Towards Efficient Build Ordering for Incremental Builds with Multiple Configurations

JUN LYU, Nanjing University, China
SHANSHAN LI*, Nanjing University, China
HE ZHANG, Nanjing University, China
LANXIN YANG, Nanjing University, China
BOHAN LIU, Nanjing University, China
MANUEL RIGGER, National University of Singapore, Singapore

Software products have many configurations to meet different environments and diverse needs. Building software with multiple software configurations typically incurs high costs in terms of build time and computing resources. Incremental builds could reuse intermediate artifacts if configuration settings affect only a portion of the build artifacts. The efficiency gains depend on the strategic ordering of the incremental builds as the order influences which build artifacts can be reused. Deriving an efficient order is challenging and an open problem, since it is infeasible to reliably determine the degree of re-use and time savings before an actual build. In this paper, we propose an approach, called BUDDI—**BU**ild **D**eclaration **DI**stance, for C-based and Make-based projects to derive an efficient order for incremental builds from the static information provided by the build scripts (*i.e.*, Makefile). The core strategy of BUDDI is to measure the distance between the build declarations of configurations and predict the build size of a configuration from the build targets and build commands in each configuration. Since some artifacts could be reused in the subsequent builds if there is a close distance between the build scripts for different configurations. We implemented BUDDI as an automated tool called BuddiPlanner and evaluated it on 20 popular open-source projects, by comparing it to a baseline that randomly selects a build order. The experimental results show that the order created by BuddiPlanner outperforms 96.5% (193/200) of the random build orders in terms of build time and reduces the build time by an average of 305.94s (26%) compared to the random build orders, with a median saving of 64.88s (28%). BuddiPlanner demonstrates its potential to relieve practitioners of excessive build times and computational resource burdens caused by building multiple software configurations.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Build Tool, Software Configurations, Incremental Builds

ACM Reference Format:
Jun Lyu, Shanshan Li, He Zhang, Lanxin Yang, Bohan Liu, and Manuel Rigger. 2024. Towards Efficient Build Ordering for Incremental Builds with Multiple Configurations. *Proc. ACM Softw. Eng.* 1, FSE, Article 67 (July 2024), 24 pages. https://doi.org/10.1145/3660774

*Corresponding author

Authors' addresses: Jun Lyu, Nanjing University, Nanjing, China, lvjun@smail.nju.edu.cn; Shanshan Li, Nanjing University, Nanjing, China, lss@nju.edu.cn; He Zhang, Nanjing University, Nanjing, China, hezhang@nju.edu.cn; Lanxin Yang, Nanjing University, Nanjing, China, lxyang@nju.edu.cn; Bohan Liu, Nanjing University, Nanjing, China, bohanliu@nju.edu.cn; Manuel Rigger, National University of Singapore, Singapore, Singapore, rigger@nus.edu.sg.

ACM 2994-970X/2024/7-ART67
https://doi.org/10.1145/3660774

1 INTRODUCTION

Practitioners use build systems such as GNU Make [17], Maven [42], Ninja [46], and Ant [5] to compile and package various artifacts into executable programs or deliverables. The emergence and prevalence of Continuous Integration (CI) have further expanded the importance of software builds, which guarantees software quality while enabling rapid iterative software updates [58]. However, the cost (*e.g.*, build time and computing resources) of software builds increases with the growing complexity and scale of software projects.

Software configurations are options to configure software and then create multiple variants, which complicate building software, exacerbating build time and computing resource costs [38, 41]. Different software configurations are available in many software projects in order to build a variety of software artifacts in varying environments and with diverse requirements. For example, configurations often allow building different versions (debug or release versions), including or excluding third-party libraries, enabling from a minimum set to all features, and adapting to different processors (x86 or ARM). Practitioners expect each configuration to work correctly. Therefore, it is common to build and test multiple software configurations in practice. For example, the Linux kernel has more than 14,500 compile-time options [34, 41]. To test the Linux kernel, KernelCI[1] needs to build the Linux kernel using thousands of compilation options every day [30, 41].

In this paper, we focus on building multiple configurations for C-based and Make-based projects—Make is one of the most widely used build systems [43–45]. In this context, each configuration typically modifies a subset of the software's configuration options. Typically, when users intend to build multiple configurations, they use a clean build, which builds a software configuration in a clean environment. For C-based and Make-based artifacts, configurations can be built from existing and completed configurations' builds, saving build time and computing resources by reusing artifacts from previous configurations' builds [49]. Only part of the source files would be overwritten when users set a new configuration option. Incremental build systems enable the new configuration to be built by rebuilding only the artifacts of the source files with the new timestamp. To the best of our knowledge, this is the first approach that has been proposed to reduce overall build time by strategically ordering incremental builds for the *software configuration set*, a set that includes multiple software configurations.

Example. To illustrate the potential benefits of identifying an efficient order for building multiple configurations, consider the following example from the *SQLite* project, and a configuration set {C_1 (disable-libtool-lock, with-pic), C_2 (enable-debug, with-readline-lib), C_3 (disable-amalgamation, with-readline-lib)}. The traditional clean build has a total build time of 348.34s. Assume that we incrementally build all configurations, and execute sequentially with *Order 1* [$C_1 \rightarrow C_2 \rightarrow C_3$], the overall build time is 236.12s. When changing the build order to *Order 2* [$C_3 \rightarrow C_2 \rightarrow C_1$], the overall build time drops down to 170.64s. The total build time for *Order 2* is 65.48s shorter than the total build time for *Order 1*. This example shows the advantage of ordering the configuration set in incremental builds. Such an efficient order can significantly reduce the overall build time of the configuration set [48].

Our core strategy is to order incremental builds to accelerate the build of the overall configuration set of C programs built by GNU Make, one of the most widely used build systems [43–45]. The rationale behind this is that build time can be reduced by incremental builds if the build declarations are similar for two configurations. Specifically, we use a lightweight static analysis of the build declarations to analyze similarities in software configurations and predict their build sizes, based on which to create an efficient order of incremental builds. The similarity-based ordering of incremental

[1] A community-led testing system to ensure the quality, stability, and long-term maintenance of the Linux kernel.

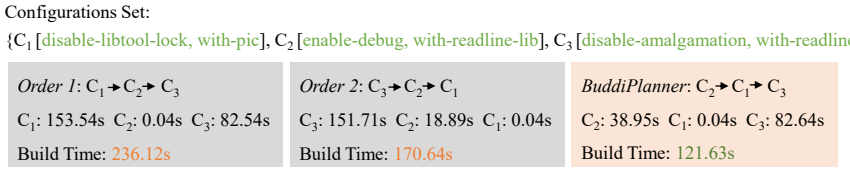

Fig. 1. Build Time Resulting from Various Build Orders on the SQLite Project

builds in software configurations can reuse artifacts from previous builds, and thus reduce the building workload.

We propose an approach—called BUDDI and implemented it as a tool called BuddiPlanner—to create a build order. BUDDI uses a distance metric to measure how similar the build targets and the build commands are between configurations. To quantitatively describe the distance, BUDDI analyzes the build scripts generated by each software configuration, the build targets, and the build commands between the two configurations. The closer the distances between the configurations, the more similar and less conflicting they are in the context of incremental builds. Moreover, BUDDI estimates the build size of each configuration, as we found that it is more efficient to start with the build of the smallest sized one among the configuration sets. This enables the reuse of artifacts from as many previous builds as possible. Overall, BUDDI is a best-effort approach based on an intuitive idea that works well in practice. BUDDI only creates a build order, it does not participate in the actual build. If the incremental build systems and build scripts are error-free, BUDDI does not result in incorrect builds.

We evaluated BuddiPlanner using 20 popular, active, and widely used projects. We selected the projects used in previous software configuration studies [49] and a further 15 configurable projects. These projects are C-based and GNU Make-based with various sizes (large (>5000) / medium (500–5000) / small (<500) number of files). For comparison, we implemented a best-effort baseline that randomly selects 10 build orders based on 20 randomly generated configurations, resulting in 200 build orders for the 20 projects (*cf.* Section 4.2). The evaluation results show that the order created by BuddiPlanner outperforms 96.5% (193/200) of the random orders in terms of build time, with an average reduction in build time of 305.94s (26%), with a median saving of 64.88s (28%). As for extensibility, BuddiPlanner can be extended to other build systems, such as Gradle [21], Ninja [46], and Bazel [7]. BuddiPlanner shows the potential to relieve the burden of practitioners dealing with long build times and the allocation of computing resources required when building with multiple software configurations. The main contributions of this paper are as follows.

- We propose an approach—called BUDDI—to order incremental builds to accelerate the building of the overall configuration set. This approach utilizes lightweight static analysis of build targets and commands in build declarations to examine configuration similarities, predict build sizes, and generate efficient orders for incremental builds.
- We implement BUDDI as an automated ordering tool called BuddiPlanner.
- We evaluate BUDDI on 20 projects. The order created by BUDDI outperforms the random orders by 96.5% in terms of build time, saving the build time of 305.94s (26%) on average, with a median saving of 64.88s (28%).

2 MOTIVATING EXAMPLE

In this section, we further explain how build orders affect build times (or result in different build times), as well as how BUDDI derives efficient orders for incremental builds using the motivating example presented in Section 1.

C_1 ["mksourceid", "sqlite3.h", "keywordhash.h", "lemon", "parse.c", "parse.h", "opcodes.h", "opcodes.c", "shell.c", "fts5parse.c", "fts5parse.h", "fts5.c", ".target_source", "sqlite3.c", "sqlite3.lo", "libsqlite3.la", "sqlite3"]	C_2 ["mksourceid", "sqlite3.h", "keywordhash.h", "lemon", "parse.c", "parse.h", "opcodes.h", "opcodes.c", "shell.c", "fts5parse.c", "fts5parse.h", "fts5.c", ".target_source", "sqlite3.c", "sqlite3.lo", "libsqlite3.la", "sqlite3"]	C_3 ["mksourceid", "sqlite3.h", "keywordhash.h", "lemon", "parse.c", "parse.h", "opcodes.h", "alter.lo", "analyze.lo", "attach.lo", "auth.lo", "backup.lo", "bitvec.lo", "btmutex.lo", "btree.lo", "build.lo", "callback.lo", "complete.lo", "ctime.lo", "date.lo", "dbpage.lo", "dbstat.lo", "delete.lo", "expr.lo", "fault.lo", "fkey.lo", "fts3.lo", "fts3_aux.lo", "fts3_expr.lo", "fts3_hash.lo", "fts3_icu.lo", "fts3_porter.lo", "fts3_snippet.lo", "fts3_tokenizer.lo", "fts3_tokenizer1.lo", "fts3_tokenize_vtab.lo", "fts3_unicode.lo", "fts3_unicode2.lo", "fts3_write.lo", "fts5parse.c", …]
Distance: $C_1 - C_2$: 206	$C_1 - C_3$: 2740	$C_2 - C_3$: 2832

Fig. 2. Build Targets and Distance from Three Configurations

C_1 "sqlite3": [
…
"-DSQLITE_THREADSAFE=1",
"-DSQLITE_HAVE_ZLIB=1",
"-DHAVE_READLINE=0",
"-DHAVE_EDITLINE=0",
"-DSQLITE_ENABLE_FTS4",
"-DSQLITE_ENABLE_RTREE",
"-DSQLITE_ENABLE_STMTVTAB",
"-DSQLITE_ENABLE_DBPAGE_VTAB",
"-
DSQLITE_ENABLE_DBSTAT_VTAB", .

C_2 "sqlite3": [
…
"-O0",
"-DSQLITE_THREADSAFE=1",
"-DSQLITE_HAVE_ZLIB=1",
"-DHAVE_READLINE=0",
"-DHAVE_EDITLINE=0, …]

Fig. 3. Build Commands from The Same Target of C_1 and C_2. Blue Font Indicates Different Commands of Two Configurations

Incremental builds of software configurations. The plethora of possible software configurations poses great challenges for software development and maintenance. *SQLite* has up to 37 configuration options, which means that it can be built with 2^{37} configurations in theory. In practice, this number may not be reached as some options cannot coexist. The order of builds between configurations affects the overall efficiency of the software configuration set build. As shown in Fig. 1, in a configuration set with three configurations, the difference in terms of build time between the two build orders is 65.48s ($236.12s - 170.64s = 65.48s$) on our machine. Each software configuration contains the build targets to be built and the build commands for how to build those targets. When the build order is $C_1 \rightarrow C_2$ or $C_2 \rightarrow C_1$, the time consumption of building the latter configuration is only 0.04s.

As shown in Fig. 2, build targets are included in three configurations. C_1 and C_2 have the same targets. Therefore, when the build order is $C_1 \rightarrow C_2$ or $C_2 \rightarrow C_1$, the latter configuration can always be built quickly. The incremental build systems use a Makefile to record a set of obsolete files for rebuilding in the next build. Incremental builds are designed to rebuild only the necessary targets. The incremental build systems check whether build targets and their prerequisites are up to date based on their timestamps. The targets remain the same if they use the same compilation flags. The new configuration updates the rules in the Makefile, and the pre-processing also updates the timestamps of files (including targets and prerequisites), causing the incremental build systems to rebuild the targets. Thus, when two configurations share common targets (*e.g.*, C_1 and C_2), incremental builds can be used to quickly build the latter configuration.

Static analysis of configurations. BUDDI performs two static analyses of configurations: the first analyzes the distance between all pairs of configurations, and the second predicts the configuration build size (in terms of build time).

To analyze the distance, BUDDI obtains the build targets and the build commands of all configurations through a lightweight static analysis. BUDDI instructs GNU Make to output all build targets and build commands by executing the command (`make -n –debug=basic`). Fig. 1 shows an example of the build target and build commands obtained by BUDDI. BUDDI predicts the build size of the configuration by the number of build targets and build commands. It identifies the compilation optimization commands (*e.g.*, `gcc -O`, `gcc -O2`), which are given different weights (*e.g.*, `gcc -O0` for 1, `gcc -O` for 2, `gcc -O2` for 3, `gcc -O3` for 4, the default is 2). A higher weight

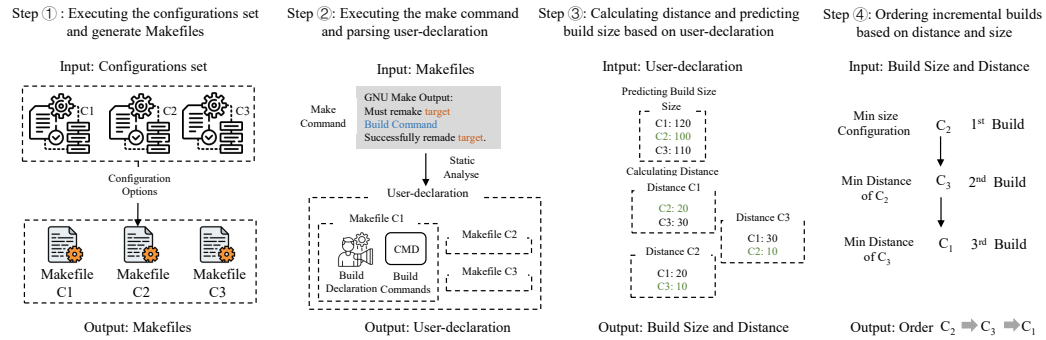

Fig. 4. Overview of BUDDI. The Dotted Line Represents the Set

indicates a higher cost, as higher levels of optimization result in longer compilation times (`gcc -O0` is the shortest) [19]. For example, C_3 has the most build targets (*cf.* Fig. 2), but its build time is comparable to that of C_1 (*cf.* Fig. 1). C_1 has some time-consuming build targets, such as "sqlite3.lo", which takes 26.31s to build.

Ordering incremental builds. BUDDI calculates the distance of the configuration from the build targets and the build commands. For example, C_1 and C_2 share identical build targets but different build commands for some of the targets. Next, BUDDI records the sum of the different commands as the distance between C_1 and C_2. C_1 and C_3 have different build targets and build commands. BUDDI records the sum of all the differences as the distance (*cf.* Fig. 2). Finally, BUDDI selects the configuration that is closest to the pre-order configuration for each iteration.

BUDDI starts from the configuration with the smallest build size when ordering configurations. As shown in Fig. 3, C_2 has fewer and faster build commands than C_1, and BUDDI chooses C_2 as the start of the build order. The intuition is that building from large size to small size may waste the work built in the previous order. For example, since the build size of C_3 is larger than C_2, the entire build time of the order $C_3 \rightarrow C_2$ is 170.60s but the build time of $C_2 \rightarrow C_3$ is 65.41s. Therefore, BUDDI selects C_2 as the start of the order, then C_1 as the next build based on distance, and finally C_3 to complete the order (*cf.* Fig. 1).

3 APPROACH

To accelerate the build of software configuration sets, we propose an approach called BUDDI with the goal of minimizing the number of redundant compilations of artifacts in the context of incremental builds. BUDDI is able to significantly reduce the overall build time of the software configuration set by leveraging the distance between the build scripts generated by the software configurations for ordering incremental builds efficiently. BUDDI analyzes the build targets and the build commands in build scripts, and then predicts the build size (specifically, in terms of build time) based on the number of build targets and build commands. The build orders by BUDDI always start from the smallest build-sized configuration, which minimizes redundant artifact compilations. BUDDI then calculates the distance in build targets and build commands between configurations and gives priority to building the configuration with the smallest distance. In general, the more differences between build targets and build commands in Makefiles, the greater the distance between them. In other words, a greater distance means that it would be harder to reuse compiled artifacts. This allows for a more efficient build by maximizing the reuse of compiled artifacts. BUDDI requires only static analysis of build scripts and achieves build acceleration by reusing artifacts wherever

possible. As long as the software configurations and build scripts are not changed, our analysis of ordering incremental builds only needs to be performed once.

3.1 Overview

An overview of BUDDI is provided in Fig. 4. It takes a software configuration set (including multiple software configurations) and first generates the corresponding Makefiles of each configuration (step ①). The Makefile specifies the building of the software configurations. Next, BUDDI instructs GNU MAKE to output the build targets and build commands declared by the user in the build scripts using the make command (step ②). Note that BUDDI eschews executing building during this step. BUDDI measures the distance between Makefiles by analyzing the build targets and the build commands to be executed when each Makefile is executed (step ③); BUDDI also predicts the build size by the build targets and build commands. BUDDI's order always starts with the smallest build size for the overall software configuration. The strategy aims to improve build efficiency by avoiding the difficulty of reusing compiled artifacts for smaller configurations after the larger ones have been built. The configurations closest to the current built configuration are then selected to create the build order (step ④). Our ordering of incremental builds has a low cost (based on lightweight static analysis) and can effectively reduce the overall build time of the configuration set. We propose a best-effort approach though it is not guaranteed to select the most efficient incremental build order.

3.2 Declaration Distance and Build Size

This section explains how BUDDI calculates the build distance and predicts the build size (steps ① to ③). Our core idea is to order incremental builds using build declarations distances between configurations. BUDDI determines the build distance and predicts the build size based on the target to be built and the specific build commands to be configured. This ensures that BUDDI can always build from the smallest configuration and build distance.

BUDDI considers similar declarations to be more suitable for incremental builds. Hence, the closer Makefiles are to each other, the more suitable they are for the incremental build. The build declaration describes the targets to be built by each Makefile and the details of their execution. After the configuration options have been applied, BUDDI can obtain the build declaration for each Makefile. It uses the GNU MAKE function (`make -n –debug=basic`) to simulate the build of all commands and printing of the build commands. BUDDI then parses the GNU MAKE output for the build targets and build commands (*cf.* an example output in Fig. 5). The first line of the figure indicates the current target to be built "sqlite3.lo". The second line indicates the build commands and prerequisites used to build "sqlite3.lo". The third line indicates that the target has been successfully built. BUDDI reads all output from GNU MAKE and finds all build targets and build commands by employing pattern matching. Although we focus on Make-based projects in this work, other build systems provide similar functionality to simulate builds. For example, Gradle provides an API [23] to achieve declared inputs/outputs of every task and declared dependencies of every task (Gradle programmers assemble build logic in a set of tasks [22]). Thus, BUDDI could be easily extended to these other build systems, such as Gradle, Ninja, and Bazel.

BUDDI uses Algorithm 1 to calculate the distance by comparing the build declaration in two Makefiles. First, BUDDI counts all the build targets and prerequisites in "MakefileB" (line 4). This simplifies the calculation of the number of targets and prerequisites in "MakefileB" that differ from those in "MakefileA". Then, if the prerequisites and build commands of targets in two Makefiles are the same, their distance is 0. If they are not the same at any point, their distance is the sum of the targets and prerequisites (lines 5–18).

Definition 1: Declaration Distance (DIS). DIS refers to the sum of differences between Makefiles.

1. Must remake target "sqlite3.lo".
2. ./libtool --mode=compile --tag=CC gcc -g -O2 -DSQLITE_OS_UNIX=1 -I. -I./sqlite/ext/async -I./sqlite/ext/session -I.//sqlite/ext/userauth -D_HAVE_SQLITE_CONFIG_H -DBUILD_sqlite -DNDEBUG -DSQLITE_THREADSAFE=1 -DSQLITE_ENABLE_MATH_FUNCTIONS -DSQLITE_HAVE_ZLIB=1 -DSQLITE_TEMP_STORE=1 -c sqlite3.c
3. Successfully remade target file "sqlite3.lo".

Fig. 5. Build Declaration and Command in *SQLite* Peoject

C1 --disable-opencl --disable-thread …		C2--enable-static …
"x264.o": [	"common/osdep.o": [	"common/osdep.o": [
1. …	1. …	1. …
2. "-fomit-frame-pointer",	2. "-fomit-frame-pointer",	2. "-fPIC",
3. "-fno-tree-vectorize",	3. "-fno-tree-vectorize",	3. "-fno-tree-vectorize",
4. "-fvisibility=hidden",	4. "-fvisibility=hidden",	4. "-fvisibility=hidden",
5. "-c",	5. "-c",	5. "-DX264_API_EXPORTS",
6. "x264.c",	6. "common/osdep.c",	6. "-c",
7. "-o",	7. "-o",	7. "common/osdep.c",
8. "x264.o"],	8. "common/osdep.o"],	8. "-o",
		9. "common/osdep.o"],

DIS = len(different target) + len (deps in different target) + len (different deps in same target) =1 + 8 + 3 = 12

Fig. 6. Build Declaration Distance between Two Makefiles

$$\text{DIS} = \text{TargetDIS} + \text{PresDIS} \tag{1}$$

DIS measures the distance between Makefiles and consists of TargetDIS and PresDIS. TargetDIS describes different build targets between two configurations. PresDIS describes the same build target (with the same target name) between two configurations but with different prerequisites. Note that the difference is only calculated with the name of the targets. The reason is that in a Makefile, the targets and prerequisites are a set of strings. We do not execute the build to recognize the differences between the targets from other perspectives (*e.g.*, binary).

Definition 2: TargetDIS. TargetDIS refers to the sum of different targets in two Makefiles.

Definition 3: PresDIS. PresDIS refers to the sum of differences in prerequisites (Pres) within the same targets.

$$\text{TargetDIS} = \sum_{n=1}^{\text{TargetSum}} \text{Pres} \tag{2}$$

where: TargetSum = Targets $\in$ {Declaration A-Declaration B}

$$\text{PresDIS} = \text{CUP- CAP} \tag{3}$$

where:

CUP = Target's Pres A $\cup$ Target's Pres B
CAP = Target's Pres B $\cap$ Target's Pres A
Target $\in$ (Declaration A $\cap$ Declaration B)

Fig. 6 illustrates how the *DIS* between the two Makefiles (C1 and C2 in the figure) is calculated. BUDDI compares the build targets in the Makefiles generated by the two software configurations. For two different build targets in the Makefile, such as "x264.o" in the figure, the distance for each target is added to 1, and the number of build commands for the build target is counted as distances. For example, all the build commands for "x264.o" in the figure have 8 lines, so the distance is increased by 8. For the same build target, the build commands are compared and the distance is increased by 1 for each different build command. For example, both Makefiles in the figure have the build target "common/osdep.o", but there is a difference of 3 build commands, so the distance is added by 3. Therefore, the *DIS* is 12 (1 + 8 + 3 = 12).

3.3 Ordering Builds

In this section, we elaborate how to use the build targets and build commands (obtained in Section 3.2) to predict the build size (Algorithm 2) and order the incremental builds (step ④) by size and distance (Algorithm 3).

First build configuration. BUDDI uses Algorithm 2 to select the first configuration to build (1^{st} build, start of the order) and Algorithm 3 to order the subsequent orders. At the beginning, the 1^{st} build in our approach will be a clean build. BUDDI considers that the 1^{st} build should be the smallest build size (in terms of Build time). Starting from the smallest configuration is intuitive, as it avoids the potential waste of the greatest amount of compiled artifacts. Starting with a larger build size may result in most of the compiled artifacts not being usable when building smaller configurations (*cf.* Scetion 4.5). Algorithm 2 outlines how BUDDI predicts the configuration with the smallest build size. BUDDI predicts the build size for each configuration by the number of build targets and the build commands. The more build targets and the more complex the build commands, the bigger the build size. BUDDI counts the number of build targets and prerequisites included in the build script for each configuration (lines 5–8). Next, BUDDI checks that the build commands (*e.g.*, `gcc -O`, `gcc -O2`) have specified common compilation optimization options (lines 9–15). BUDDI identifies the compilation optimization commands, which are assigned different weights (*e.g.*, `gcc -O0` for 1, `gcc -O` for 2, `gcc -O2` for 3, `gcc -O3` for 4, with 2 being the default).

Algorithm 1 Calculating Build Declaration Distance

```
1: INPUT: MakefilelistA, MakefilelistB, CommandA, CommandB
2: OUTPUT: DIS
3: count = 0, countB = 0
4: ListlenB = Sum(MakefilelistB.target) + Sum (Make-
   filelistB.target.deps)
5: for targets, deps in MakefilelistA.items() do
6:     if targets in MakefilelistB then
7:         if deps not in MakefilelistB[target] then
8:             count = count + Sum(deps)
9:         else if target in CommandB && CommandA[target] =
           CommandB[target] then
10:            countB = countB + Sum(deps)
11:        else
12:            count = count + len(target) + len(deps)
13:        end if
14:    else
15:        count = count + len(target) + len(deps)
16:    end if
17: end for
18: DIS = absolute(ListlenB - countB) + count
19: return DIS
```

Algorithm 2 Predicting 1^{st} Build of Configuration

```
1: INPUT: DIS[]
2: OUTPUT: FirstConfig
3: lenmakefile = 0, dismin = 0
4: for for Makefileitem in DIS: do
5:     for targets, deps in Makefileitem.items() do
6:         lenmakefile = lenmakefile+len(deps)+1
7:         countB = countB+len(deps)
8:         for dep in deps do
9:             if GCC compilation optimization not in dep then
10:                lenmakefile = lenmakefile - weights
11:            end if
12:        end for
13:        if dismin == 0 then
14:            dismin = lenmakefile
15:            firstonfig = Makefileitem
16:        else if dismin > lenmakefile then
17:            dismin = lenmakefile
18:            firstonfig = Makefileitem
19:        end if
20:    end for
21: end for
22: return firstonfig
```

Algorithm 3 Ordering Incremental Builds of Configurations

```
1: INPUT: firstconfig, DIS[]
2: OUTPUT: buildorder[]
3: buildorder = []
4: buildorder[] ← firstconfig
5: for dislist in Dis[] do
6:     nextconfig = Second Min DIS in dislist[]
7:     if firstconfig not in buildorder[] then
8:         buildorder[] ← nextconfig
9:     else
10:        mindis = Min DIS in dislist[]
11:        maxdis = Max DIS in dislist[]
12:    end if
13:    for config, dis in dislist.items() do
14:        if config in buildorder[] then
15:            continue
16:        end if
17:        if dis[mindis]) == mindis then
18:            nextconfig = SearchingConfig(mindis, buildorder[])
19:            break
20:        else if maxdis > dis[mindis] then
21:            maxdis = dis[mindis]
22:            nextconfig = config
23:        else if i == len (dislist) - 2 then
24:            if dis[mindis] == maxdis and config not in buil-
               dorder[] then
25:                nextconfig = config
26:            end if
27:        end if
28:    end for
29:    buildorder[] ← nextconfig
30: end for
31: Function SearchingConfig(mindis, buildorder[]):
32:    AllMindisConfig[]←Searching All config dis = mindis
33:    Sorting AllMindisConfig[] in options similarity with last item
       in buildorder[]
34:    return config = Max similarity config and not in buildorder[]
35: return buildorder[]
```

The compilation time is indicated by the weights, the higher the weight value the longer the compilation time. Finally, after a traversal, BUDDI obtains the software configuration with the minimum build size (lines 16–25). In Fig. 4, BUDDI predicts the build size of the three build scripts (C1, C2, C3), and obtains a build size set Size{C1: 120, C2: 100, C3: 110}. From this set, BUDDI can calculate that the smallest build size is C2, used as the 1^{st} build.

Ordering build. BUDDI uses the 1^{st} build of configuration and distance to order the builds. It is to find the closest configuration based on the current configuration (as shown in Algorithm 3). The builds are ordered by traversing the software configuration set once. Specifically, according to Algorithm 1, BUDDI calculates the distances between all configurations, the distances describe the difference in build scripts for each configuration. From Algorithm 2, we have obtained the 1^{st} build. Then, we obtain the 2^{nd} build by finding the software configuration that has the smallest distance to the 1^{st} build (lines 5–8 in Algorithm 3). In addition, the configuration with the smallest distance may already exist in the build order. Therefore, we need to find the component with the second smallest distance (lines 9–22). BUDDI considers the similarity of configuration options when the configurations are at the same distance from the pre-order configuration. Thus, from the alternative configurations (lines 17–19 and lines 31–34), the configuration whose options have the greatest overlap with those of the current configurations is selected. When ordering to the last position, BUDDI adds the configuration that is not yet in the order (lines 23–30).

Fig. 4 illustrates the main steps of generating the build order. The 2^{nd} build is identified by finding the minimum distance from the DIS of the first build. As the C3 in the figure, the 1^{st} build is C2. Then BUDDI selects the configuration with the smallest distance from C2, i.e. C3. After the 2^{nd} build is identified, the minimum distance does not exist in the order identified from the DIS of the 2^{nd} build. By iteratively repeating these steps, BUDDI obtains the build order. As shown in Fig. 4, having identified C3, BUDDI repeats the previous steps and finds that the configuration closest to C3 is C2. however, C2 already exists in the order. Then, BUDDI explores the remaining configurations and finds C1 is optimal for the 3^{rd} build.

4 EVALUATION

We implement our approach BUDDI as an automated tool called BuddiPlanner, and evaluate its effectiveness and efficiency on practical and widely-used configurable software systems.

4.1 Questions

In our evaluation, we sought to answer the subsequent questions (Qs). In addition, we explain the methodology that was used to answer these questions.

Q1: How effective is BuddiPlanner in creating build orders that can be efficiently compiled?

Goal—The main motivation of BUDDI is to reduce the build time of building multiple software configurations. Thus, we evaluated this aspect as an effectiveness metric.

Methodology—To answer Q1, we selected 20 mature and important projects. A challenge is how to select which configurations to build. Ideally, we would have built those configurations that developers or users build in practice. Thus, we tried to query practitioners about the real configurations and orders they use (*e.g.*, we asked in the SQLite forum). However, we found it difficult to obtain a clear response. Thus, we generated a configuration set containing 20 random configurations on each project. In each configuration set, we compared the whole build time between the order generated by BuddiPlanner and random orders. In addition, following the guidelines established by Arcuri and Briand [6], we conducted a Mann-Whitney U-Test to determine the statistical significance of the differences in terms of build time between BuddiPlanner's order and random orders (with $alpha = 0.05$).

Q2: How efficient is BuddiPlanner?

Goal—This question explores the time consumption of BuddiPlanner on creating an order for a configuration set.

Methodology—To answer Q2, we evaluated the time consumption of BuddiPlanner on generating build orders. BuddiPlanner consists of multiple stages, which we instrumented. These include static analysis, distance calculation, predicting 1^{st} build, and ordering build.

Q3: How effective is BuddiPlanner in creating an order from the largest build size?

Goal—This question explores whether BuddiPlanner's order of creation from other starting points is more efficient than an order created from the smallest build size as a starting point. Different starting points lead to different benefits. While intuitively, as explained above, we think it is more promising to start from the smallest build size, it may be more efficient to use the largest build size than the smallest build size, which is with reference to the first product selection from the maximum number of software features of Al-Hajjaji et al. [3, 4].

Methodology—To answer Q3, we made BuddiPlanner create an order starting from the configuration with the largest build size and compare it to the order starting from the smallest size build. We also experimented with various heuristic rules (*e.g.*, linking), but have not identified any others that perform well.

Q4: How do our experiment decisions (random configurations and random orders) affect the effectiveness of the evaluation?

Goal—This question explores whether the evaluation methodology in Q1 might make BuddiPlanner's performance seem higher than it is. The use of random configurations and random orders is one of the threats to the validity of our evaluation, because they might not be the practical configurations and orders that practitioners likely use. In addition, a single scenario (*e.g.*, experimental designs in Q1) may make BuddiPlanner perform better than it does.

Methodology—We compare BuddiPlanner's performance in different scenarios to determine if it is significantly lower than Q1's evaluation results. We designed this methodology with reference to the related work [36]. In Q1, our decisions use a set of 20 random configurations and 10 random orders. To evaluate Q4, we performed two experiments.

In the first experiment, we used multiple configuration sets and sequential build commands (*cf.* Section 4.6.1). For each project, we generated more sets of configurations (four sets). Furthermore, in order to diversify the evaluation scenarios, we randomly generated five configurations for each configuration set to distinguish them from the 20 random configurations used in the previous evaluations. At the same time, to avoid biasing the results by using random orders (used in previous evaluations), we used sequential order in this evaluation.

In the second experiment, we compared the build efficiency of the BuddiPlanner's order with that of all random orders (*cf.* Section 4.6.2). In previous evaluations, due to computational resource constraints, we were unable to compare BuddiPlanner's order to all orders (the number of all build orders is 20!). Therefore, we evaluate the set of configurations where all orders can be computed. To avoid biasing the results by using random configurations, we manually selected 4 configurations in the SQLite project. We compared the order generated by BuddiPlanner with all the orders (excluding the same order as BuddiPlanner, $4! - 1 = 23$) in terms of build time. For the four manually-selected configurations, we referenced the *SQLite* compilation documentation [54] and the Makefile in the Java branch of *SQLite* [56] to determine which configurations might likely be built by practitioners/developers: default configuration, debug enabled, all enabled, and all disabled. Practitioners do not disclose the specific configurations that *SQLite* employs, and as we have no access to that information, this is a best-effort approach.

Table 1. Important Information about Subjects

Project	Commit/Tag ID	Commits/Patch	Files	Opt.	Source	Stars
SQLite	3.41.0	25,439	2,232	37	[49]	4.5k
PHP	e9195b2	132,970	18,837	168	Self-extracted	36K
tig	06a1b89	2,789	338	11	Self-extracted	11.8k
tinycc	d1c1077	3,370	520	29	Self-extracted	1.6k
curl	33ac97e	29,674	6,430	186	[49]	31.2k
x264	32fc5fd	3,104	403	29	[49]	-
xterm	xterm-368	#368	318	114	[49]	-
xz	0b8fa31	1,669	1,106	51	[49]	-
ck	50299b7	1,665	595	17	Self-extracted	2.2k
lighttpd	140c6e3	5,000	348	59	Self-extracted	-
vim	9fcde94	18,011	4,335	71	Self-extracted	32.6k
thrift	0.18.1	6,815	3,059	52	Self-extracted	9.9k
xrdp	0.9.22.1	4,460	463	53	Self-extracted	4.8k
libusb	6bf2db6	1,808	155	25	Self-extracted	4.6k
tmux	4c60afd	10,081	315	17	Self-extracted	30.3k
libsodium	1.0.18	4,217	709	35	Self-extracted	11.1k
fswatch	ba411e0	1,366	166	25	Self-extracted	4.7k
goaccess	1.7.2	4,016	132	21	Self-extracted	16.6k
FFmpeg	4.1.11	111,845	7,029	337	Self-extracted	38.3k
ruby	3.2.2	78,732	11,328	70	Self-extracted	20.7k

Opt.: Configuration options #: Patches -: Not hosted on GitHub

4.2 Experimental Settings

Subjects. To answer the questions, we extensively evaluated our approach with 20 selected real-world projects. We reused all 5 projects used in a previous study on software configurations [49] plus other 15 well-known, active, and configurable projects to evaluate BuddiPlanner. We selected the 15 projects by following the two steps. First, we searched Github for C-based and Make-based projects and ordered them by star count. Next, we inspected the projects based on the order and manually removed the ones with less than 10 configuration options. We manually analyzed the total number of configuration options for each project by executing "`configure -h`". The projects selected come from a variety of fields, such as compilers, UNIX editors, interpreters, and multimedia processors.

The important information about these projects is displayed in Table 1. Each project has a large number of commits or patches (379–132,970) and offers a variety of configurations (11–337). For each project, we manually executed the pre-processing commands (*e.g.*, executing *autogen.sh*) to generate configurations. The configuration options available for each project were then extracted.

Baseline. We used a limited number of randomly derived configurations as a baseline. Practitioners' build configurations are not accessible in public. The most closely related work, which describes an empirical study on software configurations [49], has not proposed an approach that we could have used as a baseline to order the builds of configurations. Thus, it becomes reasonable to employ the randomly derived configurations as a best-effort baseline in this study. The projects have 11–337 configuration options, meaning that they have 2^{11}–2^{337} configurations theoretically. Constrained by computing resources, we randomly generated 20 configurations. Depending on the 20 configurations, up to 20! orders can be generated. Theoretically, the best order (minimum build time) could be found by running all the orders. However, we generated 20 build orders as a baseline (20→10), which is an arbitrary but reasonable limit. To generate random configurations, we first extracted the configurable options for each project. By executing `configure -h`, we obtained the configuration options for each project and saved all the options. We then generated 20 random configurations for each project and performed filtering on these configurations.

Filtering steps. To measure the build time, we first ensured that there were no conflicting configuration options in the experimental configuration set, as conflicting configuration options cause

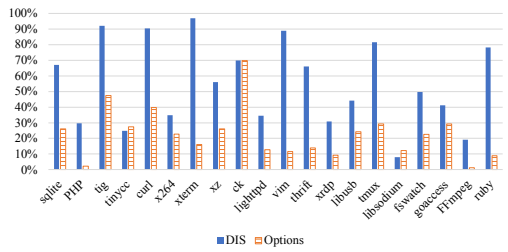

Fig. 7. Similarity of Options and Distance between Each Configuration

Listing 1 Build Approach

```
1  //Begin SQLite configuration set building
2  //Configuration C9
3  >>make clean
4  >>configure --enable-releasemode --with-readline-lib
5  >>time make
6  Building...
7  >>93.08866596221924
8  //Configuration C13
9  >>configure --enable-debug --with-readline-lib
10 >> time make
11 Building...
12 >>0.02551102638244629
```

build failures. For instance, in *x264*, the option "–system-libx264" can not be used together with "–enable-static". We checked the configuration options and eliminated conflicting options. For example, in the case above, we randomly removed one of the two options. To ensure the diversity of the generated configurations, we calculated the similarity of the configuration options and the distance between any two configurations. In terms of the similarity of configuration options, we refer to the percentage of the identical options in all the options in the configuration. As shown in Fig. 7, the average option similarity is 22% (the median is 21%). A low percentage means that each configuration has few options in common with others, which indicates the high diversity of the generated configurations. The distance shows the percentage of each configuration that has the same build target and build commands as the other configurations in each project. The average distance is 57% (the median is 55%), which illustrates the differences in the build targets and the build commands between configurations.

Build approach. To validate and evaluate BuddiPlanner, we followed a common series of steps for building the project. We first pre-configured the project manually (*e.g.*, executing "`autogen.sh`" to generate a "`configure`"). We then executed `configure -h` for each project separately to extract all the configuration options. In addition, to ensure that the build of each software configuration set is not affected by others, `make clean` is executed before the software configuration set is built to ensure that the current environment is clean (in line 3 of Listing 1). We then generate a build script by completing the configuration with "`configure`" depending on the order of execution (in lines 4 and 9) and finally execute `time make` (in lines 5 and 10). The builds for each software configuration set can be split into two builds: clean build and incremental build. We use clean builds to ensure that no other configuration builds are affected. In each configuration set, the first software configuration to be built actually performs a clean build. After that, each configuration is built on top of the build of the previous configuration in the order and run `make` directly (in lines 7 and 12). To eliminate experimental biases resulting from fluctuations in device performance, we arithmetically averaged the build time for each order by building it three times.

Experiment environment. We carried out the evaluation on a Linux server running Ubuntu 22.04 with an Intel(R) Xeon(R) Platinum 8163 CPU@2.50GHz, 96 cores, and 376GB memory.

4.3 Effectiveness (Q1)

To answer Q1, we followed the build approach in Listing 1 and automatically recorded all build time. The results of the effectiveness evaluation are shown in Table 2. The columns represent, from left to right, the name of the project, the item, and the random order count. The data items include *Time*, *Difference (Dif)*, and *Over* (×). *Time* is the total build time of the configuration set;

Table 2. Build time (Sec.) of Random Build Orders and BuddiPlanner. Green Blocks Indicate a Speedup, and Orange Blocks Indicate a Slowdown. The Bold Project Name Means That the Difference in Effectiveness Is Statistically Significant

Project	Item	Ours	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}
	Time	122.46	236.46	237.13	237.42	236.68	122.45	237.03	238.92	236.25	236.65	236.33
SQLite	Dif	-	114.00	114.67	114.96	114.22	-0.01	114.57	116.46	113.79	114.19	113.87
	Over (×)	-	48%	48%	48%	48%	0%	48%	49%	48%	48%	48%
	Time	540.26	869.26	841.69	987.03	1063.85	531.86	1064.95	1031.75	1063.87	959.66	1063.51
PHP	Dif	-	329.00	301.43	446.77	523.59	-8.40	524.69	491.49	523.61	419.40	523.25
	Over (×)	-	38%	36%	45%	49%	-2%	49%	48%	49%	44%	49%
	Time	69.36	136.47	136.42	149.63	149.75	109.87	136.51	136.25	109.44	136.43	126.30
tig	Dif	-	67.11	67.06	80.27	80.39	40.51	67.15	66.89	40.08	67.07	56.94
	Over (×)	-	49%	49%	54%	54%	37%	49%	49%	37%	49%	45%
	Time	309.78	449.85	460.02	448.22	344.36	436.55	469.73	461.01	462.35	470.24	448.75
tinycc	Dif	-	140.07	150.24	138.44	34.58	126.77	159.95	151.23	152.57	160.46	138.97
	Over (×)	-	31%	33%	31%	10%	29%	34%	33%	33%	34%	31%
	Time	1,004.32	1,156.41	1,025.05	980.17	1,053.40	1,195.54	1,052.59	1,054.10	1,053.10	1,138.61	998.31
curl	Dif	-	152.09	20.73	-24.15	49.08	191.22	48.27	49.78	48.78	134.29	-6.01
	Over (×)	-	13%	2%	-2%	5%	16%	5%	5%	5%	12%	-1%
	Time	2,452.79	2,800.52	2,936.82	2,461.91	2,474.24	2,538.17	2,881.52	3,633.40	2,595.50	2,513.43	2,506.73
x264	Dif	-	347.73	484.03	9.12	21.45	85.38	428.73	1180.61	142.71	60.64	53.94
	Over (×)	-	12%	16%	0%	1%	3%	15%	32%	5%	2%	2%
	Time	360.59	408.62	408.62	408.63	384.75	408.42	432.51	408.33	432.13	426.17	408.55
xterm	Dif	-	43.33	43.33	43.34	19.46	43.13	67.22	43.04	66.84	60.88	43.26
	Over (×)	-	11%	11%	11%	5%	11%	16%	11%	15%	14%	11%
	Time	289.17	451.19	436.40	410.42	445.17	409.17	484.61	489.64	524.97	329.63	447.13
xz	Dif	-	255.31	240.52	214.54	249.29	213.29	288.73	293.76	329.09	133.75	251.25
	Over (×)	-	57%	55%	52%	56%	52%	60%	60%	63%	41%	56%
	Time	20.53	21.25	23.18	23.29	21.20	23.09	21.27	21.05	21.06	23.04	21.10
ck	Dif	-	0.72	2.65	2.76	0.67	2.56	0.74	0.52	0.53	2.51	0.57
	Over (×)	-	3%	11%	12%	3%	11%	3%	2%	3%	11%	3%
	Time	510.02	507.09	505.80	510.49	559.33	556.96	559.16	560.44	560.77	558.41	509.55
lighttpd	Dif	-	-2.93	-4.22	0.47	49.31	46.94	49.14	50.42	50.75	48.39	-0.47
	Over (×)	-	-1%	-1%	0%	9%	8%	9%	9%	9%	9%	0%
	Time	956.53	3,103.11	2,975.67	2,604.68	2,791.81	2,979.82	2,978.79	3,168.03	2,981.30	2,793.22	2,791.51
vim	Dif		2146.58	2019.14	1648.15	1835.28	2023.29	2022.26	2211.5	2024.77	1836.69	1834.98
	Over (×)		69%	68%	63%	66%	68%	68%	70%	68%	66%	66%
	Time	869.00	882.75	877.26	877.18	877.57	877.40	877.30	877.07	877.47	877.67	876.99
thrift	Dif	-	13.75	8.26	8.18	8.57	8.40	8.30	8.07	8.47	8.67	7.99
	Over (×)	-	2%	1%	1%	1%	1%	1%	1%	1%	1%	1%
	Time	133.47	151.55	144.97	135.27	166.78	151.39	148.82	178.18	167.09	166.62	149.52
xrdp	Dif		18.08	11.5	1.8	33.31	17.92	15.35	44.71	33.62	33.15	16.05
	Over (×)		12%	8%	1%	20%	12%	10%	25%	20%	20%	11%
	Time	109.50	125.63	120.35	127.53	128.18	111.98	111.66	127.36	124.81	119.43	134.81
libusb	Dif	-	16.13	10.85	18.03	18.68	2.48	2.16	17.86	15.31	9.93	25.31
	Over (×)	-	13%	9%	14%	15%	2%	2%	14%	12%	8%	19%
	Time	40.58	130.48	118.84	188.23	105.44	50.74	48.47	45.49	49.10	49.69	76.72
tmux	Dif	-	89.90	78.26	147.65	64.86	10.16	7.89	4.91	8.52	9.11	36.14
	Over (×)	-	69%	66%	78%	62%	20%	16%	11%	17%	18%	47%
	Time	95.60	143.16	168.74	154.96	152.16	160.85	161.98	173.48	127.23	131.42	147.86
libsodium	Dif	-	47.56	73.14	59.36	56.56	65.25	66.38	77.88	31.63	35.82	52.26
	Over (×)	-	33%	43%	38%	37%	41%	41%	45%	25%	27%	35%
	Time	175.21	208.77	175.64	279.14	232.15	249.38	233.20	213.32	257.64	252.51	324.44
fswatch	Dif	-	33.56	0.43	103.93	56.94	74.17	57.99	38.11	82.43	77.30	149.23
	Over (×)	-	16%	0%	37%	25%	30%	25%	18%	32%	31%	46%
	Time	83.32	117.76	166.83	150.78	138.68	137.79	119.33	151.21	125.81	105.03	94.79
goaccess	Dif		34.44	83.51	67.46	55.36	54.47	36.01	67.89	42.49	21.71	11.47
	Over (×)		29%	50%	45%	40%	40%	30%	45%	34%	21%	12%
	Time	9,966.47	10,842.39	10,703.52	10,921.17	10,956.59	10,829.22	10,872.89	10,855.23	10,719.75	10,842.70	10,845.78
FFmpeg	Dif		875.93	737.05	954.70	990.12	862.75	906.42	888.77	753.29	876.23	879.31
	Over (×)		8%	7%	9%	9%	8%	8%	8%	7%	8%	8%
	Time	2,622.36	5,257.42	4,109.11	4,456.04	2,905.56	3,983.89	4,690.04	4,958.50	4,146.59	4,909.35	3,228.10
ruby	Dif	-	2,635.06	1,486.75	1,833.68	283.20	1,361.53	2,067.68	2,336.14	1,524.23	2,286.99	605.74
	Over (×)	-	50%	36%	41%	10%	34%	44%	47%	37%	47%	19%

O: Order. Dif: Difference. U-T: Mann-Whitney U-Test value

Dif is the difference between the build time of BuddiPlanner's order and random orders; *Over* (×) is the ratio of *Dif* to *Time* (*Dif* / *Time*), indicating efficiency. Specifically, in project *SQLite* the total build time for BuddiPlanner is 122.46s and the build time for the randomly generated build order

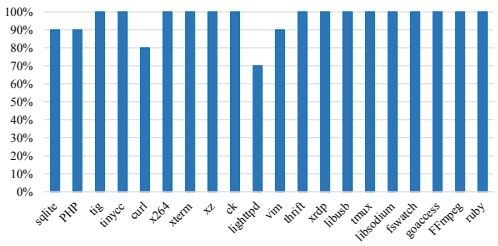

Fig. 8. The Percentage of BuddiPlanner's Order Faster Than Random Orders in Per Project

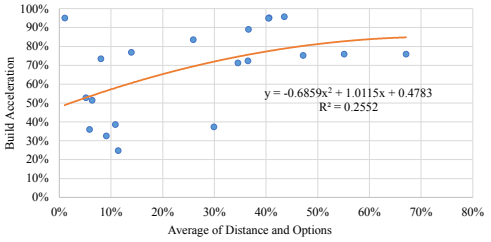

Fig. 9. Regression Results for Average of Distance and Options and Build Acceleration (BuddiPlanner / Clean Build)

(O_1) is 237.75s. The *Dif* is 114.00s (236.46*s* − 122.46*s*). *Over* (×) is 48% (114.00*s*/236.46*s*). *Time* of BuddiPlanner contains the time on ordering.

Results. We observe that the order created by BuddiPlanner is faster than 96.5% (193/200) of the random orders in terms of build time. The average build time is reduced by 8.87s (1%)–1960.26s (67%). The BuddiPlanner's order is able to save on average 305.94s (26%) of the random orders, with a median saving of 64.88s (28%). In Fig. 8, we count the percentage of each project in which BuddiPlanner's order is more efficient than random orders in terms of build time. Specifically, BuddiPlanner creates a build order that outperforms all random orders in up to 16 projects. BuddiPlanner creates build orders that effectively reduce the build time. For example, in large-scale projects, such as *vim* and *ruby*, BuddiPlanner saves up to 1960.26s (67%) and 1642.1s (36%) on average over the random orders. In small-scale projects, e.g., *SQLite* and *tig*, BuddiPlanner creates an order with an average reduction of 103.07s (43%) and 63.35s (47%) over the random orders.

Interpretation. The order created by BuddiPlanner reuses the artifacts built in the pre-order to a high extent, which makes the build more efficient than random orders. BuddiPlanner takes into account the distance between configurations and the similarity of configuration options. BuddiPlanner prioritizes the configuration closest to the pre-order build, and if there is an equal distance, it selects the configuration with the most similar configuration options, ensuring that it builds the configuration closest to the current configuration every time. As shown in Fig. 9, the distance and the options of the software configuration also show a positive correlation trend i.e. build acceleration, with the closer the distance and options the better the acceleration. For *thrift*, the order created by BuddiPlanner is only 7.99s–13.75s faster than random orders. This is because incremental builds can effectively accelerate builds between configurations. When configurations are close in terms of distance, using even a random order is effective in speeding up the build, which limits the use of BuddiPlanner.

> ***Answer to Q1:*** *The order created by BuddiPlanner outperforms 193 out of 200 random orders in terms of build time with a reduction in the build time of 8.87s (1%)–1960.26s (67%), which demonstrates the effectiveness of BuddiPlanner.*

4.4 Efficiency (Q2)

To evaluate the efficiency of BuddiPlanner, we further measured the additional overheads associated with using BuddiPlanner (Q2). The evaluation results are shown in Table 3. Each column in the table respectively indicates the time consumption of statically analyzing the build targets and build commands, calculating the distance between configurations, predicting the build size to calculate 1^{st} build, ordering incremental builds, and the sum of the above.

Table 3. Time-consumption (Sec.) of Ordering Build for Each Project

Projects	Command[1]	Static Analysis	DIS Calculation	1^{st} Build	Ordering	Sum	Sum Build[2]	%[3]
SQLite	0.81	0.18	0.29	0.03	0.01	1.32	122.46	1.1%
PHP	0.08	0.78	0.69	0.08	0.01	1.65	540.26	0.3%
tig	1.81	0.53	0.46	0.05	0.01	2.86	69.36	4.1%
tinycc	2.08	0.31	0.26	0.02	0.01	2.69	309.78	0.9%
curl	17.78	2.67	3.51	0.45	0.01	24.42	856.94	2.8%
x264	54.73	0.01	0.40	0.01	0.01	55.16	2,452.79	2.2%
xterm	0.22	0.01	0.18	0.01	0.01	0.44	360.59	0.1%
xz	2.55	1.37	1.52	0.21	0.02	5.65	289.17	2.0%
ck	0.43	0.19	0.24	0.02	0.03	0.92	20.53	4.5%
lighttpd	6.35	1.70	1.53	0.23	0.01	9.82	510.02	1.9%
vim	16.45	1.49	3.50	0.20	0.06	21.70	956.53	2.3%
thrift	4.97	0.57	0.51	0.06	0.05	6.16	869.00	0.7%
xrdp	2.79	0.40	0.29	0.04	0.01	3.53	133.47	2.6%
libusb	1.59	0.28	0.22	0.02	0.01	2.12	109.50	1.9%
tmux	2.30	0.02	0.10	0.00	0.06	2.49	40.58	6.1%
libsodium	5.44	2.38	3.00	0.43	0.01	11.27	95.60	11.8%
fswatch	3.58	0.59	0.53	0.04	0.02	4.76	175.21	2.7%
goaccess	2.34	0.57	0.43	0.05	0.01	3.41	83.32	4.1%
FFmpeg	20.63	17.58	32.23	4.43	0.01	74.88	9,966.47	0.8%
ruby	307.83	6.60	9.06	1.21	0.01	324.71	2,622.36	12.4%

[1]: Time of execution command `make -n –debug=basic` [2]:Sum build time consumption of BuddiPlanner's order [3]: Taking up build time

Results. From Table 3, we observe that BuddiPlanner orders incremental builds quickly (0.44s–324.72s) on all the projects. It takes a short time in static analysis (0.01s–17.28s), distance calculation (0.01s–32.23s), 1^{st} build (0.01s–4.43s) and ordering incremental builds (0.01s–0.06s). The most time-consuming stage of BuddiPlanner is the simulation executing the build (`make -n –debug=basic`, 0.08s for *PHP*, 307.83s for *ruby*). It requires this stage to determine which targets need to be compiled under the current configuration. However, considering that the build order only needs to be executed once if the software configuration does not change. For example, the changes do not involve pre-processor directives and Makefile, which indicates that the time consumption of ordering is acceptable.

Answer to Q2: *The extra overheads required by BuddiPlanner lie in 0.44s–324.72s over the 20 projects, which demonstrates the efficiency of BuddiPlanner.*

4.5 Efficiency of the Order From the Largest Build Size (Q3)

In this section, we evaluate the build efficiency of creating a build order from the largest build size, for which we expect a lower efficiency as compared to the smallest build size. We compared the build times of build orders created by BuddiPlanner from the largest build size and the smallest build size. Each build is executed three times.

Results. Fig. 10 shows the results of our evaluation. On all projects, BuddiPlanner creates a build order in which the 1^{st} build with the smallest build size is preferable to a build order in which the 1^{st} build with the largest build size. The smallest build size as 1^{st} build of order was reduced by an average of 160.23s (16%) compared to the order in which the 1^{st} build is with the largest build size, with a median saving of 33.99s (12%).

Interpretation. A build order whose the 1^{st} build is with the smallest build size allows for higher reuse of the artifacts of the preceding configurations. This is because the build order starting from the smallest size largely reduces the conflict of reusing artifacts. Assuming that the incremental build systems would build artifacts in a sequence from the largest to the smallest, the first build should take a significant amount of time, but subsequent smaller builds should be accelerated. As a result of our evaluation, we concluded that it is more beneficial to build from the smallest build

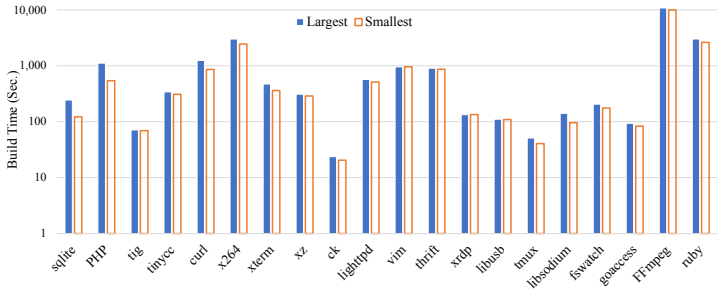

Fig. 10. Comparison of Build Efficiency with Largest-first and Smallest-first Ordering Strategies

rather than the largest build. The main differences between software configurations can be found in the difference between build targets and build commands. In different configurations, the same build target would produce different artifacts as a result of different build commands. Starting with the largest-sized build may lead to some artifacts not being effectively reused. When generating large-scale builds, intermediate artifacts are generated and cleaned up afterward to create complex artifacts. However, in small-scale build size configurations, these same artifacts are not cleaned up and can be reused in the production of large-scale builds.

> ***Answer to Q3:*** *BuddiPlanner creates a build order starting with the smallest-sized build is reduced by an average of 160.23s (16%) in terms of build time compared to the build order starting with the largest-sized build, with a median saving 33.99s (12%), which demonstrates the smallest-sized build as the* 1^{st} *build is more efficient than the largest-sized build.*

4.6 Effects of Experiment Decisions (Q4)

This section aims to evaluate the effects on the effectiveness of 20 random configurations and 10 random orders on BuddiPlanner's evaluation.

4.6.1 Multiple Configuration Sets and Sequential Build Order. In this section, we evaluated the effects on the set containing 20 random configurations (*cf.* Section 4.3) relative to the four sets of configurations and sequential build order of BuddiPlanner.

Results. Fig. 11 shows the results of our evaluation. BuddiPlanner's order outperforms the sequential order in 75% (60/80) of the random configurations in terms of build time. The utilization of multiple configuration sets containing random configurations reduces the efficiency of BuddiPlanner when compared with the configuration set containing 20 random configurations used in Q1–Q3. A randomly generated configuration set with 20 random configurations yields a positive effect (increasing the percentage better than the baseline) on the evaluation of BuddiPlanner. Random order can also be effective in speeding up the build process when configurations are close to each other. The range of time in which the efficiency of building sequential orders surpasses that of BuddiPlanner's order stands between 0.001s (0.01%)–8.55s (6%), with a limited difference between the two orders. However, we observe that BuddiPlanner's order can significantly reduce the build times for large-scale projects such as *PHP*, *X264*, *vim*, *FFmpeg*, and *ruby*. Overall, the evaluation of BuddiPlanner is positively affected by the decision to generate a configuration set containing 20 random configurations for each project individually. However, this positive effect is limited, especially in the case of large-scale projects.

4.6.2 Manually Selected Configurations and All Build Orders. In this section, we evaluated the effects on 10 random orders (*cf.* Section 4.3) relative to manually selected four configurations and all build orders of BuddiPlanner.

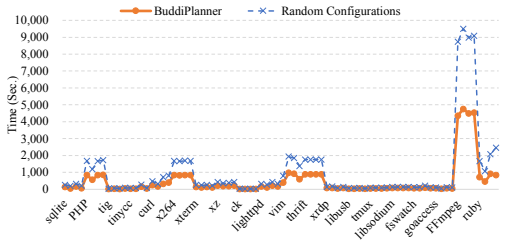

Fig. 11. Time Consumption of BuddiPlanner's Order and Sequential Order in Random Configurations

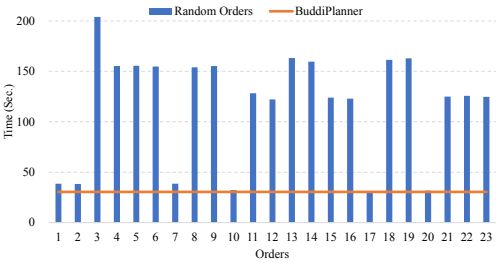

Fig. 12. Time Consumption of BuddiPlanner's Order and Random Orders in *SQLite* Configurations

Results. Fig. 12 shows the evaluation results. We conducted a comparison between BuddiPlanner's order and all possible orders (excluding the same order as BuddiPlanner), with being executed three times for each order build. The build time for BuddiPlanner's order is 30.51s. For random orders, the range of build times is between 31.24s to 203.98s. BuddiPlanner produces a build order whose build time is comparable to the optimal order. BuddiPlanner's high build efficiency is achieved by maximizing the reuse of pre-order builds through the calculation of configuration distances. In the configurations currently utilized in this evaluation, BuddiPlanner demonstrates the effectiveness comparable to the optimal order. Overall, the decision to randomize 10 orders in configurations does not have a positive effect on BuddiPlanner's evaluation.

> ***Answer to Q4:*** *The experimental design of generating a configuration set containing 20 random configurations for each project has a limited positive effect (increasing the percentage better than the baseline) on BuddiPlanner's evaluation. The application of random orders does not have a positive effect on the evaluation of BuddiPlanner. This demonstrates that the design decisions we made in the previous evolutions (Q1) did not exaggerate BuddiPlanner's performance.*

5 DISCUSSION

This section further discusses the issues relevant to BUDDI, BuddiPlanner, and this study.

Definition of distance. Our definition of distance is similar to the core idea presented in related work for similarity-driven prioritization for product-line testing [4]. The distance metric presented in the study is calculated from features that refer to specific features of a product, such as a call or GPS in a mobile phone, via the Hamming distance [13, 27]. Different features can make up different products. These features are similar to build configuration options; that is, different configuration options make up different configurations, which can build different artifacts. However, in the context of building, we consider that the building of software is essentially established by build commands and build targets. We consider it ineffective—based on our observations for the projects we selected—to compute distances by using features, which, in our case, would correspond to configuration options. Consider the three configurations—C_1 (disable-libtool-lock, with-pic), C_2 (enable-debug, with-readline-lib), C_3 (disable-amalgamation, with-readline-lib)—in Section 2. Each configuration has two configuration options. If calculated by options, C_2 and C_3 are the ones with the smallest distance, since they both have a common configuration option (with-readline-lib). However, irrespective of whether the build order is $C_1 \rightarrow C_2$ or $C_2 \rightarrow C_1$, the latter can always be built quickly by the incremental build system (on our machine the time consumption is 0.04s for both). When the build order is $C_2 \rightarrow C_3$ or $C_3 \rightarrow C_2$, the latter still takes much time to build (*cf.* Fig 1). This demonstrates that merely calculating distances by options is not effective. Our definition emphasizes the relationship and specificity between build targets and build commands. From Fig 2,

it appears that C_1 and C_2 are the same build targets. With our proposed distance metrics, for the given example, the distance between C_1 and C_2 can be calculated to be the minimum.

Actionable insights for practitioners. The benefits of strategically ordering incremental builds can be clearly demonstrated in our experiments. Prioritizing incremental builds helps reduce the time and resource pressures associated with software builds. Software configuration sets can be built faster by carefully ordering incremental builds. Given the limited time, practitioners are able to build more configurations, or build the same configurations with fewer computing resources. Practitioners may benefit from it every time they build. The resulting efficacy is not only a reduction in the amount of time and resources needed to build software, but also, or even more importantly, the ability to receive quicker feedback.

Heuristic rules. We consider that BuddiPlanner may be improved by heuristic rules. We analyzed some cases (*e.g.*, orders in Q1 that are faster than BuddiPlanner in terms of build time) and tracked build commands that can affect the build time (*e.g.*, compilation optimization and linking). We observed that compilation optimization commands can significantly affect the build time (*cf.* Section 3.3). We have not yet found other generalized heuristic rules.

Build failures and different artifacts. Changing the build order can result in different build artifacts, but these indicate errors in the build scripts. BuddiPlanner does not cause incorrect builds or different artifacts if the incremental build systems and build scripts are error-free. It has been pointed out in the previous study [49] that there are instances when different artifacts are built due to the inability of the incremental build system to recognize the build target correctly (*i.e.*, build dependency errors in Makefiles cause incorrect incremental builds [37, 53]). Many methods exist to detect build dependency errors in Makefiles [8, 16, 37, 53].

Random orders are better. In our experiments, we have demonstrated that build orders can be used to accelerate incremental builds for configuration sets. We observed that BuddiPlanner performs less efficiently when it comes to a few random orders, but outperforms 96.5% of the random orders. Determining why a random order was better in some cases is inherently difficult, as performance might depend on various software components such as compilers, linkers, or packagers for libraries. Based on the evaluation, the build time for the random orders is only 0.47s (0%) to 24.15s (2%) faster than BuddiPlanner, which is slim compared to the most cases when BuddiPlanner outperforms the random orders.

Generality. BuddiPlanner is mainly based on the Make-base build system in this stage, specifically GNU Make. BuddiPlanner could easily extend to other build systems (*e.g.*, Gradle [21], Ninja [46], and Bazel [7]), and its further adaptation and validation will be included in the future work. For instance, for Gradle projects, the methods for obtaining build targets and build commands need to be adapted.

Threats to validity. In the event of fluctuations in the performance of the experimental physical equipment, our evaluation may be invalidated. To minimize randomness, we halted all other user processes on the system and ran the experiments on each project three times. The average elapsed time for each phase was determined as the result. We calculated the standard deviation for the time of BuddiPlanner's order and each random order. The results show that running evaluations three times can effectively mitigate the impact on the validity of experimental results due to fluctuations in machine performance. The standard deviation ranges from 0.2% (0.046s in *CK*) to 0.5% (56.225s in *FFmpeg*) of order build time. Accordingly, the elapsed time for each project demonstrates BuddiPlanner's efficiency.

The effectiveness and efficiency of our approach are evaluated using 20 projects. A threat might be how the results could be generalized to other projects that we have not taken into account. As

an effort to mitigate this risk, we chose a large set of projects, including the five projects used in a previous evaluation [49] as well as 15 additional projects. These projects have been thoroughly evaluated and are widely used, indicating that they could be a representative set of well-known projects that offer multiple configurations.

The use of random configurations and random orders might be a threat to validity. Firstly, random configurations may not represent the practical configurations that practitioners would use. Although we have tried to consult practitioners in the open-source community (*i.e.*, SQLite forum [55]) for the configuration options and orders they ever used, unfortunately, we did not receive a useful response. In addition, constrained by computational resource limits, our evaluation cannot cover all possible configurable combinations and test all orders. To increase the diversity of the samples and mitigate the bias of evaluation results, we use a random generation strategy of configurations and orders. Note that BUDDI does not guarantee to create the order of incremental builds with the shortest build time. Instead, it is able to generate an order with a build time shorter than most cases in an efficient manner.

6 RELATED WORK

We introduce the most relevant research related to ordering incremental builds for a software configuration set.

Incremental build systems. Many works research on incremental build systems [18, 25, 26]. Erdweg et al. [15] proposed a build system called *pluto*, which supports fine-grained file dependencies. *Pluto* generalizes the traditional concept of timestamps and allows builders to declare their actual requirements for the content of a file. Pluto collects the builder's requirements and products and their markup in a build summary, which allows *pluto* to provide reliable and optimal incremental builds. Konat et al. [35] proposed a dynamic dependency graph-based approach that used changed files to drive the re-build of tasks, loading and executing only those tasks affected by the changes to achieve more accurate incremental builds. Curtsinger et al. [11] proposed build tools (*LaForge* and *Riker*), which correctly build without specifying dependencies or incremental build steps. Cserép et al. [10] detected only the files that needed to be built by parsing the overall code base. The file to be rebuilt was one that had been modified or whose generation rules had been changed, so they did not need to detect the timestamp of the file as Make [20] does. Randrianaina et al. [49] demonstrated that incremental build systems can be applied to the build of multiple software configurations. Furthermore, they demonstrated that a specific build order can accelerate the building of multiple software configuration sets. To the best of our knowledge, there are no studies on ordering incremental builds for software configurations.

Since Make [20] performs an incremental build strategy in configurations, the correctness of incremental builds of real projects based on the Make build system cannot be guaranteed (build dependency errors do not correctly trigger incremental builds), and many studies have attempted to detect build dependency errors to improve this problem.

Detecting build dependency errors. Common build dependency errors are *missing dependencies* and *redundant dependencies*. The former may erroneously prevent the build targets from rebuilding in incremental builds, while the latter can reduce the execution efficiency of incremental builds and parallel builds. Xia et al. [60] and Zhao et al. [61] used MAKAO [1] to parse build scripts and then analyzed the source code to predict missing dependencies. Licker et al. [37] proposed a method to detect build dependency errors by triggering a large number of incremental builds. Their method triggers incremental builds by modifying the test file (timestamp) to observe if the desired file was rebuilt. Bezemer et al. [8] and Fan et al. [16] leveraged parsing build scripts and monitoring actual builds to detect build dependency errors.

Empirical studies on build systems. Software builds are at the heart of all projects, and several empirical studies on build systems have been performed. Seo et al. [52] examined 26.6 million releases and found that dependency errors were the most common type of error in C++ (52.68%) and Java (64.71%) projects. The study shows that build maintenance accounts for 27% of the development overhead of source code and 44% of the test development overhead. Frequent maintenance of builds may affect practitioner efficiency. McIntosh et al. [44] pointed out that up to 79% of source code developers and 89% of test code developers are severely impacted by build maintenance.

Software product line (SPL). Software configuration is one of the primary measures for building software variants. The SPL community has proposed a number of ways to manage a range of variants [14, 24, 29, 39]. The primary principle of these techniques is to exploit the similarities between variants. Similarities between variants are often used in SPL testing to reduce the testing workload [12, 13, 32, 33, 40, 47]. Al-Hajjaji et al. [2, 3] proposed a similarity-based prioritization method where they select the most functionally diverse product for the next test incrementally. This method increases effectiveness in terms of fault detection ratio. They also improved the effectiveness of SPL testing through delta-oriented prioritization [4], which orders products based on their similarity in delta modelings (a method of automating product derivation for SPL, it is to apply an operation on the specified core model, such as adding an element [9]). Henard et al. [28] sampled and prioritized products based on similarities between them. On this basis, Saini et al. [50] proposed a method for calculating inter-product distances based on the original and desired features of products in the same product line. Khoshmanesh and Lutz [31] proposed a method for detecting problematic interactions involving new features using feature-level similarity metrics that can effectively detect unplanned interactions of new features with existing features. In addition, *static analysis* and *type checking* are used to find errors in configurable software and can be extended to code bases, such as the Linux kernel [51, 57, 59].

BuddiPlanner similarly orders incremental builds based on the distance between software configurations. However, BuddiPlanner takes account of the differences between the build targets and commands in the build script, in contrast to the previous approach. Differences in build scripts are more elaborate and complex than differences between features and delta modelings.

7 CONCLUSION

In this paper, we propose an approach, BUDDI, to order incremental builds to accelerate the build of the entire set of software configurations. The evaluation shows that BUDDI outperforms the random orders by 96.5% across the 20 projects evaluated, reducing build times by 8.87s (1%)—1960.26s (67%). The order generated by BuddiPlanner is able to save an average of 305.94s (26%) over the random orders, with a median saving of 64.88s (28%). Our approach is able to prioritize incremental builds of a software configuration set in an efficient manner, merely requiring 0.92s–324.72s across the tested projects. It is expected that BuddiPlanner helps relieve the burden of allocating significant build time and computational resources required by practitioners when building with multiple software configurations.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.62202219, No.62072227, No.62302210), the Jiangsu Provincial Key Research and Development Program (No.BE2021002-2), and the Innovation Project and Overseas Open Project of State Key Laboratory for Novel Software Technology (Nanjing University) (ZZKT2022A25, KFKT2022A09, KFKT2023A09, KFKT2023A10).

REFERENCES

[1] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. IEEE Computer Society, 114–123. https://doi.org/10.1109/ICSM.2007.4362624

[2] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Softw. Syst. Model.* 18, 1 (2019), 499–521. https://doi.org/10.1007/s10270-016-0569-2

[3] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. 2014. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, Stefania Gnesi, Alessandro Fantechi, Patrick Heymans, Julia Rubin, Krzysztof Czarnecki, and Deepak Dhungana (Eds.). ACM, 197–206. https://doi.org/10.1145/2648511.2648532

[4] Mustafa Zaid Saleh Al-Hajjaji. 2017. Similarity-driven prioritization and sampling for product-line testing. http://dx.doi.org/10.25673/5169

[5] Ant. Online; 2022. *Apache Ant manual.* Retrieved 2022 from https://ant.apache.org/manual/

[6] Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 1–10. https://doi.org/10.1145/1985793.1985795

[7] Bazel 2023. *Bazel: a fast, scalable, multi-language and extensible build system.* Retrieved 2023 from https://bazel.google.cn

[8] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. Germán, and Ahmed E. Hassan. 2017. An empirical study of unspecified dependencies in make-based build systems. *Empir. Softw. Eng.* 22, 6 (2017), 3117–3148.

[9] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. 2015. Abstract delta modelling. *Math. Struct. Comput. Sci.* 25, 3 (2015), 482–527. https://doi.org/10.1017/S0960129512000941

[10] Máté Cserép and Anett Fekete. 2020. Integration of Incremental Build Systems Into Software Comprehension Tools. In *Proceedings of the 11th International Conference on Applied Informatics Eger, Hungary, January 29-31, 2020 (CEUR Workshop Proceedings, Vol. 2650)*, Gergely Kovásznai, István Fazekas, and Tibor Tómács (Eds.). CEUR-WS.org, 85–93. http://ceur-ws.org/Vol-2650/paper10.pdf

[11] Charlie Curtsinger and Daniel W. Barowy. 2021. LaForge: Always-Correct and Fast Incremental Builds from Simple Specifications. *CoRR* abs/2108.12469 (2021). arXiv:2108.12469 https://arxiv.org/abs/2108.12469

[12] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative programming - methods, tools and applications.* Addison-Wesley. http://www.addison-wesley.de/main/main.asp?page=englisch/bookdetails&productid=99258

[13] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Search-based Similarity-driven Behavioural SPL Testing. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, Salvador, Brazil, January 27 - 29, 2016*, Ina Schaefer, Vander Alves, and Eduardo Santana de Almeida (Eds.). ACM, 89–96. https://doi.org/10.1145/2866614.2866627

[14] Oscar Díaz, Leticia Montalvillo, Raul Medeiros, Maider Azanza, and Thomas Fogdal. 2022. Visualizing the customization endeavor in product-based-evolving software product lines: a case of action design research. *Empir. Softw. Eng.* 27, 3 (2022), 75. https://doi.org/10.1007/s10664-021-10101-6

[15] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of the 2015 International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 89–106. https://doi.org/10.1145/2814270.2814316

[16] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 463–474. https://doi.org/10.1145/3395363.3397388

[17] Stuart I. Feldman. 1979. Make-A Program for Maintaining Computer Programs. *Softw. Pract. Exp.* 9, 4 (1979), 255–65. https://doi.org/10.1002/spe.4380090402

[18] Ray Ford and Mary Pfreundschuh Wagner. 1990. Incremental concurrent builds for modular systems' . *J. Syst. Softw.* 13, 3 (1990), 157–176. https://doi.org/10.1016/0164-1212(90)90092-Z

[19] GCC 2023. *Using the GNU Compiler Collection.* Retrieved 2023 from https://gcc.gnu.org/onlinedocs/gcc-10.4.0/gcc/#toc-GCC-Command-Options

[20] GNU make 2020. *GNU make manual.* Retrieved 2022 from https://www.gnu.org/software/make/manual/make.html

[21] Gradle 2023. *Gradle Build Tool.* Retrieved 2023 from https://gradle.org/

[22] Gradle 2023. *Gradle User Manual.* Retrieved 2023 from https://docs.gradle.org/current/userguide/userguide.html

[23] Gradle Plugins 2023. *Developing Custom Gradle Plugins.* Retrieved 2023 from https://docs.gradle.org/current/userguide/custom_plugins.html

[24] Nahid Hajizadeh, Peyman Jahanbazi, and Reza Akbari. 2023. A Method for Feature Subset Selection in Software Product Lines. *Int. J. Softw. Innov.* 11, 1 (2023), 1–22. https://doi.org/10.4018/ijsi.315654

[25] Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 748–766. https://doi.org/10.1145/2814270.2814305

[26] Matthew A. Hammer, Jana Dunfield, Kyle Headley, Monal Narasimhamurthy, and Dimitrios J. Economou. 2018. Fungi: Typed incremental computation with names. *CoRR* abs/1808.07826 (2018). http://arxiv.org/abs/1808.07826

[27] Richard W Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* 29, 2 (1950), 147–160.

[28] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Software Eng.* 40, 7 (2014), 650–670. https://doi.org/10.1109/TSE.2014.2327020

[29] Christian Kästner and Sven Apel. 2008. Type-Checking Software Product Lines - A Formal Approach. In *Proceedings of 23rd IEEE International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE Computer Society, 258–267. https://doi.org/10.1109/ASE.2008.36

[30] KernelCI 2023. *KernelCI Homepage*. Retrieved 2023 from https://kernelci.org/

[31] Seyedehzahra Khoshmanesh and Robyn R. Lutz. 2018. The Role of Similarity in Detecting Feature Interaction in Software Product Lines. In *Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, USA, October 15-18, 2018*, Sudipto Ghosh, Roberto Natella, Bojan Cukic, Robin S. Poston, and Nuno Laranjeiro (Eds.). IEEE Computer Society, 286–292. https://doi.org/10.1109/ISSREW.2018.00020

[32] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing combinatorics in testing product lines. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, Porto de Galinhas, Brazil, March 21-25, 2011*, Paulo Borba and Shigeru Chiba (Eds.). ACM, 57–68. https://doi.org/10.1145/1960275.1960284

[33] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don S. Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 257–267. https://doi.org/10.1145/2491411.2491459

[34] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in modeling performance of highly configurable software systems. *Softw. Syst. Model.* 18, 3 (2019), 2265–2283. https://doi.org/10.1007/s10270-018-0662-9

[35] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *Proceedings of the 33rd International Conference on Automated Software Engineering, 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 76–86. https://doi.org/10.1145/3238147.3238196

[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[37] Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *Proceedings of the 41st International Conference on Software Engineering, 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1234–1244. https://doi.org/10.1109/ICSE.2019.00125

[38] Jackson A. Prado Lima, Willian Douglas Ferrari Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *Proceedings of the24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 31:1–31:11. https://doi.org/10.1145/3382025.3414967

[39] Jackson A. Prado Lima, Willian Douglas Ferrari Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 31:1–31:11. https://doi.org/10.1145/3382025.3414967

[40] Urtzi Markiegi, Aitor Arrieta, Leire Etxeberria, and Goiuria Sagardui. 2019. Test case selection using structural coverage in software product lines for time-budget constrained scenarios. In *Proceedings of the 34th ACM Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*, Chih-Cheng Hung and George A. Papadopoulos (Eds.). ACM, 2362–2371. https://doi.org/10.1145/3297280.3297512

[41] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2022. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Trans. Software Eng.* 48, 11 (2022),

4274–4290. https://doi.org/10.1109/TSE.2021.3116768
[42] Maven. Online; 2022. *Apache Maven: a software project management and comprehension tool.* Retrieved 2022 from https://maven.apache.org/
[43] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. 2014. Mining Co-change Information to Understand When Build Changes Are Necessary. In *Proceedings of 30th International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 241–250. https://doi.org/10.1109/ICSME.2014.46
[44] Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 141–150. https://doi.org/10.1145/1985793.1985813
[45] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. 2015. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empir. Softw. Eng.* 20, 6 (2015), 1587–1633. https://doi.org/10.1007/s10664-014-9324-x
[46] Ninja. Online; 2022. *Ninja, a small build system with a focus on speed.* Retrieved 2022 from https://ninja-build.org/
[47] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer. https://doi.org/10.1007/3-540-28901-1
[48] Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. 2022. Towards Incremental Build of Software Configurations. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results 2022, Pittsburgh, PA, USA, May 22-24, 2022*, Liliana Pasquale and Christoph Treude (Eds.). IEEE/ACM, 101–105. https://doi.org/10.1109/ICSE-NIER55298.2022.9793538
[49] Georges Aaron Randrianaina, Xhevahire Tërnava, Djamel Eddine Khelladi, and Mathieu Acher. 2022. On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study. In *Proceedings of 44th International Conference on Software Engineering, 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1584–1596. https://doi.org/10.1145/3510003.3510190
[50] Ashish Saini, Raj Kumar, Amrita Kumari, Satendra Kumar, and Mukesh Kumar. 2023. Exploratory Testing of Software Product Linesusing Distance Metrics. In *Proceedings of the 2023 International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES)*. 542–546. https://doi.org/10.1109/CISES58720.2023.10183535
[51] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. 2022. Static data-flow analysis for software product lines in C. *Autom. Softw. Eng.* 29, 1 (2022), 35. https://doi.org/10.1007/s10515-022-00333-1
[52] Hyunmin Seo, Caitlin Sadowski, Sebastian G. Elbaum, Edward Aftandilian, and Robert W. Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 724–734. https://doi.org/10.1145/2568225.2568255
[53] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A model for detecting faults in build specifications. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 144:1–144:30. https://doi.org/10.1145/3428212
[54] SQLite 2023. *How To Compile SQLite.* Retrieved 2023 from https://sqlite.org/howtocompile.html
[55] SQLite 2023. *SQLite Forum.* Retrieved 2023 from https://sqlite.org/forum/forum
[56] Sqlite-jdbc 2023. *Sqlite-jdbc Makefile.* Retrieved 2023 from https://github.com/xerial/sqlite-jdbc/blob/master/Makefile
[57] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2022. Efficient static analysis and verification of featured transition systems. *Empir. Softw. Eng.* 27, 1 (2022), 10. https://doi.org/10.1007/s10664-020-09930-8
[58] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G. J. van den Brand. 2014. Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 401–405. https://doi.org/10.1109/ICSME.2014.62
[59] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33. https://doi.org/10.1145/3280986
[60] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2014. Build system analysis with link prediction. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong (Eds.). ACM, 1184–1186. https://doi.org/10.1145/2554850.2555134
[61] Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. 2014. Build Predictor: More Accurate Missed Dependency Prediction in Build Configuration Files. In *Proceedings of the IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*. IEEE Computer Society, 53–58. https://doi.org/10.1109/COMPSAC.2014.12

Received 2023-09-28; accepted 2024-04-16