Accelerating Build Dependency Error Detection via Virtual Build

Rongxin Wu
School of Informatics
Xiamen University
Xiamen, China
wurongxin@xmu.edu.cn

Minglei Chen
School of Informatics
Xiamen University
Xiamen, China
mlchen@stu.xmu.edu.cn

Chengpeng Wang
The Hong Kong University of Science and Technology
Hong Kong, China
cwangch@cse.ust.hk

Gang Fan
Ant Group
Shenzhen, China
fangang@antgroup.com

Jiguang Qiu
Meiya Pico Information Co., Ltd
Xiamen, China
qiujg@300188.cn

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

ABSTRACT

Build scripts play an important role in transforming the source code into executable artifacts. However, the development of build scripts is typically error-prone. As one kind of the most prevalent errors in build scripts, the dependency-related errors, including missing dependencies and redundant dependencies, draw the attention of many researchers. A variety of build dependency analysis techniques have been proposed to tackle them. Unfortunately, most of these techniques, even the state-of-the-art ones, suffer from efficiency issues due to the expensive cost of monitoring the complete build process to build dynamic dependencies. Especially for large-scale projects, such the cost would not be affordable.

This work presents a new technique to accelerate the build dependency error detection by reducing the time cost of the build monitoring. Our key idea is to reduce the size of a program while still preserving the same dynamic dependencies as the original one. Building the reduced program does not generate a real software artifact, but it yields the same list of dependency errors and meanwhile speeds up the process. We implement the tool VirtualBuild and evaluate it on real-world projects. It is shown that it detects all the dependency errors found by existing tools at a low cost. Compared with the state-of-the-art technique, VirtualBuild accelerates the build process by 8.74 times, and improves the efficiency of error detection by 6.13 times on average. Specifically, in the large-scale project LLVM that contains 5.67 MLoC, VirtualBuild reduces the overall time from over four hours to 38.63 minutes.

CCS CONCEPTS

• **Theory of computation** → **Program verification**; • **Software and its engineering** → *Software defect analysis*; **Software maintenance tools**; • **Social and professional topics** → **Software maintenance**.

KEYWORDS

Dependency error, build system, build maintenance

ACM Reference Format:
Rongxin Wu, Minglei Chen, Chengpeng Wang, Gang Fan, Jiguang Qiu, and Charles Zhang. 2022. Accelerating Build Dependency Error Detection via Virtual Build. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556930

1 INTRODUCTION

Large-scale software projects typically rely on the build systems and the corresponding build scripts to manage the build process. Developers define the rules in the build scripts to specify a series of key elements to transform the source code into executable artifacts, including the build targets, the prerequisites of each target, and the recipes for building the targets. Due to the complexity of dependency relations, it is difficult for developers to make the build scripts bug-free. Among a variety of build errors, build dependency error takes the leading position, accounting for around 53%-65% [26], and can be mainly summarized into two categories: missing dependencies (MDs) and redundant dependencies (RDs) [8]. Essentially, both types of errors are due to the inconsistency between the static dependencies (i.e., the dependencies defined in the build scripts) and the dynamic dependencies (i.e., the dependencies needed in build time) [8].

Despite the importance and severity of the build dependency error, its detecting is typically labor-intensive and time-consuming [22]. To tackle this problem, various automatic techniques have been proposed, which can be summarized into three categories. The first category of the studies leverages the static dependency relation extracted from the build scripts and then discovers the erroneous patterns from the relation [2, 31]. The unsoundness of the static dependency relation greatly limits its analysis capability, making it only work for the error cases such as the cyclic dependencies. The second category of the studies monitors the build process to obtain the dynamic dependencies. Due to the lack of the static dependency relation, these techniques either fail to detect MDs and RDs [23, 24] or suffer from the efficiency issue [18]. The third category of the studies [5, 6, 8, 28, 29] detects the dependency errors with higher efficiency than the aforementioned

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556930

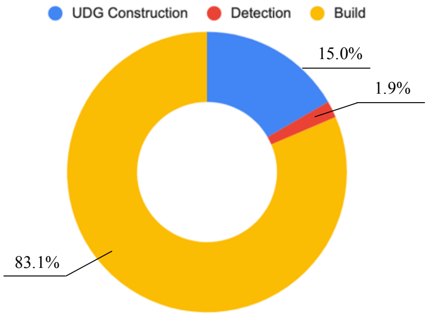


Figure 1: The overhead proportion in VeriBuild

ones. Specifically, the analysis contrasts the two kinds of dependencies to discover the inconsistency and essentially works for the errors, including MDs and RDs. A representative and state-of-the-art technique, VeriBuild[8], leverages a unified dependency graph (UDG) which fuses the static and dynamic dependencies, to greatly improve the detection efficiency. However, we still observe an unneglectable overhead incurred by monitoring the build process to capture the dynamic dependencies. As shown in Fig. 1, the performance bottleneck of VeriBuild is the build process, which takes up 83.1% of the total time.

In this work, we aim to accelerate the build dependency error detection by improving the efficiency of building the project. Our idea originates from an important observation that the dynamic dependencies obtained in the build monitoring are mainly determined by particular preprocessor directives, including the #include directive and #define directive. The bodies of the functions, which take up a large proportion of the source code, have no impact on capturing the dynamic dependencies. Thus, we propose the concept of the *virtual build* and construct a new program for the build monitoring. Intuitively, the new program preserves the original dynamic dependencies, while it takes much less time to build the program than the original one.

At a high level, our approach consists of two main phases. First, we perform the program reduction by removing the program constructs which do not affect the dynamic dependencies. Specifically, we only preserve the preprocessor directives in the source code, empty the body of each main function, and eliminate all the other function definitions. Although the new program does not behave the same as the original one at runtime, it still preserves the dynamic dependencies during the build process and can guarantee the soundness of dynamic dependency extraction. Second, we perform the virtual build, which compiles the new program and monitors the build to capture dynamic dependencies, which can be adapted to the existing works directly. Finally, we can infer a UDG from our virtual build process, which can serve as the ingredients for the dependency error detector in the subsequent phase. Obviously, our approach can seamlessly accelerate any analyzer reasoning upon the UDG, showing its great potential in detecting dependency errors efficiently for large-scale projects.

To demonstrate the effectiveness, we implement a tool named VirtualBuild by optimizing the UDG generator of the state-of-the-art approach VeriBuild, and evaluate it upon 38 open-source projects, which are the same experimental subjects of VeriBuild. Our experimental results show that VirtualBuild improves the efficiency of build dependency error detection significantly. It finishes the analysis of each project in 38.63 minutes, and detects the build dependency errors 6.13 times faster than VeriBuild. Specifically, VirtualBuild only takes 38.10 minutes to generate the UDG for the project LLVM with over five MLoC, while the time cost of VeriBuild is more than four hours. Besides, the soundness and the precision do not sacrifice in our approach, as VirtualBuild generates the same reports as VeriBuild. The results provide the strong evidence that VirtualBuild can benefit any UDG-based dependency error detector, accelerating them significantly without affecting the dependency error detection capability.

To sum up, we make the following contribution in this work.

- We introduce the concept of the *virtual build*, which utilizes a smaller program to extract dynamic dependencies of the original program with lower overhead.
- We propose the technique of the *program reduction* to construct the program for the virtual build.
- We implement VirtualBuild and evaluate it upon real-world programs. The results demonstrate the significant efficiency improvement, showing that VirtualBuild can benefit any UDG-based detector seamlessly.

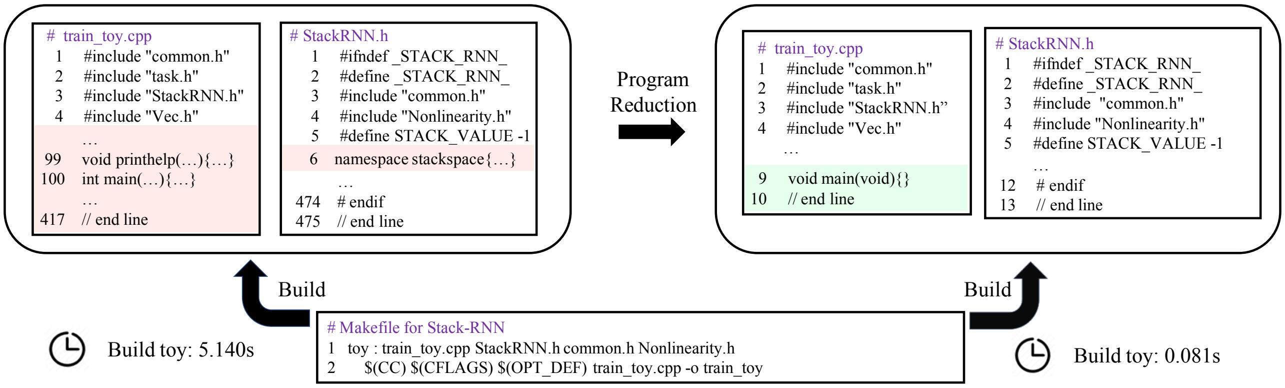
The organization of the paper is as follows. § 2 presents the overview of VirtualBuild, demonstrating the key idea and the technical challenges. We provide several preliminaries in § 3, including the program syntax and dependency model, and formalizes the virtual build dependency problem. § 4 presents the technical details, followed by the implementation details in § 5. The experimental results are presented in § 6. We discuss the related works in § 7, and conclude the paper in § 8.

2 OVERVIEW OF VIRTUALBUILD

In this section, we first introduce our motivation, and then outline our key idea of VirtualBuild. Finally, we demonstrate the technical challenges of this work.

2.1 Background

Dynamic dependencies reflect the dependency needed in the build time, and are the fundamental information for the existing studies of build dependency errors detection [6, 8, 18, 30]. To extract the dynamic dependencies, the existing approaches rely on the process named *build monitoring*, to monitor the whole building procedure, and capture all the relevant information about the file operations and processes via certain system call tracing tools (such as *ptrace*, *strace*, and so on). Build monitoring typically accounts for a large proportion of time and is the major performance bottleneck of the detection. We investigated the state-of-the-art build dependency error technique VeriBuild [8], and found that the build monitoring takes up 83.1% of the total analysis time in their evaluation results, as shown in Fig. 1. Therefore, the performance of the build dependency error detection can be improved significantly if we can reduce the overhead of the build monitoring.



```
# train_toy.cpp
#include "common.h"
#include "task.h"
#include "StackRNN.h"
#include "Vec.h"
…
void printhelp(…){…}
int main(…){…}
…
// end line
```

```
# StackRNN.h
#ifndef _STACK_RNN_
#define _STACK_RNN_
#include "common.h"
#include "Nonlinearity.h"
#define STACK_VALUE -1
namespace stackspace{…}
…
# endif
// end line
```

```
# train_toy.cpp
#include "common.h"
#include "task.h"
#include "StackRNN.h"
#include "Vec.h"
…
void main(void){}
// end line
```

```
# StackRNN.h
#ifndef _STACK_RNN_
#define _STACK_RNN_
#include  "common.h"
#include "Nonlinearity.h"
#define STACK_VALUE -1
…
# endif
// end line
```

```
# Makefile for Stack-RNN
toy : train_toy.cpp StackRNN.h common.h Nonlinearity.h
    $(CC) $(CFLAGS) $(OPT_DEF) train_toy.cpp -o train_toy
```

Figure 2: A motivating example.

2.2 Key Idea

Our idea of mitigating the overhead originates from the observation that only a small proportion of program constructs determine the dynamic dependencies during the build monitoring. Therefore, instead of compiling the complete programs, if we use a simplified version of the program which still preserves the same dynamic dependencies as the original ones, it is highly expected to reduce the time cost of the build procedure. Since our idea of building the project in a reduced version eventually generates a virtual software artifact, we refer to such build process as *virtual build*. Fig. 2 shows a motivation example of our idea, which is a C/C++ project built by Makefiles. In this example, we find that the preprocessor directives [7], i.e., the lines included in the code of programs preceded by the symbol “#”, induce the dynamic dependencies in the program. Meanwhile, the implementations of the functions do not have any influence on the dynamic dependencies. As such, we convert the source file train_toy.cpp and StackRNN.h in the left-hand side into a reduced version which is shown on the right-hand side. The reduced program only preserves the preprocessor directives, empty the function body of each main function, and remove all the non-main functions. By monitoring the build processes for the two versions of the project, we found that the dynamic dependencies remain the same, while the time cost is reduced by 98.4% (from 5.14s to 0.081s).

2.3 Technical Challenges

The effectiveness of the idea of the virtual build essentially relies on how to conduct the program reduction. There are two major technical challenges that need to be addressed in this work. First, the program reduction approach should not only preserve the programs’ statements that may induce the dynamic dependencies during the build time, but also guarantee the success of program compilation. Second, the time cost of program reduction and the virtual build should be much cheaper than the cost of the original build procedure so that the dependency error detection can benefit from it and obtain better efficiency.

Since we mainly target at detecting the build dependency errors in the make-based building systems, which are typically used for C/C++ projects, we limit the scope of the program reduction to C/C++ programs in this work. To address the above challenges, we resort to C/C++ syntax and grammar to design a grammar-guided translation approach to systematically abstract away the program constructs that do not affect the dynamic dependencies and compilation success. Moreover, our translation approach is essentially based on a context-free grammar (CFG) and the complexity of the CFG parser is linear to the program size [14]. Therefore, the program reduction features with low overhead and further ensures the high efficiency of our approach.

$$
\begin{array}{rrcl}
\text{Program} & P & := & \{F_s \mid F_h \mid F_b\} \\
\text{Source File} & F_s & := & D\ F_s \mid C_f\ F_s \mid \varepsilon \\
\text{Header File} & F_h & := & D\ F_h \mid C_s\ F_h \mid \varepsilon \\
\text{Build Script} & F_b & := & R+ \\
\text{Preproc Directive} & D & := & \text{\#include } str \mid \text{\#define } str\ str \mid \cdots \\
\text{Func Def} & C_f & := & C_s\ \{B\} \\
\text{Func Signature} & C_s & := & Ty\ \ f\ (Ty\ \ p)^* \\
\text{Func Body} & B & := & St;\ B \mid \varepsilon \\
\text{Dependency rule} & R & := & tgt : (pre)^*\ recipe
\end{array}
$$

Figure 3: The program syntax.

3 PROBLEM FORMULATION

This section first presents the syntax of a program (§ 3.1) and then defines the dependency model for the program (§ 3.2). At the end of the section, we formalize the virtual build dependency problem (§ 3.3), which is the key concern of our work.

3.1 Program Syntax

We formalize the program syntax in Fig. 3. Different from the commonly-used program syntax, it defines the syntax of multiple forms of the files in the program, including the source files, header files, and build scripts. Without the loss of generality, we assume that a source file and a header file can both contain the preprocessor directives at the beginning of the files. Two common preprocessor directives are the #include and #define directives, which begin with the string “#include” and “#define”, respectively. A source file can also contain a series of function definitions, and a

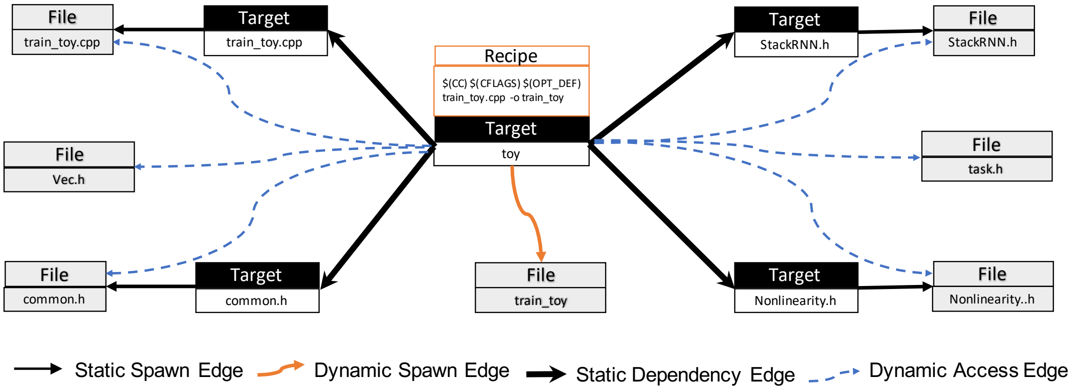


Figure 4: An example of UDG.

header file might contain the function signatures, where Ty is the type of a parameter or the return value. The function definitions and the signatures are both sentences of a context-free language. Lastly, a build script contains a series of dependency rules, which consist of the target, the prerequisites, and the recipe. A program has a unique default target, which can be built without specifying its name. In this work, we only consider the dependency errors in the build of the default target.

Overall, the program syntax is essentially defined by a context-free grammar. Particularly, the preprocessor directives can be regard by the sentences with specific prefixes, such as "#include" and "#define". According to the documentation of preprocessor directive [7], we introduce a finite set of string literals $\mathcal{L}$ to enumerate all the literal values of the suffixes of the preprocessor directives. In our paper, we use the above syntax to formalize the problem and our approach. For more general programs, we can extend the grammar to depict its syntax, which is still context-free.

3.2 Dependency Model

We borrow the dependency model in VeriBuild [8] to describe the dependencies in the build process. Specifically, we leverage a Unified Dependency Graph (UDG) to abstract the dependencies of a program and establish the build dependency error detection upon the UDG. We provide the formal definition of UDG as follows.

Definition 3.1. (Unified Dependency Graph) A Unified Dependency Graph $UDG = (V, E)$, where V and E are the sets of the nodes and edges, repetitively. Specifically, a target node $v_t \in V_T \subseteq V$ represents a target in a build script, and a file node $v_f \in V_F \subseteq V$ represents a file in the disk. An edge has four forms as follows:

- A static dependency edge $e_{SE} \in E_{SE} \subseteq V_T \times V_T$ indicates the dependency between two targets defined in the scripts.
- A static spawn edge $e_{SS} \in E_{SS} \subseteq V_T \times V_F$ indicates that the file is statically available before the build of the target.
- A dynamic access edge $e_{DA} \in E_{DA} \subseteq V_T \times V_F$ indicates the file is accessed in the build of the target.
- A dynamic spawn edge $e_{DS} \in E_{DS} \subseteq V_T \times V_F$ indicates the file is generated or modified in the build of the target.

Particularly, the sets of dynamic access edges and dynamic spawn edges depict the dynamic dependencies of the program.

Example 3.2. Fig. 4 shows the UDG for the target toy in Fig. 2. There are four prerequisites specified for the target toy, so there are four static dependency edges starting from it. The four prerequisites are the existing files, and the dependency relations are indicated by four static spawn edges. Besides, five header files and one source file are accessed in the build of the target toy, which yields six dynamic access edges. A file named train_toy is created as the output of the build, which is indicated by a dynamic spawn edge.

Intuitively, a UDG actually provides minimal but sufficient information for build dependency error detection. As long as we obtain a sound UDG for a given program, we can design the sound detectors for various kinds of dependency errors, such as MDs and RDs [6, 8, 28]. By traversing the UDG with a specific policy, we can collect the dependency fact for each dependency rule, enabling us to discover a variety of dependency errors. The detection process is quite standard as long as the UDG is generated. We follow the existing approaches to detect the errors upon the UDG [6, 8], of which the details are not discussed in this work.

Example 3.3. When building the target toy in Fig. 2, we can detect two MDs by traversing the UDG in Fig. 4. Two header files, namely Vec.h and task.h, are not specified as the prerequisites, while they are accessed in the build process.

3.3 Problem Statement

Before stating the problem we focus on, we first provide a formalization of the UDG-based build dependency error detector as follows, based on which we establish our problem further.

Definition 3.4. (UDG-based Build Dependency Error Detector) A UDG-based build dependency error detector $\mathcal{D}$ is a pair of mappings $(\mathcal{D}_g, \mathcal{D}_a)$, Here, $\mathcal{D}_g$ maps a pair of a program P and a target t to a UDG corresponding to the build of the target t, and $\mathcal{D}_a$ maps a UDG to a list of build dependency errors.

As shown by Definition 3.4, the detector is essentially a composition of two parts, namely the UDG generator and the UDG analyzer, which correspond to $\mathcal{D}_g$ and $\mathcal{D}_a$, respectively. According to our observation, the build monitoring takes up a large proportion of time overhead, which evidences that the computation of $\mathcal{D}_g$ is time-consuming. To achieve the efficient detection of build dependency errors, we aim to find another program to support the UDG generation with lower overhead but yield the same UDG for further analysis. Formally, we formalize the problem as the virtual build dependency problem.

Definition 3.5. (Virtual Build Dependency Problem) Given a program P with the default target t^* and a UDG-based build dependency error detector $\mathcal{D}$, we aim to find a new program P' and a UDG-based build dependency error detector $\mathcal{D}'$ such that:

- The UDG analyzers of $\mathcal{D}$ and $\mathcal{D}'$ are the same, i.e., $\mathcal{D} = (\mathcal{D}_g, \mathcal{D}_a)$ and $\mathcal{D}' = (\mathcal{D}'_g, \mathcal{D}_a)$.
- Utilizing P and P', $\mathcal{D}'_g$ induces the same UDG as $\mathcal{D}_g$, i.e., $\mathcal{D}_g(P, t^*) = \mathcal{D}'_g(P \cup P', t^*)$.
- The total cost of computing $\mathcal{D}'_g(P \cup P', t^*)$ is lower than the one of computing $\mathcal{D}_g(P, t^*)$.

Example 3.6. In Fig. 2, the header file StackRNN.h and the source file train_toy.cpp on the right-hand side are the program P' we need to solve the virtual build dependency problem. Several particular program constructs, such as the non-main functions and the body of the main function, are removed from the original ones. With the new program, we can utilize any existing UDG generator $\mathcal{D}_g$ to construct the UDG more efficiently, which essentially yields a new UDG generator $\mathcal{D}'_g$. Finally, P' and $(\mathcal{D}'_g, \mathcal{D}_a)$ are exactly the solution of the virtual build dependency problem.

In this work, we only concentrate on the acceleration of the UDG generators, and reuse the UDG analyzers in the existing works. Although it is also an interesting problem to improve the efficiency in analyzing the UDG, we believe the virtual build dependency problem is meaningful enough due to the huge overhead of the build process. An effective solution can significantly improve the efficiency of the build dependency error detection, promoting the practicality of original detectors seamlessly.

4 APPROACH

This section presents the details of our approach, including the program reduction (§ 4.1) and the virtual build (§ 4.2). At the end of the section, we present the whole procedure of detecting build dependency errors via virtual build (§ 4.3).

4.1 Program Reduction

As demonstrated above, the build process on which the UDG generation depends is quite time-consuming for large-scale programs. To accelerate the generation and obtain the same UDG, we start from the observation that the build dependencies of a program do not rely on several particular program constructs, and only preprocessor directives really matter for the dynamic dependency extraction. For example, the body of a function does not have any impact on the build dependency, which implies that we can still obtain the same UDG after removing all the statements in each function. Furthermore, only the existence of the main function really matters for the build of a target, so we can directly eliminate all the non-main functions from the source code. Given a program in the syntax shown in Fig. 3, only #include directives, #define directives, and the content of the build scripts determine the dependency model. Thus, it is possible to construct a new program that only contains such program constructs to preserve the equivalence of the dependency model, while the build process could be achieved much more efficiently than the one upon the original program.

Algorithm 1: Program reduction

```
   Input: $P$: A program;
   Output: $P'$: A reduced program;
1  $P' \leftarrow \emptyset$;
2  foreach $F \in P$ do
3      /* Process build scripts */
4      if Type($F$) = Build Script then
5          $F_v \leftarrow F$;
6      else
7          /* Process header files and source files */
8          $F_v \leftarrow$ EmptyFile;
9          foreach $C \in F$ do
10             if Prefix($C$) $\in \mathcal{L}$ then
11                 $F_v \leftarrow$ append($F_v, C$);
12             if $C$ is main function then
13                 $C_v \leftarrow$ EmptyMainFunc;
14                 $F_v \leftarrow$ append($F_v, C_v$);
15     $P' \leftarrow P' \cup \{F_v\}$;
16 return $P'$;
```

To obtain the program for a more efficient build, we perform the program reduction on a given program. Alg. 1 shows the procedure of the program reduction, which takes as input a program P and outputs a reduced program P'. Essentially, the reduction process is a grammar-guided translation based on the syntax in Fig. 3. For each build script in the program P, we preserve its content unchanged so that the edge sets E_{SE} of the UDGs for P and P' are the same (lines 3 to 4). For each source file or header file, we only store the preprocessor directives (lines 10 to 11), and remove all the functions except for the main functions (lines 12 to 14). Particularly, we empty the function body of each main function (lines 12 to 13), as it does not affect the static dependency relation or the build monitoring process. If a file does not contain any content after the reduction, we still store it in the disk to ensure that it is available in the build process, which is the necessity of a successful build. Finally, we enumerate all the files together to form a new program P' (line 15), which is what we need for the further virtual build.

Example 4.1. Consider the program in Fig. 2. Based on the syntax, we can identify the #include and #define directives precisely, and preserve the lines 1 to 4 of the file train_toy.cpp and the lines 1 to 5 of the file StackRNN.h in the new program. To make the target build and generate output files successfully, we remain the main function in the file train_toy.cpp of the new program, while we empty its body to avoid the unnecessary compile process.

Intuitively, the reduced program P' abstracts away the constructs which do not affect the build dependencies. Its reduced size can significantly reduce the compilation overhead in the build process. Lastly, it is worth noting that the program reduction in Alg. 1 runs with quite low overhead. Assume that the program conforms to the

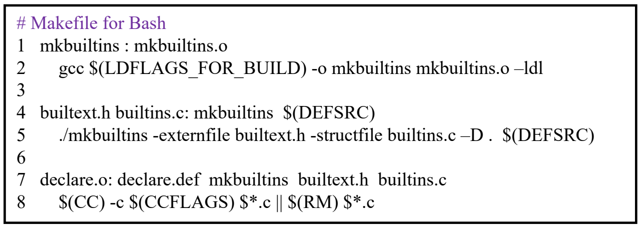
Algorithm 2: Virtual build

```
Input: $P$: A program;  $\mathcal{D}_g$: a UDG generator;
Output: $UDG$: Unified dependency graph of $P$;
1  /* Program reduction */
2  $P' \leftarrow$ ReduceProgram($P$);
3  /* Virtual build */
4  foreach $t \in$ Targets($P'$) do
5      $status \leftarrow$ Build($P', t$);
6      if $status$ is $Success$ then
7          $UDG_t \leftarrow \mathcal{D}_g(P', t)$;
8      else
9          /* Fall back to actual build */
10         $UDG_t \leftarrow \mathcal{D}_g(P, t)$;
11 /* Merge the UDGs */
12 $UDG = \bigcup_{t \in \mathtt{Targets}(P')} UDG_t$;
13 return $UDG$;
```

syntax in Fig. 3. The program reduction essentially identifies a series of strings with specific prefixes, such as "#include" and "#define" in the set $\mathcal{L}$, and further simplifies them in specific manners, which can be achieved in the linear time complexity to the size of the program. Our evaluation also evidences that the program reduction only introduces little overhead even if the program is large-scaled.

4.2 Virtual Build

With the benefit of the reduced program, we can extract the dynamic dependencies by performing the build monitoring upon it, which enables us to generate the UDG efficiently. According to our investigation of real-world programs, several targets in the build scripts are built by an executable file generated by its prerequisites. For example, in the build script of the project Bash shown in Fig. 5, the target declare.o takes the targets builtext.h and builttins.c as its prerequisites, both of which are built by the output file of building the target mkbuiltins, i.e., the file mkbuiltins.o. To perform the build monitoring, we have to build the two targets using the original program. Otherwise, the target declare.o at the line 7 can not be successfully built due to the lack of the files builtext.h and builttins.c. Inspired by this observation, we design the virtual build upon two programs, and fall back to the actual build if an error occurs in the build of the reduced program.



```
# Makefile for Bash
1  mkbuiltins : mkbuiltins.o
2      gcc $(LDFLAGS_FOR_BUILD) -o mkbuiltins mkbuiltins.o –ldl
3
4  builtext.h builtins.c: mkbuiltins  $(DEFSRC)
5      ./mkbuiltins -externfile builtext.h -structfile builtins.c –D .  $(DEFSRC)
6
7  declare.o: declare.def  mkbuiltins  builtext.h  builtins.c
8      $(CC) -c $(CCFLAGS) $*.c || $(RM) $*.c
```

Figure 5: An example of dynamically generated executable files for building targets

Algorithm 3: Dependency error detection

```
Input: $P$: A program;
       $\mathcal{D}$: A UDG-based build dependency error detector;
Output: $Errs$: A list of build dependency errors;
1  $\mathcal{D}_g \leftarrow$ GetUDGGenerator($\mathcal{D}$);
2  $\mathcal{D}_a \leftarrow$ GetAnalyzer($\mathcal{D}$);
3  /* Generate UDG with virtual build */
4  $UDG \leftarrow$ VirtualBuild($P, \mathcal{D}_g$);
5  /* Dependency error detection */
6  $Errs \leftarrow \mathcal{D}_a(UDG)$;
7  return $Errs$;
```

Alg. 2 shows the overall procedure of the virtual build. By invoking the procedure shown in Alg. 1, we can obtain the reduced program P'(lines 1 to 2). Then, we attempt to build each target that is the direct or indirect prerequisite of the default target in the reduced program P' (lines 4 to 10). If the target t in the reduced program P' can be successfully built, we can directly construct its UDG by invoking the existing UDG generator upon the reduced program (lines 6 to 7). Otherwise, we roll back to the actual build upon the original program (lines 9 to 10). Finally, we merge all the UDGs together by the graph union operator $\cup$, yielding a UDG equivalent to $\mathcal{D}_g(P)$ (lines 11 to 12). Therefore, Alg. 2 can further ensure that the dependency errors detected upon the output UDG are the same as $\mathcal{D}$. We present the details of adopting the virtual build to the detection in § 4.3.

Example 4.2. Consider the dependency rule at line 7 in the build script shown in Fig. 5. At the beginning, other targets are built successfully upon the reduced program. The failure of building the target declare.o enforces the virtual build fall back to the actual build. This further triggers the actual build of their transitive prerequisites, and finally generates the executable file mkbuiltins.o to support the build of the targets builtext.h and builtins.c.

The virtual build in Alg. 2 is essentially a hybrid mechanism of generating the UDG of a program. To reduce the overhead of the build, it eagerly attempts to obtain dynamic dependencies in the build of the reduced program. To ensure the program is built successfully, Alg. 2 shifts to the actual build if necessary. Although it can fall back to the actual build of the default target in the worst case, our evaluation can show that there are quite a few cases in which the actual build is triggered. Even if several targets are built upon the original program, other targets can still be able to be built by the virtual build process upon the reduced program, which still saves the time consumption in the UDG generation.

4.3 Dependency Error Detection

As formulated in Definition 3.4, we decompose a UDG-based dependency error detector $\mathcal{D}$ into two components, namely the UDG generator $\mathcal{D}_g$ and the UDG analyzer $\mathcal{D}_a$. With the benefit of the virtual build, we can construct another UDG generator $\mathcal{D}'_g$ to achieve the acceleration. In this work, we only focus on the optimization of the build process and directly reuse the UDG analyzer $\mathcal{D}_a$ for the further detection phase upon the UDG.

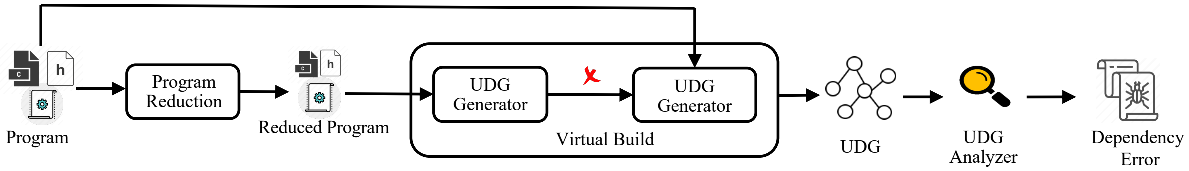


Figure 6: Schematic overview of VirtualBuild

Leveraging the virtual build-based UDG generator, we can obtain the overall build dependency error detection algorithm shown in Alg. 3. It takes as inputs any UDG-based detector $\mathcal{D}$ and a program P, and finally returns the same dependency errors in the program as $\mathcal{D}$ does. Our acceleration is mainly achieved at line 4, avoiding the actual build as much as possible. Intuitively, Alg. 3 constructs a new UDG-based detector, of which the UDG generator is much more efficient than the one of $\mathcal{D}$, and the UDG analyzer is the same as the original one. Thus, our approach can improve any UDG-based detector seamlessly [8, 28], showing its generality of accelerating the build dependency error detection.

5 IMPLEMENTATION

We implement our approach as a tool named VirtualBuild to accelerate the build dependency error detection. VirtualBuild is an extension of the state-of-the-art dependency error detector VeriBuild [8]. It reuses the UDG analyzer of VeriBuild to hunt the build dependency errors while leveraging the virtual build to improve the efficiency of generating the UDG. Fig. 6 shows the architecture of VirtualBuild. It takes as inputs the program and then performs the program reduction to generate the reduced program. The two programs are fed to the module of the virtual build simultaneously, fusing two kinds of build processes to accelerate the UDG generation and ensure the correctness of the UDG.

To achieve the program reduction, we utilize the parser generator ANTLR [1] and customize a parser to identify the irrelevant program constructs that do not affect the result of the build monitoring. By removing irrelevant constructs, VirtualBuild generates a new program with a much smaller size after the reduction. Particularly, the body of each main function is set to empty so that the output files of the related targets can be generated with lower overhead. It is worth mentioning that VirtualBuild only performs the syntactic analysis in the program reduction, which can be achieved in the linear time complexity to the program size. The high efficiency of the program reduction is also evidenced by our evaluation.

For a real-world program, it can include a large number of the system files, which also affects the overhead of the build process. In our implementation, we perform the program reduction upon the system files as a preprocessing phase. Thus, the reduced system files can replace the original ones to support the virtual build. Moreover, the preprocessing is only conducted one time for all the programs, as different programs share the same system files in a given environment. Hence, we directly redirect an original system file to the reduced one when we analyze a specific program.

In the virtual build, we adopt the build monitoring module in VeriBuild to obtain the dynamic dependencies. Different from VeriBuild, we utilize LD_PRELOAD trick to monitor build process instead of utilizing *strace*. It can decrease the overhead of the build monitoring significantly. Besides, we observe that several source files and header files can be generated in the build process. To handle such files, we attempt to identify them in the virtual build and reduce their sizes by performing the program reduction.

Lastly, the UDG generator and the UDG analyzer both come from VeriBuild. We do not change the major implementation of VeriBuild and simply adapt it to our virtual build and detection phases. However, we found several implementation flaws in VeriBuild, which may affect the number of dependency errors of large-scale programs. To handle these issues, we fix the incorrect implementations according to the feedback from the authors of VeriBuild, and further, compare it with VirtualBuild in the evaluation. We will not publish the source code of VirtualBuild because we plan to commercialize it in the near future.

6 EVALUATION

In this section, we evaluate the effectiveness and efficiency of VirtualBuild by investigating the research questions as follows:

- **RQ1:** What percentage of code is removed in the program reduction?
- **RQ2:** What is the efficiency improvement VirtualBuild achieves in the build dependency error detection?
- **RQ3:** What is the overhead of each stage in VirtualBuild?
- **RQ4:** Does VirtualBuild report the same build dependency errors as the existing approaches?

Highlights. VirtualBuild is quite effective and efficient in the acceleration of the build dependency error detection.

- VirtualBuild reduces the size of the program by 85.32% on average, enabling the virtual build to compile less source code than the existing approaches.
- VirtualBuild finishes analyzing any experimental project in 38.63 minutes. The UDG generation is accelerated by 6.16 times on average.
- The two stages of VirtualBuild, i.e., the program reduction and the virtual build, are efficient in analyzing large-scale projects. On average, they take 0.91 and 85.62 seconds in each analysis, respectively.
- VirtualBuild ensures the soundness and the completeness, reporting almost the same reports of build dependency errors as VeriBuild in the experiment projects.

6.1 Experimental Setting

Projects. We select 38 C/C++ projects from the benchmark of VeriBuild, which are listed in Table 1. The benchmark contains the open-source projects of various sizes, ranging from tens of lines

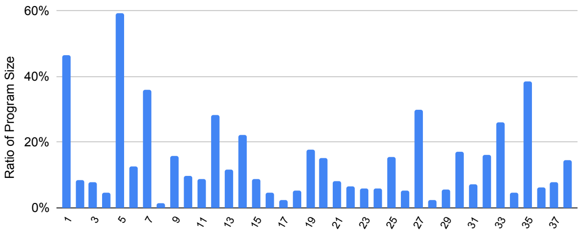


Figure 7: The size ratio of the reduced and original programs

of code to the millions of lines of code, which can demonstrate the efficiency and practicality of VirtualBuild in analyzing the large-scale programs. Four experimental subjects of VeriBuild are not C/C++ programs or are currently unavailable, so we discard them in our evaluation.

Environment. Each group of the experiments is conducted on a computer running Ubuntu 18.04.6 LTS system with an Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz forty-core processor and 512 GB physical memory.

6.2 Results

This section presents the answers to the research questions and demonstrates the efficiency and effectiveness of VirtualBuild.

6.2.1 Percentage of Removed Code. To show the change of the program size in the program reduction, we quantify the lines of the code after and before the program reduction, and compute the size ratio of the two versions. Specifically, we utilize the command cloc to measure the lines of the header files and source files. Several files are not used in the build process, so we do not consider their sizes and only count the lines of the files used in the build.

Fig. 7 shows the ratio of the reduced program size and the original program size. The projects are listed in the same order shown in Table 1. On average, a reduced program only contains 14.68% of the code in the original one, and 85.32% of the code is removed in the reduction. Several programs, especially the large-scaled programs, i.e., the project Python, contain a large number of macro definitions to support the deployment in various platforms, so the ratios are relatively larger than the ones of other programs. Besides, the project named greatest contains a large number of lines of macro definitions, which are introduced to define functions. We do not adopt particular strategies to filter such cases, so the ratio of the program sizes is much larger than others. However, the sizes of most of the projects are reduced by at least 80%. Thus, the virtual build processes much less source code than the actual build, which can reduce the overhead of the build process.

Answer to RQ1: VirtualBuild reduces the size of a program by 85.32% on average in the program reduction.

6.2.2 Efficiency Improvement. To quantify the efficiency improvement, we measure the time cost of VirtualBuild in the UDG generation and the overall analysis. Also, we compare the overhead of VirtualBuild with the one of VeriBuild to demonstrate the benefit of our approach. To have a fair comparison, we re-evaluate VeriBuild upon the subjects under our evaluation environment.

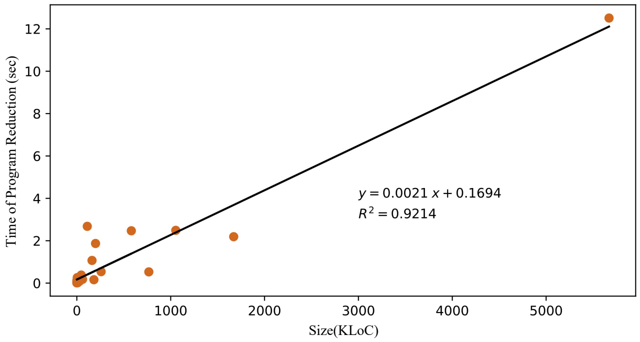


Figure 8: The regression result of the overhead of program reduction and the number of lines

Table 1 shows the experimental results. It is shown that VirtualBuild finishes the UDG generation for any experimental project in 38.10 minutes (2,286.07 seconds), and the overall analysis in 38.63 minutes (2,317.79 seconds). On average, the UDG generation and the overall analysis are accelerated by 6.16 times and 6.13 times, respectively. More importantly, the build process is accelerated by 8.74 times averagely, showing the effectiveness of virtual build. Particularly, the time overhead of detecting dependency errors in the project LLVM only takes 38.10 minutes, while VeriBuild takes over four hours to finish the overall analysis. The significant efficiency improvement benefits from the overhead reduction of the build time. For the project httpd, VirtualBuild is slightly slower than VirtualBuild, as it has to probe the targets that demand the actual build iteratively in Alg. 2, introducing the extra overhead in the virtual build. Apart from the project httpd, there are other four projects in which the virtual build can fall back to the actual build, including lighttpd, Bash, OpenSSL, and LLVM. However, in the above five projects, only a few (no more than three) targets make the virtual build fall back to the actual build. Therefore, the virtual build still benefit the overall efficiency significantly for almost all the projects, evidencing the practicality of VirtualBuild in the build script maintenance for large-scale programs.

Answer to RQ2: VirtualBuild finishes the UDG generation and the overall analysis of any project in 38.10 minutes and 38.63 minutes, and achieves the 6.16× and 6.13× speedups on average, respectively.

6.2.3 Overhead Breakdown. To investigate the overhead of each stage in VirtualBuild, we quantify the time consumption of the program reduction, the virtual build, and the UDG construction separately. As explained in § 5, we scan the system files in the preprocessing phase, and only reduce them one time. Thus, we do not consider the overhead of the reduction of the system files.

In Table 1, the columns Reduce, Build, and Construct show the overhead of the program reduction, the virtual build, and the UDG construction, respectively. According to the experimental data, the program reduction introduces little overhead to the overall analysis. On average, each project can be reduced in 0.91 second. Even for the project LLVM, which contains over 5 MLoC, VirtualBuild can finish the reduction in 13 seconds, showing its great potential in analyzing large-scaled programs. Furthermore, we adopt the

Table 1: The overhead of VirtualBuild and VeriBuild

Project	Size		VeriBuild(sec)			VirtualBuild(sec)					Speedup(×)		
	#Files	#Lines	Build	UDG	Total	Reduce	Build	Construct	UDG	Total	Build	UDG	Total
Generic-C-Project	3	5	0.24	0.35	0.35	0.01	0.15	0.11	0.27	0.27	1.58	1.29	1.29
http-parser	6	6,347	13.74	14.32	14.33	0.05	0.19	0.09	0.33	0.33	72.29	43.38	43.41
CacheSimulator	10	953	0.50	0.60	0.60	0.06	0.21	0.07	0.34	0.34	2.39	1.77	1.77
Stack -RNN	10	1,746	10.15	10.55	10.55	0.14	0.16	0.05	0.35	0.35	63.46	30.15	30.15
greatest	11	1,581	0.92	1.02	1.02	0.04	0.36	0.07	0.47	0.47	2.54	2.16	2.16
kleaver	14	734	1.22	1.52	1.53	0.06	0.48	0.27	0.81	0.81	2.54	1.87	1.89
namespaced_ parse	15	599	0.53	0.64	0.65	0.05	0.28	0.13	0.46	0.47	1.91	1.40	1.39
mpc	16	4,566	14.96	15.20	15.21	0.10	1.26	0.10	1.46	1.47	11.87	10.41	10.34
libco	30	3,798	4.75	5.03	5.04	0.07	0.92	0.24	1.23	1.23	5.16	4.09	4.10
fzy	31	3,727	0.90	1.02	1.03	0.05	0.22	0.06	0.33	0.34	4.11	3.10	3.04
cctz	33	7,782	10.46	10.78	10.80	0.04	1.05	0.18	1.27	1.28	9.96	8.49	8.44
gwion-util	34	1,238	1.00	1.14	1.15	0.13	0.36	0.10	0.59	0.60	2.79	1.94	1.92
Bftpd	37	5,003	2.72	3.17	3.18	0.07	0.74	0.39	1.20	1.21	3.68	2.64	2.63
grbl	51	4,514	1.21	1.34	1.35	0.26	0.42	0.12	0.80	0.81	2.88	1.67	1.66
gravity	56	17,817	3.83	4.24	4.26	0.11	1.22	0.33	1.66	1.68	3.14	2.55	2.54
fastText	68	6,749	10.06	10.55	10.56	0.07	0.61	0.18	0.86	0.87	16.49	12.26	12.13
8cc	82	10,165	1.63	1.78	1.79	0.02	0.48	0.11	0.61	0.62	3.40	2.92	2.89
clib	105	14,843	3.86	4.12	4.14	0.09	1.37	0.16	1.62	1.64	2.82	2.54	2.52
zlib	106	28,998	6.52	6.81	6.83	0.09	1.32	0.17	1.58	1.59	4.94	4.31	4.29
cJSON	107	19,437	6.19	6.91	6.96	0.09	2.63	0.60	3.32	3.37	2.35	2.08	2.06
LAME	146	47,124	13.13	14.74	14.78	0.38	5.18	1.55	7.11	7.14	2.53	2.07	2.07
GNU Aspell	184	30,821	45.39	46.82	46.92	0.13	11.36	1.22	12.71	12.80	4.00	3.68	3.67
neven	186	22,743	1.94	3.19	3.20	0.12	1.18	1.22	2.52	2.53	1.64	1.27	1.26
Tmux	206	52,244	28.06	29.35	29.44	0.19	7.26	0.81	8.26	8.33	3.87	3.55	3.53
lighttpd	232	61,072	43.98	47.35	47.46	0.18	28.66	3.31	32.15	32.24	1.53	1.47	1.47
tig	238	43,394	9.92	10.44	10.51	0.28	2.21	0.39	2.88	2.94	4.49	3.62	3.57
ck	244	31,743	1.90	2.94	2.95	0.22	0.56	1.01	1.79	1.80	3.40	1.64	1.64
Cppcheck	399	182,121	106.20	107.05	107.10	0.16	2.72	0.38	3.26	3.30	39.04	32.84	32.45
Bash	442	126,820	42.45	43.99	44.16	5.05	19.53	1.29	25.87	26.04	2.17	1.70	1.70
Redis	457	111,435	29.89	32.51	32.56	2.68	5.13	2.04	9.85	9.89	5.83	3.30	3.29
httpd	502	199,327	120.24	136.50	136.76	1.87	140.96	25.21	168.04	168.28	0.85	0.81	0.81
Capstone	604	161,930	32.06	33.51	33.59	1.07	4.57	0.92	6.56	6.63	7.01	5.11	5.07
OpenSSL	1,767	580,653	458.36	486.53	491.12	2.47	447.49	24.48	474.44	478.92	1.02	1.03	1.03
GMP	1,878	258,875	135.53	141.14	141.51	0.54	71.68	4.47	76.69	77.02	1.89	1.84	1.84
PHP	2,077	766,944	523.11	552.13	552.95	0.53	132.71	19.91	153.15	153.93	3.94	3.61	3.59
Python	2,645	1,053,963	125.68	129.58	129.85	2.49	83.67	2.93	89.09	89.36	1.50	1.45	1.45
OpenCV	3,746	1,670,991	2,191.72	2,222.68	2,225.04	2.19	110.72	15.72	128.63	131.02	19.80	17.28	16.98
LLVM	41,788	5,670,236	15,667.89	15,798.07	15,829.89	12.51	2,163.69	109.87	2,286.07	2,317.79	7.24	6.91	6.83
Average			**517.71**	**524.73**	**525.82**	**0.91**	**85.62**	**5.80**	**92.33**	**93.41**	**8.74**	**6.16**	**6.13**

regression analysis on the overhead of the program reduction. As shown in Fig. 8, the value of R square is 0.9214, which is quite closed to 1, indicating the gentle overhead of the program reduction.

Meanwhile, the virtual build also features low overhead. Compared with the actual build, the virtual build spends less time compiling the body of each function. Particularly, the virtual build of the project LLVM only takes less than forty minutes, while the actual build has to take more than four hours. The significant efficiency improvement demonstrates the possibility of integrating VirtualBuild in the integrated development environment or the continuous integration platform, supporting the developers to maintain the dependencies during the development phase.

We also quantify the overhead of constructing the UDG according to the dynamic dependencies obtained in the virtual build. Specifically, we reuse the original module of VeriBuild to construct the UDG, which is not our major concern in this work. It can be one of the future works to optimize the UDG construction to further decrease the overhead.

Answer to RQ3: The program reduction and the virtual build take 0.91 and 85.62 seconds per project on average, respectively, reducing the overhead of the build dependency error detection significantly.

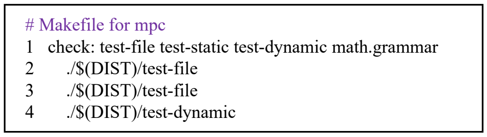
6.2.4 Soundness and Completeness. To demonstrate the soundness and completeness of the virtual build, we quantify the difference between the reports of missing dependencies (MDs) and redundant dependencies (RDs) discovered by VirtualBuild and VeriBuild.

Table 2 shows the comparison of two sets of the reports. For the MD reports, there are 9,437 MDs uncovered by both the analyses, while one MD is only reported by VeriBuild. Besides, both the analyses report 6,103 RDs, while there is one RD only reported by VeriBuild. As Table 2 indicates, the reports of VirtualBuild and VeriBuild highly overlap, showing that our approach almost has no impact on the reports of dependency errors.

Furthermore, we delve into the details of the two divergent cases. Fig. 9 shows the MD not detected by VirtualBuild. In the build of the target check, the three executable files, namely test-static,

Table 2: The comparison of MD and RD reports

Bug Type	Common Report	VeriBuild	VirtualBuild
MD	9,437	1	0
RD	6,103	1	0
Percentage	**99.99%**	**0.01%**	**0%**



```
# Makefile for mpc
check: test-file test-static test-dynamic math.grammar
    ./$(DIST)/test-file
    ./$(DIST)/test-file
    ./$(DIST)/test-dynamic
```

Figure 9: The MD not detected by VirtualBuild

test-file, and test-dynamic, are executed. The execution of the three files accesses the file named digits.txt, which is not created by any previous targets or specified as a transitive prerequisite of the target check. However, the virtual build only generates the three empty executable files, which do not access any files in the execution. Thus, digits.txt is not identified as an instance of the MD by VirtualBuild. Fortunately, such a case is not common in our evaluation, and we only discover this case in the project mpc. The other divergent case is a false positive of VeriBuild. The incorrect dynamic dependencies collected by *strace* cause the spurious RD report of VeriBuild, while our implementation of VirtualBuild avoids it with the benefit of LD_PRELOAD trick.

Lastly, Table 2 shows that VeriBuild detects more MDs and RDs than the ones reported in [8]. The major reason is that the original implementation of VeriBuild constructs the UDG in a faulty manner. After communicating with the authors [8], we fixed the bug and re-analyzed the experimental subjects for the comparison.

Answer to RQ4: The dependency errors reported by VirtualBuild are almost the same as the ones reported by VeriBuild, sharing 15,640 reports out of 15,642 ones in total.

6.3 Discussion

In this section, we discuss the threat to the validity of VirtualBuild, several limitations, and future works.

6.3.1 Threat to Validity. The major threat to the validity of our approach is whether the speedup of the experimental subjects depends on the environment. Generally, the environment of the machine affects the build time and the efficiency of other procedures in the UDG generation and analysis. To minimize the stochastic factors, we stop other user processes in the system, repeat the evaluation of a single experimental subject ten times, and choose the average time consumption of each stage as the final result. Therefore, the speedup of each subject is convincing enough to evidence the effectiveness of VirtualBuild.

6.3.2 Limitations and Future Works. Although VirtualBuild is highly efficient and effective, it still has several drawbacks that need to be tackled in the future. First, the current implementation of VirtualBuild only accelerates the build process of C/C++ programs. For other programming languages, VirtualBuild can not perform the program reduction to support the virtual build. However, our approach can be further generalized by specifying the subsets of the program constructs that affect the build monitoring process. Second, we find that several projects utilize the macros to define the functions, which are not identified and reduced by VirtualBuild. More aggressive program reduction strategies can be proposed to handle such cases. Third, VirtualBuild has to fall back to the actual build if the virtual build of the reduced program fails, in which Alg. 2 introduces the extra overhead in the build of such targets. This prevents VirtualBuild from achieving a more significant acceleration of several projects, and even slows down the detection upon the project httpd in our evaluation. In the future, VirtualBuild can be further optimized by identifying the targets that demand the actual build in advance. The static analysis of the build scripts might provide meaningful guidance to determine such targets without any build process, further reducing the time consumption of the virtual build in Alg. 2 effectively.

7 RELATED WORK

There has been a large amount of literature covering the related area, which is discussed in detail as follows.

Static analysis of build scripts. Several works establish the graph models for the dependencies encoded in the build scripts, which indicate the expected dependencies specified by the developers. For instance, Gunter [11] introduces a Petri net to depict the dependencies in a build script and performs the correctness checking upon it. Similarly, SYMake uses symbolic dependency graphs to detect the bad smells, including the cyclic and duplicated dependencies [31]. Based on the dependency graph, the missing dependencies can also be predicted by data mining techniques [36, 37]. However, the analyses can not guarantee soundness, missing many dependency issues, and suffer the low precision and recall without the dynamic dependency information.

Build script testing. Dynamic testing has shown to be effective in discovering the dynamic dependencies of a program, providing a new way of detecting the dependency errors in the build script. One typical kind of approaches attempts to monitor the build process in the runtime and infer the dependencies based on the execution state [18, 21, 25, 32]. For example, MkCheck [18] utilizes a fuzzing-like process to obtain a dependency graph, which indicates the file-level dependency relation in the build. Bazel [16] builds each build task by creating a separate environment to prevent the anonymous outcomes from being stored in the system. Other commercial tools, including Tup [27], IBM ClearCase [15] and VESTA [35], only check whether the runtime states violate the dependencies or not. Although these approaches can trigger the dependency issues with concrete executions, they can not detect the dependency issues if the build process does not trigger them, making the underlying dependency issues difficult to be discovered. Moreover, they suffer the huge overhead caused by an incredibly large number of the builds, especially in the cases of large programs. The insight underlying the virtual build can benefit the build script testing, reducing the overhead of each build task significantly.

Hybrid analysis of build scripts. Several recent works combine the advantages of the static and dynamic approaches to detect dependency errors, which have been proven to be effective for various build systems. Bezemer et al. [6] proposes a tool to detect unspecified dependencies in the Makefile systems. VeriBuild establishes a unified graph representation to encode the expected dependencies and the dynamic dependencies and designs different policies of traversal to detect the dependency issues, such as missing dependencies and redundant dependencies [8]. These hybrid approaches, similar to the dynamic ones, require a successful build of the projects, which has been shown to be quite time-consuming for the large-scale programs [8, 28]. Our work provides a general mechanism to accelerate the analyses, reducing the overhead of the build process without sacrificing soundness and completeness.

Build script refactoring. Refactoring a build script has been a popular topic in the community of software maintenance [17, 33]. For example, Makao[3] leverages an aspect-oriented approach and visualizes the dependency relation by the graphs, which enables the developers to perform the refactoring for the build scripts in a better style. Similarly, Formiga and SYMake support the renaming and the removal of the targets in the build scripts [12, 31]. Although the refactoring of a build script is relatively trivial, it is still an open problem to establish reasonable criteria and propose a new way to generate the new build scripts better than the original one. This problem has significant importance in the migration of the build systems [9, 10]. We believe the techniques of build script refactoring and the solution we processed are complementary, which can localize [4, 5, 20, 23, 34] and repair the dependency issues [13, 19] with a low overhead based on the refactoring design.

8 CONCLUSION

We have introduced VirtualBuild, a UDG-based dependency error detector to support the maintenance of build systems. It leverages a light-weighted syntactic analysis to perform the program reduction and generates the reduced program for the virtual build, supporting the efficient build monitoring in the UDG generation. By fusing the build upon the original and the reduced programs, VirtualBuild obtains the same dynamic dependencies and constructs the same UDG as the one based on the actual build. VirtualBuild is quite efficient in analyzing real-world programs. It finishes the analysis of the project LLVM with 5.67 MLoC in 38.63 minutes, while the state-of-the-art approach has to take over four hours. Meanwhile, it reports the same results as the existing ones, ensuring soundness and completeness. The insight underlying VirtualBuild enables us to extend existing UDG-based dependency error detectors via the virtual build seamlessly, showing its great potential in the efficient maintenance of real-world build systems.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments. The authors are supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001), Natural Science Foundation of China (61902329), Xiamen Youth Innovation Fund (3502Z20206036), the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft, TCL, Tencent, and Huawei. Chengpeng Wang is the corresponding author.

REFERENCES

[1] 2022. ANTLR. https://www.antlr.org/ [Online; accessed 05-May-2022].

[2] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2008. The evolution of the Linux build system. *Electronic Communications of the EASST* (2008). https://doi.org/10.14279/tuj.eceasst.8.115.119

[3] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *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

[4] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for make-based build crashes. In *Proceedings - 30th International Conference on Software Maintenance and Evolution (ICSME).* https://doi.org/10.1109/ICSME.2014.87

[5] Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for build code errors in makefiles. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 600–601. https://doi.org/10.1145/2591062.2591135

[6] Cor Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering* (2017). https://doi.org/10.1007/s10664-017-9510-8

[7] Microsoft Build. 2022. Preprocessor Directives. https://docs.microsoft.com/en-us/cpp/preprocessor/preprocessor-directives?view=msvc-170 [Online; accessed 05-May-2022].

[8] 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 *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 463–474. https://doi.org/10.1145/3395363.3397388

[9] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014. Automated migration of build scripts using dynamic analysis and search-based refactoring. *ACM SIGPLAN Notices* (2014). https://doi.org/10.1145/2714064.2660239

[10] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 599–616. https://doi.org/10.1145/2660193.2660239

[11] Carl A. Gunter. 1996. Abstracting Dependencies between Software Configuration Items. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT1996, San Francisco, California, USA, October 16-18, 1996*, David Garlan (Ed.). ACM, 167–178. https://doi.org/10.1145/239098.239129

[12] Ryan Hardt and Ethan V. Munson. 2013. Ant build maintenance with Formiga. In *Proceedings of the 1st International Workshop on Release Engineering, RELENG 2013, San Francisco, California, USA, May 20, 2013*, Bram Adams, Christian Bird, Foutse Khomh, and Kim Moir (Eds.). IEEE Computer Society, 13–16. https://doi.org/10.1109/RELENG.2013.6607690

[13] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1078–1089. https://doi.org/10.1145/3180155.3180181

[14] Xiang Hu, Haitao Mi, Zujie Wen, Yafang Wang, Yi Su, Jing Zheng, and Gerard de Melo. 2021. R2D2: Recursive Transformer based on Differentiable Tree for Interpretable Hierarchical Language Modeling. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 4897–4908. https://doi.org/10.18653/v1/2021.acl-long.379

[15] International Business Machines Corporation (IBM). 2020. IBM Rational Clearcase. https://www.ibm.com/us-en/marketplace/rational-clearcase [Online; accessed 05-May-2022].

[16] Google Inc. 2022. Bazel - a fast, scalable, multi-language and extensible build system. https://bazel.build/ [Online; accessed 05-May-2022].

[17] Lukás Jendele, Markus Schwenk, Diana Cremarenco, Ivan Janicijevic, and Mikhail Rybalkin. 2019. Efficient Automated Decomposition of Build Targets at Large-Scale. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019.* IEEE, 457–464. https://doi.org/10.1109/ICST.2019.00055

[18] Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 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

[19] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 43–54. https://doi.org/10.1145/3293882.3330578

[20] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. https://doi.org/10.1109/SANER.2018.8330201

[21] Bill McCloskey. 2022. memoize. https://github.com/kgaughan/memoize.py [Online; accessed 05-May-2022].

[22] Eric S Raymond. 2003. *The art of Unix programming*. Addison-Wesley Professional.

[23] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated Localization for Unreproducible Builds. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA, 71–81. https://doi.org/10.1145/3180155.3180224

[24] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. 2019. Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 527–538. https://doi.org/10.1109/ASE.2019.00056

[25] Ryan G. Scott, Omar S. Navarro Leija, Joseph Devietti, and Ryan R. Newton. 2017. Monadic composition for deterministic, parallel batch processing. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 73:1–73:26. https://doi.org/10.1145/3133897

[26] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: A case study (at google). In *Proceedings - International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2568225.2568255

[27] Mike Shal. 2009. Build system rules and algorithms. *Published online (2009). Retrieved July* 18 (2009), 2013. http://gittup.org/tup/build_system_rules_and_algorithms.pdf

[28] 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

[29] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in puppet programs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 26–37. https://doi.org/10.1145/3377811.3380384

[30] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2020. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 169:1–169:28. https://doi.org/10.1145/3428237

[31] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. 2012. SYMake: a build code analysis and refactoring tool for makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 366–369.

[32] Brush Technology. 2022. fabricate. https://github.com/brushtechnology/fabricate [Online; accessed 05-May-2022].

[33] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab S. Mirrokni. 2015. Automated Decomposition of Build Targets. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 123–133. https://doi.org/10.1109/ICSE.2015.34

[34] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-break My Build: Assisting Developers with Build Repair Hints. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (2018), 41–4110.

[35] VestaSys. 2020. Vesta Configuration Management System. http://www.vestasys.org/ [Online; accessed 05-May-2022].

[36] 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

[37] Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. 2014. Build predictor: More accurate missed dependency prediction in build configuration files. In *Proceedings - International Computer Software and Applications Conference (COMPSAC)*. https://doi.org/10.1109/COMPSAC.2014.12