Automated Decomposition of Build Targets

Authors: Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, Vahab Mirrokni
Presenter: Wentao Fan

Problem Statement

- **Underutilized targets** result in less modular code, overly large artifacts, slow builds, and unnecessary build and test triggers.
- **Manually decomposing** a target is tedious and error-prone.

Solution

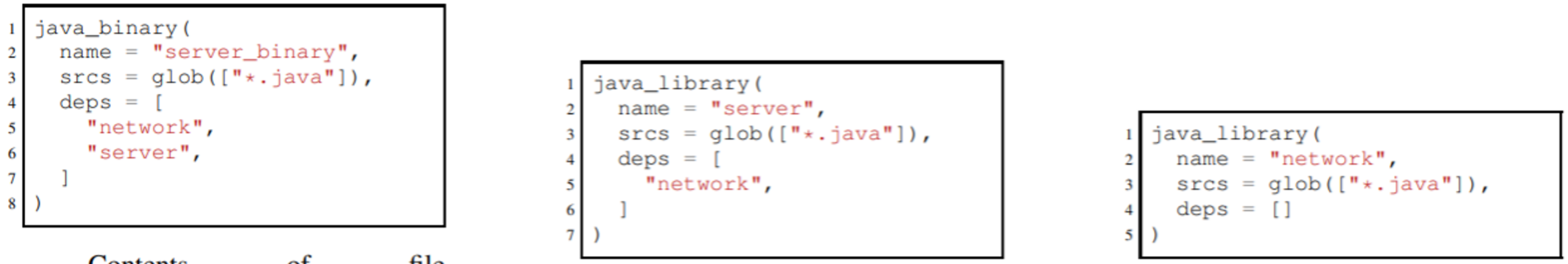
- Quantify the benefit of a decomposition in terms of the **number of triggers** that it saves.
- Formalize the decomposition problem as a **graph problem** and prove that finding the best decomposition is NP-hard.
- Present **DECOMPOSER**—a tool for decomposing targets.
- Present **REFINER**—a tool that refactors build specifications to take advantage of a decomposition.

Build System

- A **build system** is responsible for transforming source code into libraries, executable binaries, and other artifacts. The build system takes as input a set of targets that programmers declare in build files.
- Ensure that the **required dependencies of the target** are built.
- **Build the desired target** from its sources and dependencies.
- The final artifact depends on the **kind of the target**.

Build System: Build Target

- Programmers have to specify four attributes in the specification of a target: name, kind, source files, and dependencies.



```
java_binary(
  name = "server_binary",
  srcs = glob(["*.java"]),
  deps = [
    "network",
    "server",
  ]
)
```

(a) Contents of file `server_binary/BUILD`

```
java_library(
  name = "server",
  srcs = glob(["*.java"]),
  deps = [
    "network",
  ]
)
```

(b) Contents of file `server/BUILD`

```
java_library(
  name = "network",
  srcs = glob(["*.java"]),
  deps = []
)
```

(c) Contents of file `network/BUILD`

Fig. 1: Three `BUILD` files that declare targets `server_binary`, `server`, and `network` shown in Figure 2. Attribute `name` specifies the name of the target. The `srcs` attribute specifies the source files of the target. The expression `glob(["*.java"])` resolves to all Java files in the enclosing directory of the `BUILD` file. The `deps` attribute lists the targets that need to be built to compile the source files of the target.

Build System: Dependency Graph

- Build Graph (Target-level Dependencies)
- Cross References Graph (File-level Dependencies)

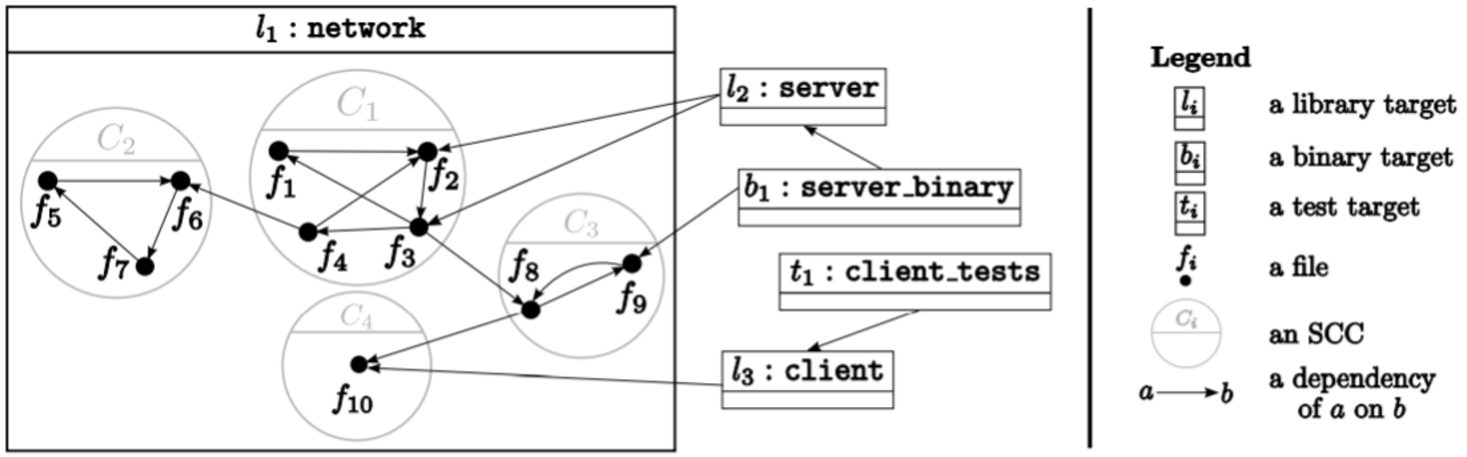


Fig. 2: A contrived graph that illustrates both target-level and file-level dependencies for an underutilized target named `network` and denoted as l_1 for brevity. C_i represents a strongly connected component (Section VI-A) of the cross references graph of l_1.

Build System: Dependency Graph

- **Definition 1**: The cross references between the source files of a target τ can be represented as a graph $G(\tau)$, called the cross references graph of τ. The vertices of $G(\tau)$ are members of $S(\tau)$ and there is an edge $(f1, f2) \in E(G(\tau))$ if and only if $f1 \rightarrow f2$
- The graph G(l1) is a subgraph of the graph shown in Figure 2. In this example, G(l1) consists of ten vertices corresponding to the files of l1 and the dependency edges between these files.

Build System: Continuous Integration(CI)

- The CI system computes the set of targets that may be affected by a code change.
- In Figure 2, if any of the source files of network change, the CI system will invoke the build system to build the targets that transitively depend on network and run the tests included in the test targets that transitively depend on network.

Underutilized Targets

- If a target has some dependent targets that need only a subset of its source files, the authors consider the target as an **underutilized target**.
- Underutilized targets lead to less modular software, larger binaries, slower builds, and unnecessary builds and tests triggered by the CI system.
- **Dependency Granularity**: The finest levels of dependencies that existing build systems track are target-level dependencies.

Target Decomposition

- **Target decomposition**: A refactoring to remove underutilized targets is to decompose them into smaller targets. The smaller targets are called **constituent targets**.
- For the example in Section “Underutilized Targets”, this refactoring would decompose the underutilized target network into 2 constituent targets network_a and network_b such that S(network_a) = S1, S(network_b) = S2 and network_a depends on network_b.
- **Decomposition Granularity**. Finer-grained decompositions can remove a larger number of unneeded dependencies.
- **Validity**. Let τ /[τ1, τ2] denote a decomposition of target τ into two constituent targets τ1 and τ2, and makes τ depend on both τ1 and τ2. A decomposition τ /[τ1, τ2] is valid if and only if τ2 $\nrightarrow$ τ1.

Target Decomposition

- **Trigger Saving**. The authors measure the benefit of a decomposition by the number of binary and test triggers that it saves. Let Δ(τ /[τ1, τ2]) denote the quantitative benefit of τ /[τ1, τ2]. Δ(τ /[τ1, τ2]) is referred as the trigger saving of τ /[τ1, τ2].
- **Definition 2**: D(τ) denotes the set of binary and test targets that transitively depend on target τ.
- Let p1 be the probability that a change affects only a file in S(τ1). Similarly, let p2 be the probability that a change affects only a file in S(τ2). Approximate p1 by |S(τ1)|/(|S(τ1)|+ |S(τ2)|) and p2 by |S(τ2)|/(|S(τ1)|+|S(τ2)|). These formula are approximations and not exact values.

Target Decomposition

- **Definition 3**: Δ(τ /[τ1, τ2]), the trigger saving of decomposition τ /[τ1, τ2], is:

$$p_1|\mathcal{D}(\tau_2) - \mathcal{D}(\tau_1)| + p_2|\mathcal{D}(\tau_1) - \mathcal{D}(\tau_2)|,$$

where

$$p_1 = \frac{|S(\tau_1)|}{|S(\tau_1)| + |S(\tau_2)|}, \quad p_2 = \frac{|S(\tau_2)|}{|S(\tau_1)| + |S(\tau_2)|}.$$

Target Decomposition

- Δ(τ /[τ1, τ2]) is the expected number of binary and test targets that won’t be triggered after applying the decomposition and updating the dependents of τ. The greater Δ(τ /[τ1, τ2]) is, the more triggers will be saved by the decomposition

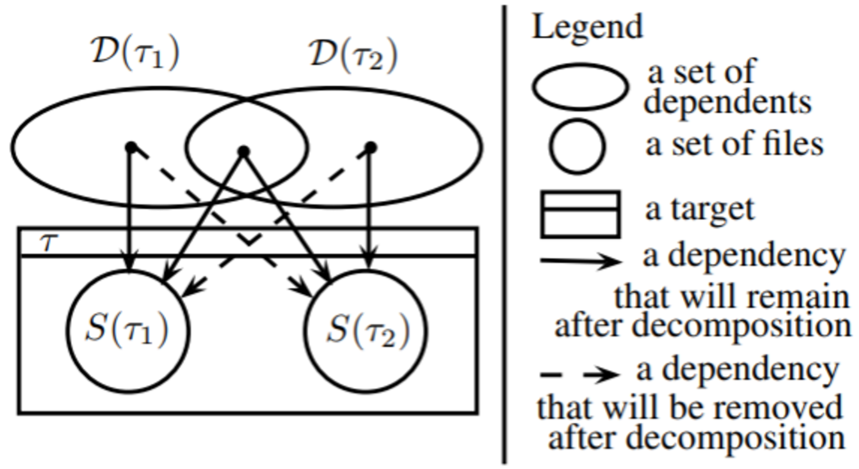


Fig. 3: Decomposition $\tau/\langle\tau_1,\tau_2\rangle$ removes unneeded dependencies (dashed arrows) that cause unnecessary build or test triggers. $\Delta(\tau/\langle\tau_1,\tau_2\rangle)$ is the average number of triggers that the decomposition would save every time a change affects the files in only $S(\tau_1)$ or only $S(\tau_2)$.

Hardness of Decomposition

- **Theorem**: Given a target τ, finding the decomposition τ /[τ1, τ2] that maximizes Δ(τ /[τ1, τ2]) is an NP-hard problem.
- **Proof**: The authors prove NP-hardness by showing a reduction from the maximum clique problem in graph theory. The proof is included in an accompanying technical report

Decomposition Algorithm

This paper proposes an efficient greedy algorithm that finds effective decompositions in practice. The algorithm suggests a decomposition in the following steps:

- Compute the strongly connected components (SCCs) of the cross references graph of the given target.
- Find the binary and test targets that transitively depend on each SCC.
- Partition the SCCs of the target into two sets with a goal of maximizing the trigger saving (Definition 3).
- Update the build specifications to apply the decomposition.

Decomposition Algorithm: SCC

- A directed graph G is strongly connected if and only if for each pair of vertices v1, v2 ∊ V (G), v1->Gv2 and v2->Gv1. A strongly connected component of a graph G is a maximal subgraph of G that is strongly connected.
- the authors refer to a strongly connected component as an **SCC(Strongly Connected Components)**.
- **Condensation Graph**. If each SCC of G is contracted to a single vertex, the resulting graph is the condensation graph of G denoted as C(G). In Figure 2, C(G(I1)) has four vertices C1, C2, C3, and C4 and 3 edges.
- The algorithm proposes a decomposition to **only 2 constituent targets** by default, because finegrained decomposition has some problems(Section “Target Decomposition”).

Decomposition Algorithm: Dependents

- A decomposition τ /[τ1, τ2] is **ideal if it maximizes Δ(τ /[τ1, τ2])** (Definition 3). Δ(τ /[τ1, τ2]) depends on D(τ1) and D(τ2) (Definition 2), i.e., the set of binary and test targets that transitively depend on τ1 and τ2, respectively.
- D(τ, C) is the set of binary and test targets that transitively depend on SCC C of G(τ).
- This paper computes D(τ), the set of binary and test targets that transitively depend on τ by taking the union of D(τ, C) for all SCC C of G(τ).

Decomposition Algorithm: Unifying Components

- **Unification** is an operation that takes two components C1 and C2 of G(τ) and creates a new component C such that S(τ, C) = S(τ, C1) ∪ S(τ, C2).
- If C1 and C2 are unified to C, the authors will have D(τ, C) = D(τ, C1) ∪ D(τ, C2).
- Figure 4 shows 2 subsequent unifications applied on the condensation graph of target l1 in Figure 2.

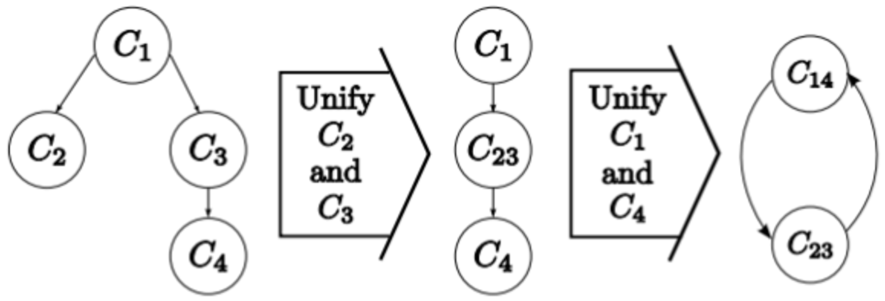


Fig. 4: Unifying the components of the cross references graph of target `network` in Figure 2. The graph on the left is $\mathcal{C}(G(\texttt{network}))$. First, C_2 and C_3 are unified to C_{23}. Then, C_1 and C_4 are unified to C_{14}. The final condensation graph (on the right) is invalid because it has a cycle. As a result, a decomposition corresponding to C_{14} and C_{23} is invalid, too.

Decomposition Algorithm: Unifying Components

- **Iterative Unification**: After computing the SCCs of the cross references graph of a target, the algorithm iteratively unifies two components at each step until only two are left.
- Let δ(τ, C1, C2) be the cost of unifying components C1 and C2 of G(τ). Similar to Definition 3, δ(τ, C1, C2) is defined as:

$$p_1|\mathcal{D}(\tau, C_2) - \mathcal{D}(\tau, C_1)| + p_2|\mathcal{D}(\tau, C_1) - \mathcal{D}(\tau, C_2)|,$$

where

$$p_1 = \frac{|S(\tau, C_1)|}{|S(\tau, C_1)| + |S(\tau, C_2)|}, \quad p_2 = \frac{|S(\tau, C_2)|}{|S(\tau, C_1)| + |S(\tau, C_2)|}.$$

Decomposition Algorithm: Unifying Components

- **Avoiding Invalid Decompositions**: The unification algorithm as described above may produce invalid decompositions.
- In Figure 4, unifications can produce an **invalid** decomposition, because the targets introduce a **circular dependency** to the build graph.
- **Lemma 1**: Contracting two vertices that are adjacent in a topological ordering of a DAG results in another DAG.
- **Lemma 2**: Contracting two root vertices (i.e., vertices without incoming edges) or two leave vertices (i.e., vertices without outgoing edges) of a DAG results in another DAG.

Decomposition Algorithm: Constituent target

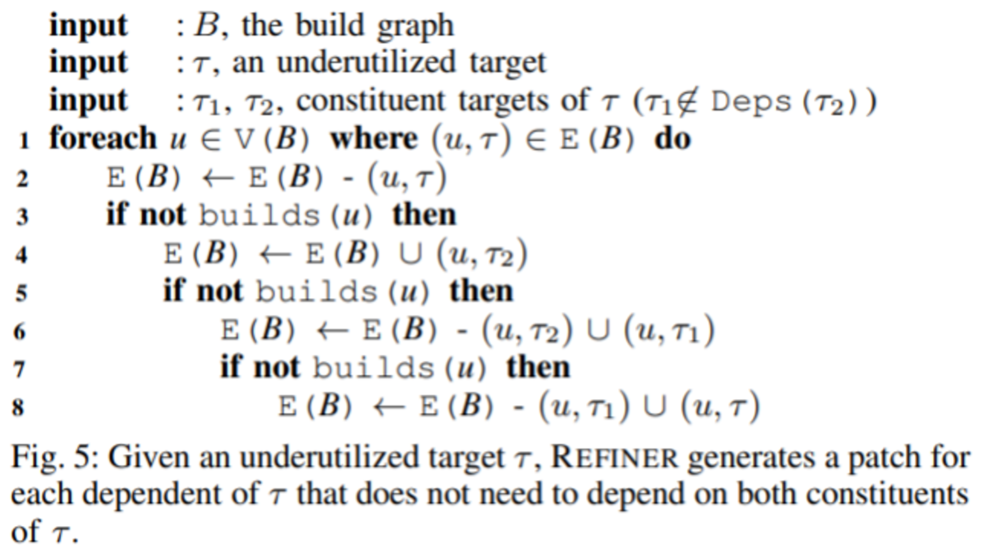
- Currently, rewriting the build specifications to introduce the constituent targets is **semi-automated**. The iterative unification of the components of τ terminates when only two components are left.
- Set $S(\tau)$ to $\emptyset$ and **specify the constituent targets**.
- Run a separate tool that **removes unneeded** dependencies and **converts dependencies to direct** ones.

Dependency Refinement

- **Dependency refinement**: To unleash the full benefits of a decomposition, the dependents of the target need to change to depend on only the needed constituent targets.
- **REFINER** is developed to automate the dependency refinement.
- Given an underutilized target, REFINER automatically and safely generates a patch.
- Figure 5 lists the pseudocode of REFINER. REFINER examines every dependent u of the given underutilized target τ(line 1).

Dependency Refinement

- First, REFINER **removes** the dependency of u on τ (line 2). If u continues to build successfully, this suggests that the dependency on u was **unneeded**.
- Otherwise, REFINER first tries a dependency on τ2 (line 4) and then τ1 (line 6). If u cannot be built successfully with a dependency on either τ1 or τ2, it means that u needs both τ1 and τ2.



```
input   : $B$, the build graph
input   : $\tau$, an underutilized target
input   : $\tau_1$, $\tau_2$, constituent targets of $\tau$ ($\tau_1 \notin$ Deps($\tau_2$))
1 foreach $u \in$ V($B$) where $(u, \tau) \in$ E($B$) do
2     E($B$) $\leftarrow$ E($B$) - $(u, \tau)$
3     if not builds($u$) then
4         E($B$) $\leftarrow$ E($B$) $\cup$ $(u, \tau_2)$
5         if not builds($u$) then
6             E($B$) $\leftarrow$ E($B$) - $(u, \tau_2) \cup (u, \tau_1)$
7             if not builds($u$) then
8                 E($B$) $\leftarrow$ E($B$) - $(u, \tau_1) \cup (u, \tau)$
```

Fig. 5: Given an underutilized target τ, REFINER generates a patch for each dependent of τ that does not need to depend on both constituents of τ.

Soundness

- Graphs(target-level and file-level dependency graphs) are **sound** when all the dependencies that appear in source and build files are included in the graphs.
- **Soundness of DECOMPOSER**. The paper shows that if the file-level and target-level dependency graphs are sound, the greedy decomposition algorithm will also be sound. The decomposition τ /[τ1, τ2] does not affect the target-level dependencies that do not involve τ, τ1, and τ2.
- **Soundness of REFINER**. The paper shows that REFINER is also sound if the target-level dependencies are sound. REFINER may change only the dependencies of each dependent u of the underutilized target τ.

Implementation

- **DECOMPOSER** is a Java program which gets the file-level dependencies of a target from a service.
- The **Facade design pattern** is employed to provide abstractions for the services that DECOMPOSER relies on.
- DECOMPOSER uses **FlumeJava** for analyzing targets in parallel.
- **REFINER** is a Python program that relies on the build system, the target-level dependencies, and a headless tool for rewriting build specifications.

Evaluation

- **RQ1 : What percentage of targets can be decomposed?**
- DECOMPOSER reported that 19,994 (50%) of the analyzed targets were decomposable.
- A target is decomposable if and only if its cross references graph has at least two SCCs.

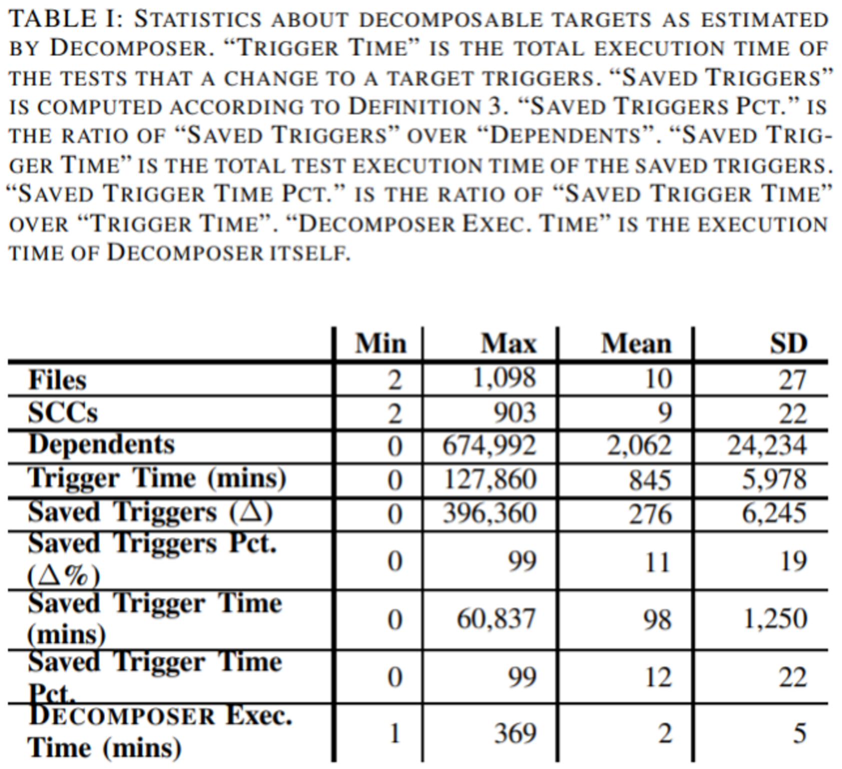


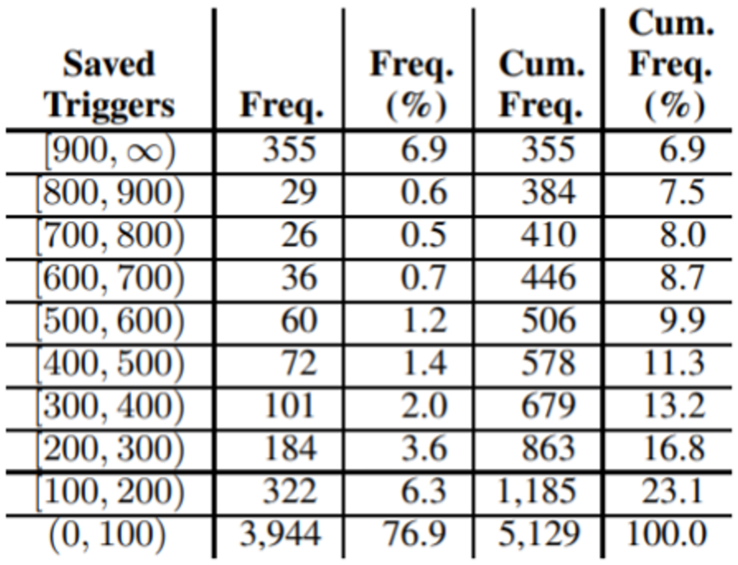
TABLE I: STATISTICS ABOUT DECOMPOSABLE TARGETS AS ESTIMATED BY DECOMPOSER. "TRIGGER TIME" IS THE TOTAL EXECUTION TIME OF THE TESTS THAT A CHANGE TO A TARGET TRIGGERS. "SAVED TRIGGERS" IS COMPUTED ACCORDING TO DEFINITION 3. "SAVED TRIGGERS PCT." IS THE RATIO OF "SAVED TRIGGERS" OVER "DEPENDENTS". "SAVED TRIGGER TIME" IS THE TOTAL TEST EXECUTION TIME OF THE SAVED TRIGGERS. "SAVED TRIGGER TIME PCT." IS THE RATIO OF "SAVED TRIGGER TIME" OVER "TRIGGER TIME". "DECOMPOSER EXEC. TIME" IS THE EXECUTION TIME OF DECOMPOSER ITSELF.

	Min	Max	Mean	SD
Files	2	1,098	10	27
SCCs	2	903	9	22
Dependents	0	674,992	2,062	24,234
Trigger Time (mins)	0	127,860	845	5,978
Saved Triggers (Δ)	0	396,360	276	6,245
Saved Triggers Pct. (Δ%)	0	99	11	19
Saved Trigger Time (mins)	0	60,837	98	1,250
Saved Trigger Time Pct.	0	99	12	22
DECOMPOSER Exec. Time (mins)	1	369	2	5

Evaluation

- **RQ2 : How effective are the decompositions that Decomposer suggests?**
- The authors measure the effectiveness of a decomposition by calculating the number (RQ2.1) and percentage (RQ2.2) of saved triggers and the duration (RQ2.3) and percentage (RQ2.4) of saved test execution time.

TABLE II: DISTRIBUTION OF THE NUMBER OF SAVED TRIGGERS



Saved Triggers	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
[900, ∞)	355	6.9	355	6.9
[800, 900)	29	0.6	384	7.5
[700, 800)	26	0.5	410	8.0
[600, 700)	36	0.7	446	8.7
[500, 600)	60	1.2	506	9.9
[400, 500)	72	1.4	578	11.3
[300, 400)	101	2.0	679	13.2
[200, 300)	184	3.6	863	16.8
[100, 200)	322	6.3	1,185	23.1
(0, 100)	3,944	76.9	5,129	100.0

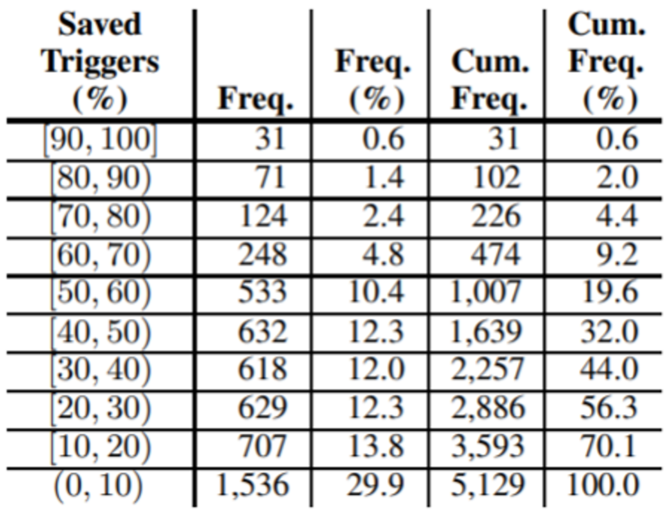
Evaluation

- **RQ2.1 : How many triggers can Decomposer save?**
- DECOMPOSER estimates that the decompositions it suggests for 26% of the decomposable targets (5,129 of 19,994) would save at least one trigger. Moreover, it found that on average decomposing a target saves 276 triggers (Table I) per change to the target. Table II shows that decomposing any one of 355 targets would save at least 900 triggers of the target.

Evaluation

- **RQ2.2 : What percentage of triggers can Decomposer save?**
- The decompositions suggested by DECOMPOSER save 11% of the triggers on average (Table I). Table III shows that decomposing any one of only 31 targets would save at least 90% of the triggers per change to the target.

TABLE III: DISTRIBUTION OF THE PERCENTAGE OF SAVED TRIGGERS

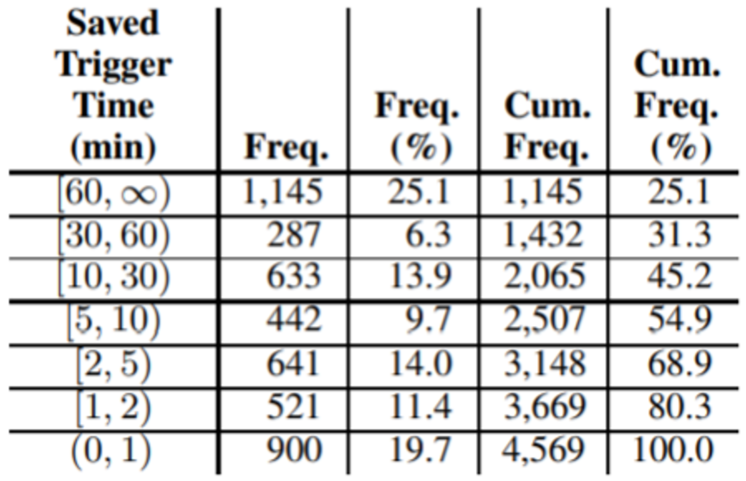


Saved Triggers (%)	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
[90, 100]	31	0.6	31	0.6
[80, 90)	71	1.4	102	2.0
[70, 80)	124	2.4	226	4.4
[60, 70)	248	4.8	474	9.2
[50, 60)	533	10.4	1,007	19.6
[40, 50)	632	12.3	1,639	32.0
[30, 40)	618	12.0	2,257	44.0
[20, 30)	629	12.3	2,886	56.3
[10, 20)	707	13.8	3,593	70.1
(0, 10)	1,536	29.9	5,129	100.0

Evaluation

- **RQ2.3 : How much test execution time can Decomposer save?**
- The decompositions that DECOMPOSER suggests save 98 minutes of the test execution time of a decomposable target on average (Table I). Table IV indicates that decomposing any of 1,145 targets would reduce the test execution time per change to the target by at least an hour.

TABLE IV: DISTRIBUTION OF SAVED TRIGGER TIMES

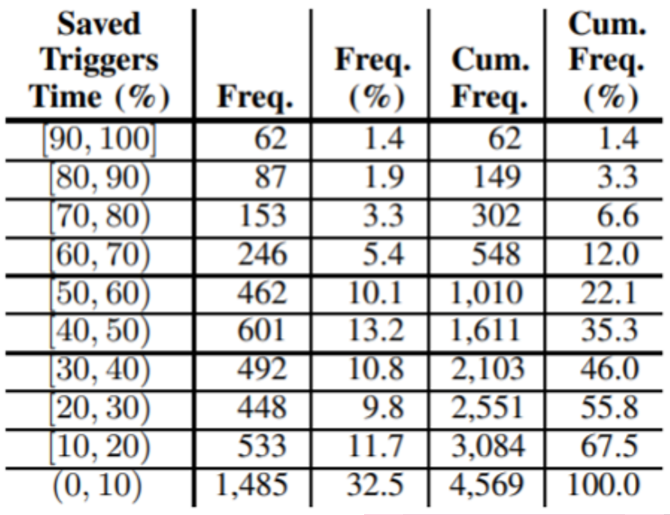


Saved Trigger Time (min)	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
[60, ∞)	1,145	25.1	1,145	25.1
[30, 60)	287	6.3	1,432	31.3
[10, 30)	633	13.9	2,065	45.2
[5, 10)	442	9.7	2,507	54.9
[2, 5)	641	14.0	3,148	68.9
[1, 2)	521	11.4	3,669	80.3
(0, 1)	900	19.7	4,569	100.0

Evaluation

- **RQ2.4 : What percentage of test execution time can Decomposer save?**
- On average, a decomposition that DECOMPOSER proposes for a target would save 12% of the execution time of the tests(Table I).
- Table V indicates that the decompositions for 1,010 targets would save at least 50% of the test execution time of each targets.

TABLE V: DISTRIBUTION OF THE PERCENTAGE OF SAVED TRIGGER TIME



Saved Triggers Time (%)	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
[90, 100]	62	1.4	62	1.4
[80, 90)	87	1.9	149	3.3
[70, 80)	153	3.3	302	6.6
[60, 70)	246	5.4	548	12.0
[50, 60)	462	10.1	1,010	22.1
[40, 50)	601	13.2	1,611	35.3
[30, 40)	492	10.8	2,103	46.0
[20, 30)	448	9.8	2,551	55.8
[10, 20)	533	11.7	3,084	67.5
(0, 10)	1,485	32.5	4,569	100.0

Evaluation

- **RQ3 : How efficient is Decomposer?**
- On average, DECOMPOSER analyzes a target in two minutes (Table I).
- Table VI shows the average breakdown of the execution time of each phase of DECOMPOSER.
- Each edge of this graph indicates a dependency of a target on another target.

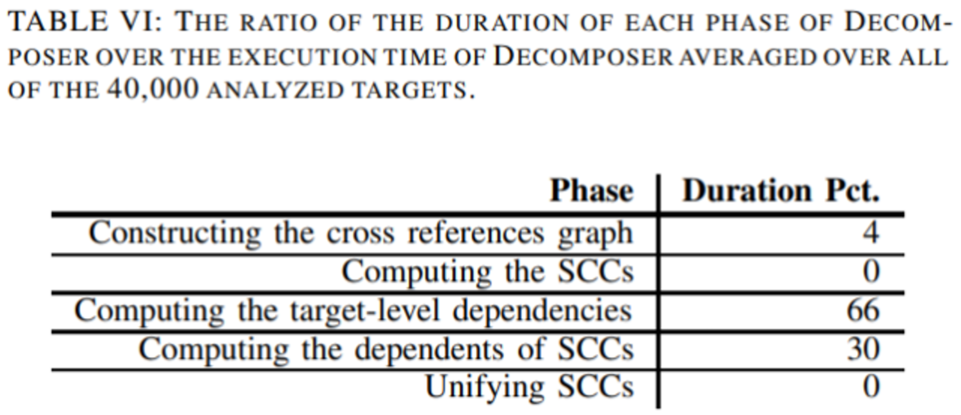


TABLE VI: THE RATIO OF THE DURATION OF EACH PHASE OF DECOMPOSER OVER THE EXECUTION TIME OF DECOMPOSER AVERAGED OVER ALL OF THE 40,000 ANALYZED TARGETS.

Phase	Duration Pct.
Constructing the cross references graph	4
Computing the SCCs	0
Computing the target-level dependencies	66
Computing the dependents of SCCs	30
Unifying SCCs	0

Evaluation

- **RQ4 : How receptive are programmers to the changes that Decomposer and Refiner propose?**
- As a preliminary evaluation, the authors selected seven targets for decomposition. DECOMPOSER estimated high trigger savings for these targets and the dependents of these targets declared all their direct dependencies.
- The authors submitted code changes based on the results of DECOMPOSER for these seven targets. Six code changes got reviewed, four of which got approved. Two code changes got rejected, because the reviewer expected the target to change rarely.

Related Work

- **Empirical Studies**. McIntosh etc al. [24] studied the version histories of ten projects and found that build maintenance accounts for up to 27% overhead on source code development and 44% overhead on test development. In another study of six open-source projects [23], McIntosh et al. found that the size of build files and source files are highly correlated.
- **Underutilized Targets**. In the authors' prior work, they discussed several code smells specific to build specification. They introduced a tool called Clipper that takes a binary target as input and ranks the libraries by utilization rates.

Related Work

- **Software Remodularization.** Researchers have developed tools for remodularizing legacy software. These tools employ clustering, search-based, or information retrieval techniques to find a set of modules that optimizes some metrics.
- **Analyzing, Visualizing, and Refactoring Makefiles**. MAKAO is a tool that visualizes Makefiles by analyzing their dynamic build traces. SYMake is a static analysis tool that can detect several code smells of Makefiles.
- **Test Selection.** It is used to select a subset of the tests to run on a future version of the program without compromising the fault-detection capability of the test suite.

Limitations and Future Work

- **Generalizability**: The evaluation results are limited to Java targets at Google.
- **Soundness**: Currently, the target-level dependencies miss the dependencies on generated targets, and the file-level dependencies include only the static dependencies.
- **Objective function**: DECOMPOSER uses the number of saved triggers as an objective function to find a decomposition. The authors plan to experiment with different objective functions in the future.
- **Decomposition Algorithm**: DECOMPOSER employs a greedy algorithm to suggest a decomposition, which is fast. However, finding an approximation algorithm with a provable guarantee of closeness to the optimal decomposition or proving the lack of such an algorithm are open problems.
- **Adoption**: About a dozen programmers at Google have used DECOMPOSER currently. The author's vision is to integrate DECOMPOSER into the programming workflow to gain a wider adoption.

Conclusion

- This paper focuses on a specific code smell of build specifications that the authors identified in Google's code base, namely, underutilized build targets.
- The authors a tool for large-scale identification and decomposition of underutilized build targets.
- The evaluation results show that the tool is both effective and efficient at
  - Estimating the benefits of decomposing build targets, and
  - Proposing decompositions of build targets.

Discussion

- Weakness on Evaluation?
- Possible improvements on Decomposer/Refiner?
- Is the complexity of the algorithm reasonable?

Thank You!!