous algorithm to deal with all test nodes in the given project. Given a set of test nodes, TestOptimizer applies Algorithm 1 to each test node to find a split that results in the highest reduction. In case the algorithm returns a partitioning where the new test node is mapped to no tests, TestOptimizer considers that the test node under analysis cannot be split further. In each iteration, TestOptimizer finds the test node that has the best split, i.e., producing the highest reduction in the number of test executions. TestOptimizer then updates the overall partitioning with the returned partitioning for that best split and adds the new test node into the set of all test nodes. TestOptimizer repeats these steps until there exist no test node that can be split any further. At the end, TestOptimizer returns all suggestions ranked in descending order based on their reductions in number of test executions. If there are any new test nodes that share the exact same dependencies as another test node (new or existing), TestOptimizer also makes the suggestion to combine the two test nodes into one.

TestOptimizer also allows developers to specify thresholds for a split. From Section II-C, this is the threshold value c, representing the minimal number of test executions a split must reduce by. The threshold is implemented by adding an additional condition to line 20 in Algorithm 1. These criteria can help eliminate trivial suggestions that may not be worthwhile of the effort required to implement the suggestion.

Returning to our running example, consider the threshold c as 100 test executions. Our technique first splits Y, creating new test node Y_1, as shown in Table I. It next splits Z, creating new test node Z_1, where Z now only includes t_8 and Z_1 includes t_9. Our technique finally splits Y_1, creating new test node Y_2; Y_1 now includes t_5 and Y_2 includes t_7. TestOptimizer terminates since no further split can achieve the given criterion. TestOptimizer also suggests the two test nodes Y_2 and Z_1 including tests t_7 and t_9, respectively, can be combined into one test node A_1, since both the tests share exactly the same dependency. Figure 2 shows the final dependency graph after applying our suggestions. Overall, for this example, our technique reduces the number of test executions by $2,230$.

C. Properties of TestOptimizer

Using our technique, we can guarantee that every suggestion does indeed ensure a reduction in number of test executions. However, our technique may fail to find a split that reduces the cost, even if there exists one. This is due to the *greedy* nature of the algorithm that only moves *one* group of tests

(that share the same dependency) from n to the split node n' (one that provides the maximum reduction in number of test executions). For lack of space, we omit such an example (there exists one) that shows the limitation of our algorithm.

IV. Implementation

We implemented TestOptimizer as a prototype on top of CloudBuild [8], a widely used build system at Microsoft. Currently, CloudBuild handles thousands of builds and executes millions of tests each day.

A. Code Coverage

For our implementation, we targeted the tests written using the Visual Studio Team Test (VsTest) framework [4], as the majority of tests in CloudBuild are executed using VsTest. Since TestOptimizer requires actual dependencies of each individual test, we use the Magellan code coverage tool [3] to collect those dependencies. In particular, we first instrument the binaries in a project, where a binary corresponds to a build node, using Magellan. We then execute the tests on the instrumented binaries and save a coverage trace for each test. The coverage trace includes all blocks that are executed by the test. We map these blocks to the build nodes to construct the set of actual dependencies for each test. Our implementation also handles the special constructs such as *AssemblyInitialize* and *ClassInitialize* that have specific semantics in VsTest. For example, *AssemblyInitialize* is executed only once when running all the tests in a test node, but the binaries exercised should be dependencies to all the tests in that test node.

B. Test Node Affinity

We use *affinity* to refer to the set of build nodes that are intended to be tested by a test node. Affinity helps avoid suggesting moving tests that are already in the right test node and also helps improve the scalability of our technique. Ideally, one would ask the developers to provide the set of build nodes each test node is intended to test. However, as it is infeasible to ask the developers to spend time labelling every single test node, we instead develop a heuristic to automatically compute the affinity for any given test node n.

First, we compute the dependencies $NDeps(n, \Pi)$ of the test node. Next, we compute $\Gamma(b, n)$ of each $b \in NDeps(n, \Pi)$ as the number of tests in n that covers b during their execution.

$$\Gamma(b, n) = |\{t \in n | b \in TDeps(t)\}|$$

Finally, let $\Gamma^{Max}(n) = \max_{\{b' \in NDeps(n,\Pi)\}} \Gamma(b', n)$, the maximum Γ value among all build nodes in n. We compute the affinity as:

$$\textit{Affinity}(n) = \{b \in NDeps(n, \Pi) | \Gamma(b, n) = \Gamma^{Max}(n)\}$$

Once the affinity is computed, any test that does not exercise all dependencies in $\textit{Affinity}(n)$ is considered to be amenable for movement, called an *amenable test*. In the context of Algorithm 1, affinity can be implemented by modifying line 3 to skip tests that exercise all dependencies in $\textit{Affinity}(n)$. Our experimental results show that affinity helped exclude a large number of tests from our analysis, and it also provided more logical suggestions. Furthermore, we found that developers tend to agree with the affinity computed for each test node (Section V-D). As per the developer feedback, our technique is able to identify the affinity correctly for 99% of the test nodes in our subjects.

Subject	KLOC	# Build Nodes	# Test Nodes	# Tests	# Build Count Entries
ProjA	468	431	135	7,228	297,370
ProjB	926	416	90	5,870	127,030
ProjC	1,437	574	111	7,667	810,030
ProjD	3,366	1,268	184	12,918	6,522,056
ProjE	29,058	8,540	173	17,110	34,486,308
Overall	35,255	11,229	693	50,793	42,242,794
Average	7,051	2,245	138	10,158	8,448,558

TABLE II
STATISTICS OF SUBJECTS USED IN OUR EVALUATION

C. Output

TestOptimizer generates an HTML report with all suggestions that can help reduce the number of test executions. The report displays metrics concerning the number of test executions with the current placement of tests and the build count for the current test nodes. For each test node with tests amenable for movement, the report suggests how to split the test node to reduce the number of test executions. Furthermore, the report also suggests any existing test nodes where the tests can be moved into instead of making a new test node. Existing test nodes are suggested only when the test node shares the exact same dependencies as the tests to be moved. The suggestions are shown in a ranked order starting with the test nodes that achieve the highest reductions. This ranked list can help developers in prioritizing their effort. For completeness, the report also includes the test nodes that were found to not contain any tests amenable for movements as to give the developer a more complete picture.

V. Evaluation

In our evaluation, we address the following three research questions:

- **RQ1.** How many test executions can be saved by applying TestOptimizer suggestions?
- **RQ2.** How scalable is TestOptimizer in handling large real-world projects?
- **RQ3.** What is the developer feedback for the suggestions of TestOptimizer?

A. Experimental Setup

We applied our technique on five medium to large proprietary projects that use CloudBuild as the underlying build system. Table II shows the statistics of our subjects. For confidentiality reasons, we refer to our subjects as ProjA, ProjB, ProjC, ProjD, and ProjE. The row "Overall" is the sum of all the values in each column. The row "Average" is the arithmetic average of all the values in each column. All subjects primarily use C# as the main programming language, but also include some code in other languages such as C++

or Powershell. Column 2 shows the size of C# code in each subject. As shown in the table, our subjects range from medium-scale (468 KLOC) to large-scale projects (29,058 KLOC). Column 3 shows the number of build nodes, and Column 4 shows the number of test nodes. Column 5 shows the number of (manually written) tests in each subject.

For each subject, we collect historical data about the number of times sets of build nodes are built in a previous range of time. CloudBuild maintains this information about each build in a SQL Server database. Using this database, we computed the "Build Count" information for sets of build nodes during the time period of 180 days starting from Feb 1^{st}, 2016. Column 6 shows the number of table entries in the database for each subject, where each entry is a set of build nodes for the subject along with the number of times those build nodes were built together. In our running example from Figure 1, this number would correspond to the number of entries in the box titled "Build Count" for each subject. The historical data is over a period of 180 days for all subjects, except ProjE. We could not collect 180 days worth of historical data for ProjE, our largest subject, due to out-of-memory errors, since our tool performs all in-memory computations. Therefore, for ProjE, we used only 30 days worth of historical data (from the same start date).

The split criterion we used as a configuration option to our technique is to split a test node only if the reduction is at least $2,000$ test executions (setting the threshold value c to be $2,000$). We used this criteria based on our discussions with the developers of our subjects. Finally, although we compute the reduction based on the historical data over a range of time R in the past, we present to developers the results as potential savings for a future range of time R (as future savings is what matters more to developers), under the assumption that the past is a good indicator for the future.

B. Results

Regarding RQ1, Table III presents the results showing the reduction in terms of number of test executions after applying TestOptimizer. Column 2 shows the number of test nodes where TestOptimizer found amenable tests, based on affinity (Section IV-B). Test nodes with amenable tests are *amenable test nodes*. Column 3 shows the number of amenable tests. The percentage of amenable test nodes range from 5.55% (5 / 90 in ProjB) to 17.34% (30 / 173 in ProjE). These results indicate that developers often ensure that tests are placed in the correct test node. However, the fact that there are tests in incorrect test nodes suggests that developers can still make mistakes as to where the tests belong to, especially if they lack a global view of the project, so there is still room for improvement. Up to 3,946 tests across all subjects can be moved as to reduce the number of test executions.

Column 4 shows the number of tests that were actually suggested to be moved by TestOptimizer, meaning their movement can provide a substantial reduction in the number of test executions. Column 5 shows the number of new test nodes that need to be created for the tests to be moved into (not including any existing test nodes that already exactly share the dependencies of the tests to be moved). Column 6 shows the number of test executions (in millions) for the original placement of tests based on historical data. Columns 7-9 show the reductions if developers were to implement the suggestions. Column 7 shows the reduction in terms of number of test executions, while Column 8 shows the percentage of reduction when compared against the original number of test executions (Column 6). However, these columns show the reduction relative to the number of test executions of *all* test nodes in the project; there are many test nodes that TestOptimizer found as non-amenable. Column 9 also shows percentages but compared against the number of test executions concerning *only* the amenable test nodes, essentially compared against only the test nodes that TestOptimizer can actually suggest movements from. When considering only the amenable test nodes, we see the percentage of reduction in number of test executions is higher compared with the reductions based on all the test nodes. For the percentages in the table, the row "Overall" is computed as the total reduction in number of test executions across all subjects over the total number of original test executions. "Average" is the arithmetic average of all the percentages. To address RQ1, TestOptimizer can help reduce millions of test executions (up to 2,377,208) among our subjects. By considering only the amenable test nodes, these reductions range from 5.49% to 16.54% among our subjects.

Regarding RQ2, Column 10 shows the time (in minutes) taken by TestOptimizer to analyze the historical data and to run through its algorithm to suggest movements for each subject. Our results show that the analysis time ranges from 0.44 minutes up to 994.32 minutes ($\approx$16 hours). The time seems to be a factor of the number of build nodes, test nodes, tests, and the amount of historical data available. We find this time can still be reasonable as we envision our technique to be run infrequently. We also envision TestOptimizer can be run incrementally by only analyzing newly added tests as to suggest to developers the best test node to place new tests; TestOptimizer can work quickly when analyzing only a small number of tests. Furthermore, TestOptimizer is still a prototype, and it can be improve by implementing lazy loading of data and caching previous computations to avoid repeated calculations of build count.

C. Spurious Dependencies

In our experience, we found that some of the test nodes have additional developer-specified dependencies that are not required by any tests inside that test node. We refer to such dependencies as *spurious dependencies*. The primary reason for spurious dependencies could be the evolution of the application code. More specifically, developers could have originally specified a test node in the build system to have some dependencies, but after code changes the tests were moved around so that the test node no longer depends on some of those developer-specified dependencies. As such, when a spurious dependency is built, the dependent test node is built unnecessarily, causing a number of wasteful test executions.

Subject	# Amenable Test Nodes	# Amenable Tests	# Moved Tests	# New Test Nodes	# Orig Test Execs (millions)	Reduced Test Execs			Time to Analyze (min)
						#	All %	Amenable %	
ProjA	20	1,343	1,047	25	9.88	635,815	6.43	10.56	12.93
ProjB	5	328	291	4	2.36	46,393	1.96	5.49	0.44
ProjC	15	364	333	12	17.10	779,523	4.56	13.87	11.89
ProjD	15	358	312	13	63.56	2,377,208	3.74	7.76	315.41
ProjE	30	1,553	1,250	28	18.26	1,498,045	8.20	16.54	994.32
Overall	85	3,946	3,233	82	111.16	5,336,984	4.80	10.22	1,334.99
Average	17	789	646	16	22.23	1,067,396	4.97	10.84	266.99

TABLE III
RESULTS OF APPLYING TESTOPTIMIZER ON OUR SUBJECTS

Subject	Spurious Deps		# Test Execs (millions)	Reduced Test Execs	
	# Test Nodes	# Deps		#	%
ProjA	8	12	10.04	796,250	7.93
ProjB	35	44	2.40	82,498	3.43
ProjC	36	43	20.40	4,842,269	20.00
ProjD	68	116	69.68	8,497,928	12.20
ProjE	115	190	24.21	7,445,681	30.75
Overall	262	405	126.73	21,664,626	17.09
Average	52	81	25.34	4,332,925	14.86

TABLE IV
RESULTS WHEN CONSIDERING SPURIOUS DEPENDENCIES

Subject	Sugg.	Accepted Sugg.	Valid/Rejected Sugg.	Invalid Sugg.
ProjA	20	16	3	1
ProjB	5	4	1	0
ProjC	5	5	0	0
ProjD	15	13	0	2
Overall	45	38	4	3
Average	11	9	1	1

TABLE V
FEEDBACK FROM DEVELOPERS OF FOUR SUBJECTS

A spurious dependency can be simply removed and all the tests within the test node will still run properly. Given TestOptimizer, spurious dependencies are the build nodes declared as dependencies in the build specification for a test node but are not covered by the test node's tests. Table IV shows the number of test nodes that have spurious dependencies in each subject (Column 2) along with the total number of spurious dependencies those test nodes depended on (Column 3). For some subjects, such as ProjB and ProjE, we see that almost 50% of all test nodes have spurious dependencies.

Although detecting spurious dependencies is not a core contribution, it is an additional advantage of our technique. The reduction in number of test executions after both moving tests and removing spurious dependencies is the reduction developers would actually obtain. Table IV shows the effects of having spurious dependencies and the reduction in the number of test executions for each subject. Column 4 shows the number of test executions (in millions) for each subject using developer-specified dependencies for each test node. The number of test executions is higher than the values shown in Table III, as they include the effects of these spurious dependencies. Columns 5 and 6 show the reduction in number of test executions for each subject after the suggested test movements from TestOptimizer relative to the number of test executions obtained using the developer-specified dependencies. Compared to the reductions seen in Table III, the reduction is higher. We also see cases where spurious dependencies seem to be a big problem. To give an estimate of how the reduction in the number of test executions map to savings in terms of machine time, we calculated the average time (154 ms) across all the tests of the five subjects. Using this average test execution time and the reduction of number of test executions, our results show that TestOptimizer can help save 38.61 days of machine time.

D. Developer Feedback

Regarding RQ3, we approached the developers of our subjects to receive their feedback on suggested test movements. We received feedback from the developers of four of our subjects: ProjA, ProjB, ProjC, and ProjD, and yet to receive the feedback for the remaining subject. Table V presents the results for each of these four subjects. Column 2 shows the number of test movement suggestions reported by our tool, which is counted by the number of test nodes where our tool found tests amenable for movement. Column 3 shows the number of suggestions accepted by developers, and they intend to implement the suggestions. Column 4 shows the number of suggestions that the developers considered as valid, but they do not intend to implement the suggestions. Finally, Column 5 shows the number of suggestions that the developers considered as invalid. As shown in our results, 84.44% of the suggestions were indeed accepted by the developers. Furthermore, among the 16 accepted suggestions in ProjA, developers already implemented **six** of the suggestions.

Overall, we received highly encouraging feedback from the developers. The feedback also helped us understand how code evolution resulted in wasteful test executions. For example, a developer informed us that some application code was moved from one build node to a new one as a part of a major refactoring. However, the related test code was not moved into the relevant test node. Since the original test node still had a dependency on the new build node, tests continued to execute properly, but the tests would also execute when any other dependency the original test node had was built. After analyzing our suggestions, the developer felt that our technique could also be quite helpful finding a better organization of the tests. This response is encouraging, since it demonstrates TestOptimizer's ability in addressing issues beyond wasteful test executions. Another feedback was that developers may not be aware of an existing test node better suited for their

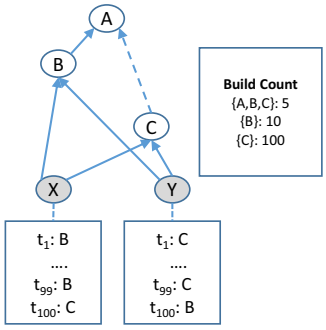

Fig. 3. An example dependency graph illustrating the scenario where developers tend to accept our suggested movements

tests, especially with many developers and many nodes in the project. Therefore, developers tend to place tests in some test node they are familiar with; they later add more dependencies to that test node, eventually leading to wasteful test executions. Developers also appreciated the idea that our suggestions can help break edges from test nodes to build nodes in the dependency graph, thereby reducing the build time along with the test execution time (breaking edges prevents test nodes from being built, which itself takes some time in the build system). The developers we approached also asked that we provide these reports at regular intervals (e.g., once per week), so they can monitor and improve the health of their project.

Accepted: We present the two common scenarios under which developers tend to accept our suggestions (Column 3 in Table V). We use the dependency graph shown in Figure 3 as an illustrative example. In the graph, test node `X` that is dependent on build nodes `B` and `C`. The figure also shows the build counts of `B` and `C`, where the build count of `C` is much greater than the build count of `B`. The figure shows that the `X` includes 100 tests, where most of them are dependent on `B` and only a few (just one in this example) depends on `C`. The primary issue is that most of the tests in `X` are wastefully executed due to the high build count of `C`. In this scenario, our technique typically suggests to move the tests in `X` into a test node that is dependent only on `B`. We noticed that developers tend to accept our suggestions in these scenarios, since they help reduce a large number of test executions in test nodes such as `X`. Another common scenario where developers often accepted our suggestion is when there already exist a test node with the exact same set of dependencies where the tests can be moved. In this scenario, the effort involved in implementing the suggestion is minimal, i.e., developers just need to copy-paste the tests into the existing test node.

Valid/Rejected: We present the common scenario under which developers considered that the suggestions are valid, but not willing to implement those suggestions (Column 4 of Table V). We use the same dependency graph in Figure 3 as an illustrative example, this time focusing on test node `Y` instead. In `Y`, the majority of the tests have a dependency on `C` instead of `B`. Although this scenario could result in wasteful test executions with respect to tests such as t_{100}, it is not as

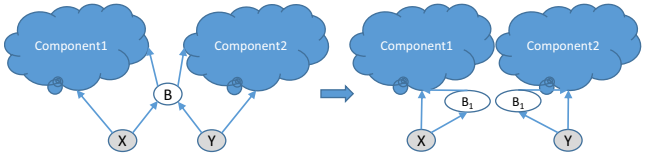

Fig. 4. An example scenario where our suggestion leads to a higher-level refactoring of a build node

significant (since only a few tests are impacted) compared to the dependency graph shown in Figure 3. Due to the lower benefit in implementing the suggestion, developers seemed reluctant in moving such tests, especially when there is no existing test node the tests can simply be moved into.

Invalid: Regarding the suggestions that are considered as invalid (Column 5 of Table V), we found that they are primarily due to implementation issues. TestOptimizer relies on code coverage to collect test traces, in this case using Magellan. In case a test exercises a binary via only a constant or refers to a class using constructs such as *typeof*, we noticed that Magellan does not collect the necessary dependency, a limitation in the code coverage tool. Due to such missing dependencies, our tool suggested invalid movements that were rejected by the developers. In the future, we plan to explore further on how to collect dependencies in these scenarios.

Interesting Scenario: We finally present an interesting scenario where our suggestion led to a higher-level refactoring of a build node. Figure 4 shows two major components in the project, where each component contains several build nodes. There are two test nodes `X` and `Y`, which are intended to test build nodes in Component1 and Component2, respectively. However, due to the build node `B`, changes in Component1 can trigger tests in `Y`, and similarly changes in Component2 can trigger tests in `X`. In an attempt to implement our suggestion in `X`, instead of splitting `X`, the developer split the build node `B` into two build nodes B_1 and B_2. The resulting dependency graph is shown in the figure, where the split helped remove dependencies between the build nodes and test nodes. This is very encouraging feedback, since our suggestions can help developers make insightful decisions in removing major unnecessary dependencies in the project. In our future work, we plan to explore how to automatically suggest these higher-level refactorings for build nodes.

E. Verifiability

Due to legal issues, we can neither reveal the names of our subjects nor release the TestOptimizer code. We can share anonymized output of TestOptimizer for each subject[1].

VI. Related Work

Our work is related to existing work on regression test selection [7], [10], [11], [17], [19], [24]. Given a change, these techniques aim to select a subset of affected tests. The key idea of these techniques is to maintain some metadata, such as statements covered by tests on the previous version,

[1]http://mir.cs.illinois.edu/awshi2/testoptimizer

and leverage this metadata to select a subset of tests that are affected by the change. Recent work by Gligoric et al. [9] found regression test selection that tracks dependencies at the coarse granularity level of files to be effective. The build system our work focus on performs regression test selection at the even coarser granularity level of modules.

Harrold et al. [10] specify that selection can be based on coarser-grained entities, such as methods, classes, or modules. A major advantage of module-level regression test selection is that it is extremely light-weight and highly scalable, which is why it is widely used in practice. TestOptimizer improves this regression test selection that naturally comes with the build system. In contrast to previous finer-grained regression test selection techniques, our technique does not require any additional information beyond what is needed for building the modules. Although previous techniques can further reduce the number of test executions by doing the selection on each test node, we argue that these previous techniques do not help as much due to the following reasons. First, existing techniques assume that tests are executed at the end of the build. Therefore, even if there exists a single test (in a test node) that is impacted by the change, the entire test node still needs to be built. In contrast to that, our technique can help skip the build of that test node as well. Second, existing techniques require storage and maintenance of metadata, which needs to be updated along with the changes in the underlying modules. This aspect adds non-trivial overhead in the case of millions of test executions. Instead, our technique does not require any additional information beyond what build system already saves. Finally, our technique can also be extended to refactor application code as well to further fine-tune the dependency graph; we plan to focus on this in our future work.

There has been some previous work on improving regression testing in the industrial environment. Elbaum et al. [6] proposed improving regression testing at Google through a combination of test selection in a pre-submit phase and test prioritization in a post-submit phase. Instead of collecting coverage for test selection and prioritization, they considered historical test failures to determine what tests to run and what order to run them. Herzig et al. [12] proposed a test-selection technique based on cost model. Their technique dynamically skips tests when the expected cost of running a test exceeds the expected cost of not running the test. All tests are still run at least once before the code is released, so defect detection is merely delayed (but can potentially be more costly to fix). These previous techniques complement our technique. In contrast to their techniques, we can guarantee that some tests can be ignored by moving to a different test node since they are not relevant for the given change. Therefore, we envision that developers can use our technique to avoid running tests that are not relevant for a change and then use their techniques to further reduce the number of test executions.

There exists work on techniques to improve the efficiency of builds [14], [15], [20]. Telea and Voinea [21] developed a tool that decomposes header files (in C/C++ code) to remove performance bottlenecks. Morgenthaler et al. [16] developed a tool that ranks libraries based on their utilization rank to identify under utilized nodes that can be refactored.

Our work is closely related to Vakilian et al. [22] that attempts to decompose build nodes into two or more nodes to improve build times. Their work shares a similar goal with our work, i.e., avoid unnecessary building of nodes when dependencies change. TestOptimizer significantly differs from their technique due to the following reasons. First, we focus on moving individual tests in test nodes into either existing or new test nodes. We allow movements to existing test nodes, as opposed to their technique that only creates new nodes. Second, since test nodes are essentially the leaf nodes in the dependency graph, we do not have to update dependencies for any child nodes. Third, their technique does not take historical information into consideration, i.e., they assume all nodes are built in every build. In contrast to that, we take historical information into consideration to ensure that tests are moved away from test nodes that have a high historical build count. Finally, their techniques identifies dependencies statically, whereas our technique uses dynamic analysis (code coverage), which is more precise. Although, it would be ideal to evaluate both techniques on common subjects, it is not practical since both are proprietary applications and are targeted for different technologies.

VII. Conclusions

In this paper, we present TestOptimizer, a technique that helps reduce wasteful test executions due to suboptimal placement of tests. We formulate the problem of wasteful test executions and develop an algorithm for reducing the number of wasteful test executions. We have implemented our technique in a prototype tool on top of Microsoft's CloudBuild. We show the effectiveness of TestOptimizer by applying it on five proprietary projects. Our results show that our technique can reduce 21.66 million test executions (17.09%) across all our subjects. Furthermore, developers of four of our subjects accepted and intend to implement ≈80% of our suggestions; developers have already implemented some of these suggestions as well. Beyond saving machine resources, the reduction in test executions can also help reduce the developer effort in triaging the test failures, if these irrelevant tests are flaky. In future work, we plan to use TestOptimizer incrementally to suggest ideal placements for newly added tests, soon after they are added. We also plan to extend TestOptimizer to refactoring build nodes as well to improve build times. Finally, we plan on exploring large-scale automated refactoring techniques [23] for applying our suggestions.

Acknowledgments

We would like to thank Vimuth Fernando, Wing Lam, Owolabi Legunsen, Nicholas Lu, Muhammad Suleman Mahmood, Darko Marinov, Sasa Misailovic, Amarin Phaosawasdi, and the anonymous reviewers for their feedback on previous drafts of this paper. August Shi did part of the work for this paper as an intern at Microsoft. August was also supported by the NSF Grant Nos. CCF-1409423 and CCF-1421503.

REFERENCES

[1] Bazel. http://bazel.io/.

[2] Fastbuild. http://www.fastbuild.org/docs/home.html.

[3] Magellan code coverage framework. http://research.microsoft.com/en-us/news/features/magellan.aspx.

[4] Visual studio team test. https://msdn.microsoft.com/en-us/library/ms379625.aspx.

[5] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[6] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, pages 235–245, 2014.

[7] E. Engström, M. Skoglund, and P. Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In *ESEM*, pages 22–31, 2008.

[8] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula. CloudBuild: Microsoft's distributed and caching build service. In *ICSE*, pages 11–20, 2016.

[9] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, pages 211–222, 2015.

[10] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA*, pages 312–326, 2001.

[11] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *ICSM*, pages 362–367, 1988.

[12] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *ICSE*, pages 483–493, 2015.

[13] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.

[14] S. Mcintosh, B. Adams, and A. E. Hassan. The evolution of java build systems. *Empirical Softw. Engg.*, 17(4-5):578–608, 2012.

[15] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. Hassan. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Softw. Engg.*, pages 1–47, 2014.

[16] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for build debt: Experiences managing technical debt at Google. In *MTD*, pages 1–6, 2012.

[17] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–251, 2004.

[18] C. Prasad and W. Schulte. Taking control of your engineering tools. *IEEE Computer*, 46(11):63–66, 2013.

[19] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[20] R. Suvorov, M. Nagappan, A. Hassan, Y. Zou, and B. Adams. An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel. In *ICSM*, pages 160–169, 2012.

[21] A. Telea and L. Voinea. A tool for optimizing the build performance of large software code bases. In *CSMR*, pages 323–325, 2008.

[22] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni. Automated decomposition of build targets. In *ICSE*, pages 123–133, 2014.

[23] H. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. Large-scale automated refactoring using ClangMR. In *ICSM*, 2013.

[24] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Soft. Test Verif. Reliab.*, 22(2):67–120, 2012.