ABC: Accelerated Building of C/C++ Projects

Ying Zhang[†], Yanyan Jiang[†], Chang Xu[‡*], Xiaoxing Ma[‡*] and Ping Yu[‡]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
[†]{zhangying_nju, jiangyy}@outlook.com, [‡]{changxu, xxm, yuping}@nju.edu.cn

***Abstract*—Software building is recurring and time-consuming. Based on the finding that a significant portion of compilations in incremental build is unnecessary, we propose bypath compilation, an efficient build technique that avoids unnecessary recompilation with automated detection of redundant dependencies and unessential changes in source files. The technique is lightweight and transparent to software developers, and can be easily applied to existing build systems. We evaluated our approach on a set of real-world open source projects. The results show that 83% ~ 97% of the recompilations are unnecessary and our approach can accelerate the incremental build up to 44.20%.**

***Index Terms*—Build system; Incremental build; Bypath compilation**

I. Introduction

Broad categories of computer software, ranging from operating systems, compilers, web servers and database management systems to our daily desktop applications, are implemented in C/C++ programming language. The development routine of C/C++ project could be outlined as an *edit-build-test* cycle. Typically, programmer edits a few source files, and then builds the whole project, runs test suite on the executable file generated in the building process for validation and verification. Accelerating the build phase helps to increase the productivity of programmers.

The build process is typically conducted by an automated build system, which transforms source files into executables or library artifacts. The build system follows the *build files*' specification, drives the complier, linker and other tools in appropriate order, and generates intermediate object files and the final output artifacts. *Make* [1] is the de facto build system for C/C++ projects. There are also alternatives, such as *brazel* [2] *Scons* [3], *ninja* [4] and *waf* [5].

One of the most important characteristics of build process is that build is usually *incremental*, that is, only changed source files are recompiled while reusing a large portion of unchanged object files from the previous build. However, even if build systems support incremental build, we found it is still quite common that the build system recompiles more source files than the programmer actually changed, i.e., a notable portion of recompilations is *unnecessary* as they generate the same outputs with their previous compilations. Unnecessary recompilations are mainly caused by *redundant dependencies* and *unessential changes*.

Existing build systems cannot distinguish necessary and unnecessary recompilations. They determine whether an object should be recompiled only by timestamp information (such as *make* and most other build systems) or textual change (such as *Scons*), which are inaccurate and incurs large amount of unnecessary recompilations. These systems have little knowledge about the syntax or semantic of the underlying programming language, leaving a gap between build systems and compilers.

Current research [6]–[10] focused on removing *redundant dependencies* by refactoring. A careful refactorartion reduces redundant dependences, and thus accelerate the build process. However, not all real-world projects are well organized, and refactoring requires substantial error-prone labor work to keep consistency between the build specification and the code base.

We present the *bypath compilation* approach to Accelerate the Build of C/C++ projects and its implementation *ABC*. *ABC* can effectively and efficiently eliminate *redundant dependencies* and *unessential changes* in the code that have no semantic meaning and do not alter the compilation output (e.g., reformatting code and editing comments), speeding up the build process.

We evaluated our approach for its effectiveness and efficiency on a set of real-world open source projects. The results showed that at most 97% of recompilations are unnecessary and incremental build could be accelerated up to 60.79% by our tool.

In summary, we made the following contributions in this paper:

1) We confirmed the existence of unnecessary recompilations in incremental build of real-world C/C++ projects. These unnecessary recompilations are mainly caused by redundant dependencies and unessential changes.
2) We proposed a bypath compilation approach to accelerate the build of C/C++ projects and its implementation *ABC*. Our approach accelerates incremental build by eliminating unnecessary recompilations. Our approach is orthogonal to existing build techniques and is transparent to the build system and the programmer.
3) We evaluated our approach on a set of real-world open source projects. The results showed that our approach is both effective and efficient.

The rest of the paper is structured as follows. We first discuss the background of build system based on an example project in Section II. Then we present our bypath compilation approach in Section III. Section IV is the evaluation and results. We discuss related works in Section V. Finally, we conclude with our thoughts for future work in Section VI.

* Corresponding authors.

II. BACKGROUND

In this section, we introduce the software build procedure. Starting from a motivating example in Section II-A demonstrating how build system works, we illustrate the concept of dependency, essential and unessential changes in Section II-B, II-C and II-D, respectively. Finally, we carry out the observation that cascading dependencies amplifies the effect of unessential changes, which motivates us to design an optimized solution in Section III.

A. Motivating Example

To demonstrate the issue of redundancies in incremental software build and illustrate our fundamental ideas, we present a motivating example of a mini compiler project. The project organization and the corresponding build script (*Makefile*) are shown in Fig 1 (a) and (b), respectively.

The common practice of C/C++ project distinguishes two kinds of source files: (1) code files commonly with a *.c* or *.cpp* extension, and (2) header files commonly with a *.h* or *.hpp* extension. Header files typically only contain declarations of constant symbols, global variables, types and functions, and the actual implementation of functions and definitions of variables are distributed across different code files. If a code file uses items declared in a header file, it must include the header file using the `#include` preprocessing directive, instructing the compiler to find the proper definitions of those items.

A header file is usually included by several code files. In our example, constant definition of error codes and miscellaneous helper functions are declared in *util.h*, which is included by all the three code files.

A code file and all header files it included compose a *compilation unit*. In the preprocessing phase, each `#include` directive in the code file is expanded by corresponding header file's text in place, yielding a temporary intermediate file (commonly with a *.i* extension) of the compilation unit. The intermediate file is then parsed, compiled and optimized into an object file (with *.o* or *.obj* extension commonly). Object files of several compilation units are linked into executable or library files to complete the compilation procedure. We omitted the intermediate files in Fig. 1(a) as they are temporary. Each code files of *main.c*, *code_gen.c* and *parser.c* corresponds to a compilation unit. Each compilation unit is compiled into an object file respectively (*main.o*, *code_gen.o* and *parser.o*). All object files are linked into the final executable file *prog*.

B. How build system works

The project structure in Fig. 1(a) is a directed acyclic graph (DAG). Arrows represent dependencies between DAG nodes. An arrow from node *A* to node *B* indicates that node *B* is needed in prior of generating node *A*. A node in the project DAG is also called a target, except those without outgoing arrows, i.e. source files. A target usually depends on several sub-targets or source files. In *Makefile*, the dependency is expressed in the pattern of `A: B C ...` , as line 1, 3, 5, 7 in Fig. 1(b).

Though C/C++ compilers can resolve dependencies within a compilation unit according to the `#include` directives, they do not have the knowledge of inter-compilation-unit dependencies in the linking phase. For example, Line 1 in Fig. 1(b) links all the three compilation units into the executable file *prog*. This cannot be expressed by the `#include` mechanism. Therefore, most build systems require explicitly intra- and inter- compilation unit dependencies depicted in a build file, e.g. *Makefile* for *make*. Dependencies could be derived manually by programmer or automatically (by extracting `#include` directives using the command `gcc -M foo.c`, for example). The entire dependency graph can be constructed by parsing the build file.

We will take *make* as an example to discuss how the build system works. When *make* performs a *fresh build*, (i.e., building the project from scratch without any previous object files available), it traverses the project DAG in the topologically sorted order, and for each build target executes associated commands.

After the fresh build, the intermediate targets, i.e. object files, are preserved for the next build. Then programmer changes some source files and triggers a successive build. We call this kind of build *incremental build*. *Make* will act much like a fresh build, except that at the source files level, it will check whether a source file is changed by comparing the file's modifying timestamp with its corresponding object file's. If the source file is "older" than its object file, *make* considers this source file not changed since last build, and then reuses its object file previously compiled. Otherwise, *make* considers that this source file is changed, and runs the associated commands to recompile it. The newly generated object file is always newer than any target depending on it, so these targets should also be regenerated, triggering cascading actions. Furthermore, if several targets depend on the same changed source file, all these targets should be regenerated. This is why *make* usually recompiles many source files even if only minor modification to a header file.

In our example project, all the source files are compiled at the fresh build, generating object files *main.o*, *code_gen.o*, *parser.o* and the final executable file *prog*. If *main.c* is changed, its corresponding target *main.o* will be regenerated in the following incremental build, also cause *prog* to be relinked with the new *main.o* and previous *code_gen.o*, *parser.o*. If *util.h* is changed, all the three object files should be regenerated, and *prog* should be relinked.

C. Dependencies in build system

Dependencies play a crucial role in build system. Incorrect dependencies will cause inconsistencies in compilation and linking procedures, and finally lead to incorrect software builds. Even if dependencies are correct, they may be redundant, causing unnecessary build actions. Both manual and automated dependency specification approaches may derive redundant dependencies. Manually maintained dependencies in *Makefile* are surely prone to be redundant. Automatically

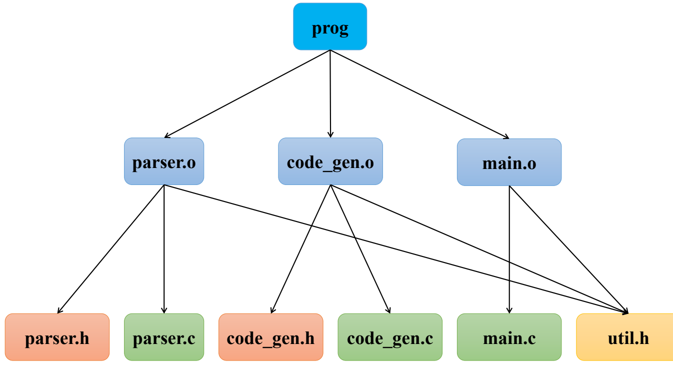

Fig. 1(a). Strcuture of a mini compiler project

```
1. prog: main.o code_gen.o parser.o
2.     gcc -o prog main.o code_gen.o parser.o

3. main.o: main.c util.h
4.     gcc -o main.o -c main.c

5.     code_gen.o: code_gen.c code_gen.h util.h
6.     gcc -o code_gen.o -c code_gen.c

7. parser.o: parser.c parser.h util.h
8.     gcc -o parser.o -c parser.c
```

Fig. 1(b). *Makefile* of the example project

dependencies are derived using the `#include` directives, while `#include` directives are many times redundant. That is because including an unused header file does not cause compilation error, and it is easier for programmers to include more header files than the exactly header files for different compilation units.

We use different sets below to describe redundant dependencies. These definitions are similar to Tichy's work [11]. Given a compilation unit U, it contains a code file C, and header files H_1, H_2, ..., H_n.

Let

$dep(C) = \{H_1, H_2, \ldots, H_n\}$: The set of header files (directly and indirectly) depended by code file C;
$decl(H)$: The set of declarations in header file H, such as constant symbols, global variables, types and functions;
$use(C)$: The set of external declarations used in code file C.

We have

$$use(C) \subseteq decl(\, dep(C)\,) \tag{1}$$

where the right side in Eq. (1) is

$$decl(\, dep(C)\,) = decl(\{H_1, H_2, \ldots, H_n\}) = \bigcup_{i=1}^{n} decl(H_i) \tag{2}$$

In most cases, $use(C)$ is the proper subset of Eq.(2). That is, C does not use all the declarations.

If

$$use(C) \cap decl(H_r) = \varnothing \tag{3}$$

i.e., C does not use any declaration in H_r, then the dependency from C to H_r is redundant. H_r's corresponding `#include` directive should be removed from C.

Dependencies can be optimized to accelerate the project's build process. [6], [8], [12]–[14] reduce dependencies by clustering project components. [7], [9], [10] remove redundant dependencies by analyzing the declaration and usage within code files and header files. These approaches require project refactoring, which is risky for large projects.

D. Essential changes and unessential changes

It is discussed previously that *make* determines whether a source file is changed by comparing its modifying timestamp with its object file's. *Scons*, another build system, takes a different approach. It checks whether a file is changed by comparing its checksum with its previous version's. In this way, *Scons* can get the exactly changed files, at the expense of computing each file's checksum.

Though *Scons* can get the exactly changed files, changes in these files may not have syntactical or semantic meanings to compiler. We identify two kinds of changes in source files here: *essential changes* and *unessential changes*.

A change in a code file is *unessential* if the changed file is compiled into object file that is identical with its unchanged version. Otherwise, the change is *essential*.

Note that this definition relies on the behavior of compiler. We use the compiler as the criterion because determining whether a piece of code and its changed version are semantically equivalent is undecidable.

Unessential changes in a single code file could be:

1) Reformatting code (adding / removing spaces or empty lines);

2) Editing comments;

3) Renaming local variables or parameters;

These patterns listed above make no semantic difference, and thus are unessential. The definition of unessential change for header file is different from code file's. Because a header file may be included in several compilation units, yielding different object files respectively.

A change in a header file is *unessential* to a certain compilation unit if the unit is compiled into an object file that is identical to the one as if the header file is unchanged. Otherwise, the change is *essential* to this unit.

Changes in a header file may be unessential to compilation unit *A* but be essential to compilation unit *B*. In Fig. 2, header file *util.h* is included by all compilation units. The *parser* compilation unit only uses a part of declarations in *util.h* (the region marked with the @ symbol). Suppose that changes in the *util.h* only influence *code_gen.c* (the region marked with the asterisk symbol) and have no effect for *parser.c*. These changes are unessential to the *parser* unit, but are essential to the *code_gen* unit.

In an incremental build, unessential changes trigger recompilations but yield the same object files as the previous build. Furthermore, the targets depending on these newly generated object files are also relinked. All these recompilations and relinkings are unnecessary, dragging the incremental build slow.

Let H be the changed header file, and H' the previous version of it. Define

$$\begin{aligned} \mathit{diff}(H) &= (\,\mathit{decl}(H') \setminus \mathit{decl}(H)\,) \cup (\,\mathit{decl}(H) \setminus \mathit{decl}(H')\,) \\ &= \mathit{add}(H) \cup \mathit{del}(H) \end{aligned} \tag{4}$$

where $\mathit{add}(H) = \mathit{decl}(H') \setminus \mathit{decl}(H)$ contains added declarations in H and $\mathit{del}(H) = \mathit{decl}(H) \setminus \mathit{decl}(H')$ contains deleted declarations. A modified declaration is represented by an old item in $\mathit{del}(H)$ and a new item in $\mathit{add}(H)$. It is obvious that $\mathit{diff}(H)$ is the symmetric difference of $\mathit{decl}(H)$ and $\mathit{decl}(H')$. $\mathit{diff}(H)$ is called *semantic difference*.

For a code file C including H, conditions

$$\mathit{use}(C) \cap \mathit{decl}(H) \neq \varnothing \tag{5}$$

and

$$\mathit{use}(C) \cap \mathit{diff}(H) = \varnothing \tag{6}$$

indicate that the changes in H are unessential to this compilation unit containing code file C. Condition (5) is the negation of condition (3). It constraints that header file H is not redundant to code file C.

Furthermore, dependencies amplify the effect of unessential changes in header files, and redundant dependencies cause even more problems. If H and H' are both redundant to C, then condition (5) is false, nothing declared in H and H' is used in C. Changes in H are meaningless, or say unessential to C. In this case, condition (6) is also true. It indicates that we can use condition (6) to detect unnecessary recompilations triggered either by redundant dependencies or by unessential changes in a unified way.

III. Bypath Compilation

Most existing build systems, such as *make* and *Scons*, have little knowledge about the semantics of source code being built, and therefore cannot distinguish unnecessary recompilations caused by redundant dependencies or unessential changes. In the case that most recompilations are redundant (e.g., adding a function declaration in a header file), existing build systems waste lots of time generating semantically identical binaries. By detecting and eliminating unnecessary recompilations in prior, there is a chance to accelerate the incremental build. This observation motivates us to design the *bypath compilation* approach.

Bypath compilation is a lightweight approach that acts as a wrapper for the compiler, and is transparent to the build system. When incremental build starts, the build system at first detects changed source files as usual, decides files need to be recompiled based on dependencies. Before build system actually compiles the source file, bypath compilation tool intercepts the command-line arguments and checks whether the recompilation is unnecessary. If so, we just skip the recompilation and reuse its previous object file. Otherwise, we pass the command to compiler and execute the recompilation.

Fig. 3 shows the flow graph of bypath compilation. The left part denotes a normal compilation. The height of the bar indicates consumed time. The right part illustrates the workflow of our bypath compilation approach. Bypath compilation first checks whether the designated compilation task is unnecessary by a fast source code parsing. According to the checking result, the execution is divided into two paths. The true path indicates an unnecessary compilation, and thus nothing needs to be done. This fast path only costs the essentiality checking time. The false path, on the other hand, indicates that compilation is necessary. In this case, the compiler should be invoked to generate the correct binary, and this slow path consumes slightly more time than the normal compilation.

Let T_f be the time of unnecessary recompilation checking, T_s be the time of recompilation, then the slow path consumes $T_f + T_s$. Let p be the proportion of unnecessary recompilation, also the approximated probability of execution following the fast path.

The average time of bypath compilation T_{avg} is

$$T_{avg} = p \cdot T_f + (1-p) \cdot (T_f + T_s) = T_f + (1-p) \cdot T_s \tag{7}$$

The acceleration rate R_{acc} is

$$R_{acc} = \frac{T_s - T_{avg}}{T_s} = p - \frac{T_f}{T_s} \tag{8}$$

We could conclude from Eq. (8) that acceleration depends on (1) the proportion of unnecessary recompilation p ($0 \leq p \leq 1$) and is expected to near 1, meaning a large proportion of unnecessary recompilation; (2) the ratio $\gamma = T_f / T_s$ ($0 \leq \gamma < 1$) and is expected to be near 0, reflecting that the fast path is significantly faster than the slow path.

In extreme cases that compilations are almost necessary (i.e., $p \approx 0$), our bypath compilation has a negative acceleration ratio, exhibiting slowdown of the build process. We expect those exceptional cases are minor, and bypath compilation approach performs only slightly slower than normal compilation. The performance of our approach depends on the proportion and characteristic of dependencies and unessential changes, may vary for projects and incremental builds.

Nevertheless, for real-world projects, we expect the fast paths would occur much more frequently than slow paths, i.e. the bypath should actually be the main path, making the average time shorter than normal compilation. The evaluation in Section IV confirms our expectation that for our six real-world subjects p varies from 83% to 97%.

According to Eq. (8), the time of checking unnecessary recompilation T_f is critical. If we can achieve negligible checking time (i.e., $\gamma \approx 0$), acceleration is guaranteed. If T_f is twice faster than a normal compilation (i.e., $\gamma = 0.5$), we could get R_{acc} about 33% ~ 47%, which is a significant speedup.

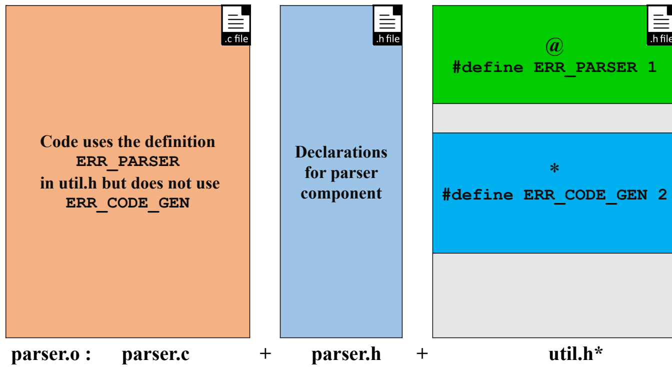

Fig. 2. Changes in a compilation unit

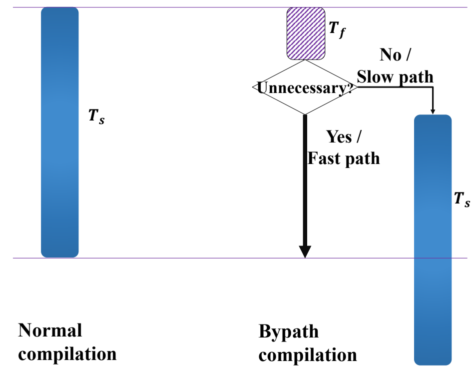

Fig. 3. Execution flow of normal and bypath compilation

Using the semantic differencing technique in condition (6) to detect unnecessary recompilations is a straightforward way. The sets $dep(C)$, $use(C)$, $decl(H)$ and $diff(H)$ could be collected by parsing corresponding source files (to compute $diff(H)$, we need its previous version H', which could be acquired from a version control system). Unnecessary recompilations are those who satisfy $use(C) \cap diff(H) = \varnothing$.

However, existing semantic differencing techniques [15], [16] do not put efficiency as their primary goal, which is critical in our scenario (remind that we are expecting T_f to be minimized). We find that a checking tool has to perform syntax and semantics work as a compiler does. Therefore, we can use a compiler to preprocess the code, and conduct a simple compilation. The procedure of fast compilation essentially identifies unnecessary changes such as add comments or modifying headers. As soon as the checking compiler is faster than the actual building compiler, the build procedure can be accelerated. Specifically, if we are using compiler with optimization flags for build process, a faster compiler could be the same compiler without optimization flags. We could also use a different compiler. For example, *clang* generally compiles code faster than *gcc* if optimization options are turned off. If *gcc* is the building compiler, *clang* could be used as the checking compiler.

IV. Implementation and Evaluation

We present the evaluation of bypath compilation in this section. The prototype implementation is briefly introduced in Section IV-A, followed by two research questions on effectiveness and efficiency in Section IV-B. Correspondingly designed experiments on a set of real-world open source projects and their evaluation results are shown in Section IV-C and IV-D, respectively.

A. Implementation

We implemented a prototype tool called *ABC*, adopting our bypath compilation technique. It uses the *clang* 3.4 with no optimization flag (`-O0`) as the fast compiler to check unnecessary recompilations. *ABC* is implemented as a compiler wrapper. Taking *make* and *clang* for example, *ABC* replaces the default compiler by using the command `make CC="abc clang"`. *ABC* is compatible with different build systems and compilers, as long as the build system supports the `CC` argument or equivalent mechanism to alter the compiler, and the compiler is compatible with standard C/C++ language.

B. Research Questions

We organize evaluation of bypath compilation around the following research questions.

RQ1 (Effectiveness): How many unnecessary recompilations could be detected and eliminated by our approach?

RQ2 (Efficiency): How much acceleration could our approach achieve?

C. Experiment Setup

We evaluated *ABC* on a set of real-world open source C/C++ projects. These projects all use standard GNU *make* as their build system except Samba which uses *waf* [5]. We used the *clang* 3.4 to compile these projects. These projects vary in application domain and sizes, making them representative. Project profiles are listed in Table I.

To evaluate the effectiveness and efficiency of bypath compilation, we apply incremental build for successive commits in the repository. We selected the latest version of each project as the last version, then counted back 100 versions, getting the first version. These commits reflect the recent development activities over the project. Selected commits can be as large as major feature releases, or as small as minor bug fixes. The evaluation operated as follows. Firstly, we reset the project to the first version, and make a fresh build. Secondly, we checkout the project to the next version, changing contents of modified source files. Then we made incremental build with build tools (ordinary *make* and *ABC*), measuring the build time respectively using the Linux *time* utility. This procedure is repeated until we have reached the last version.

To answer RQ1, we used an implementation of *ABC* with log function enabled. It gathers the following information along with the *make* run: (1) command-line arguments intercepted from make. The commands contain the source file names and compilation arguments; (2) whether the code files were changed and whether the object files generated by fast compilation were changed.

To answer RQ2, we switched off the log function of *ABC*, and measured the total incremental build time with and without

TABLE I. PROFILES OF SUBJECTS

Subject	Domain	LoC[a]	Branch	Version (From - To)
cpython	Programming Language	427.8k	3.5	96442 - 96738
ffmpeg	Multimedia Codec	841.5k	master	05cc8c - 32c8d8
llvm	Compiler Infrastructure	1023.0k	master	b82800 - 3bd8f1
nginx	Web Server	102.8k	stable1.2	4804 - 5208
ruby	Programming Language	337.2k	trunk	e4fa17 - 8430f9
samba	File Server	1340.4k	master	408c96 - e1a87d

TABLE II. COUNTS AND TIME OF OVERALL RECOMPILATIONS

Subject	Count[b] (#)			Overall Build Time[c] (Second)		
	All	Unn.	%	*clang*	*ABC*	R_{acc}
cpython	2354	2277	96.73%	220.60	123.09	44.20%
ffmpeg	12072	11299	93.60%	1520.58	1531.62	-0.73%
llvm	7108	6091	85.69%	5625.15	5189.19	7.75%
nginx	1538	1420	92.33%	64.27	47.63	25.88%
ruby	2193	1814	82.72%	470.50	416.67	11.44%
samba	8126	7899	97.21%	1748.64	1652.63	5.49%

a. The LoC is approximate as projects evolving continuously.
b. All : Overall recompilation; Unn. : Unnecessary recompilation;
c. *clang*: Build using *clang* compiler without *ABC*; *ABC* : Build using *ABC*; R_{acc} : Acceleration rate.

ABC for each version of each subject three times, computing the average time as the results.

The evaluations were conducted on a Dell desktop machine with an Intel Core i7-4470 3.4GHz CPU and 8GB memory running Ubuntu Linux 14.04 64 bits.

D. Results

RQ1: Build statistics are shown in Table II. Columns 2 and 3 denote the total amount of recompilations and unnecessary ones, respectively, while Column 4 shows the percentage of unnecessary recompilations. A remarkable percentage of 83% ~ 97% of the recompilations are unnecessary.

Let us take the version 96443 of cpython project as a typical case to explain the data. Three files (*Include/ patchlevel.h*, *Misc/NEWS* and *README*) were changed in version 96443, where the latter two are not C/C++ source code.

The values of definitions "`#define PY_RELEASE_SERIAL`" and "`#define PY_VERSION`" were changed in the file *patchlevel.h*, and only the file *Python/getversion.c* included *patchlevel.h* directly. However, *patchlevel.h* is also included by *Include/Python.h*, which is the cpython project's common header file and is included by 221 code files out of totally 352 code files. Thus, *patchlevel.h* is depended by all these code files indirectly. It is obvious that most of the dependencies are redundant. The changes in *patchlevel.h* are also unessential to most of these code files, but they triggered all the 221 code files to be recompiled.

A closer examination showed that only four recompilations are actually necessary, which are *Python/getversion.c*, *Modules/getbuildinfo.c*, *Modules/main.c* and *Python/sysmodule.c*.

RQ2: The Columns 5 ~ 7 of Table II demonstrate the overall incremental build time for all the 100 versions of each subject project.

Fig. 4 (a) ~ (f) are the incremental build time of each version with *ABC* and their acceleration compared to those using *clang* without *ABC*. Positive accelerations are laid on top of *ABC*'s bars in these figures, while negative ones are laid on the bottom of *ABC*'s bars. We could observe from these figures that *ABC* significantly accelerated the incremental build for most of these real-world projects except ffmpeg and llvm. *ABC* performed best for cpython and achieved maximum 60.79% acceleration rate.

A more careful study shows that incremental builds can be roughly categorized into one of the following three kinds.

1) There were builds that changed files such as documentation or resource files other than source files. In these cases, there was no recompilation, but the build system still consumed few seconds to check the whole project. We call these cases *empty runs*. The build time was nearly zero and almost the same no matter with or without *ABC* for empty run.
2) In some cases, a few source files were changed, and these changes were essential. Recompilations were limited in these files and were not amplified by redundant dependencies. *ABC* did not yield neither significant acceleration nor slowdown in these cases. These cases correspond to the short bars in the figures.
3) In other cases, versions did change a few source files, however, changed files are transitively dependent by many source files. This scenario typically occurs when a header is changed due to API update, and many of the project's code files were unnecessarily recompiled. *ABC* works best in these cases, corresponding to the long bars in the figures.

Finally, it is also worth noting that *ABC* got negative acceleration, i.e. were slower than normal compilations for some versions of ffmpeg and llvm, even if such slow-downs were not significant and acceptable. We analyzed these cases, finding that the link phase dominated these build processes, and the overhead of slow paths slightly degenerates the performance of *ABC*.

V. RELATED WORK

The compilation procedure can be optimized in various ways: parallel build, distributed build, caching, to name but a few. We briefly summarize these techniques in this section. It is worth noting that most of these techniques are orthogonal to our *ABC* approach and thus can be seamlessly integrated with our approach.

Existing build systems were briefly discussed in Section II. *Make*, *Scons* and most other build systems [2]–[5] detect file changes by comparing timestamps or checksums at file granularity, and therefore cannot eliminate unnecessary recompilations caused by redundant dependencies and unessential

Fig. 4. Acceleration of *ABC*. x-axis denotes incremental builds while y-axis denotes time (in seconds).

changes. For instance, *Scons* could detect that a file's checksum is identical to its previous version, thus stops cascading actions depending on this object file in time. A similar tool is *ccache* [17] that uses the source file's checksum as hashcode to look up cached object file, and addresses duplicated compilations in a distributed development environment.

Smart recompilation is a fine-grained technique that can avoid unnecessary recompilations by compiling only semantically non-equivalent snippets in a program. Our work has the same insight and goal with smart recompilation. As discussed in Section III, semantic differencing is complex and difficult to implement. Our approach uses a fast compiler to approximate the effect of semantic difference, and the notions of *use* and *decl* sets are from Tichy's work [11]. Morgenstern's master thesis [18] presented an inconsistency management system (IMS) implementing smart recompilation for C projects. IMS is a set of tools requiring programmer's interaction thus cannot be used automatically. Adams et. al. [19] evaluated the smart recompilation technique's efficiency using code lines as the metric other than real consumed time. [20] is an implementation of smart compilation for Java projects. Its evaluation showed that it is even less efficient than *make*.

Binkley et al. [15] presented the first semantic differencing tool for the C programming language. Its evaluation showed that this tool consumes considerable time. [16] proposed a technique to specify meaningful changes for Java programs. This technique works as a kind of DSL, and thus is flexible to specify new pattern of meaningful changes. [21] researched the non-essential changes in Java projects' version histories and their impact on change-based analyses such as code clone detection. The definition of non-essential changes is slightly different from our unessential changes but shares the same core insight. It reported up to 15.5% of non-essential changes.

There is also research work aimed at reducing redundant dependencies via structural refactorations [6]–[10], [12]–[14]. Refactoring, though substantially reduces redundant dependencies, would be risky for large-scale projects, and requires large amount of human efforts. Our approach, on the other hand, is capable of eliminating the effect of redundant dependencies at build time. Precompilation [22], [23] is another technique alleviating the dependency problem. It uses precompiled

header files instead of recompiling them for each compilation unit for every build. The header files should be stable and commonly used in the project. Google's empirical experience of dependency problem is discussed in [24].

Software build issues are also studied in company with the software process, and build is always an indispensable phase of development cycle. McIntosh et.al. [25] analyzed the coupling between the source code and build system evolvements. They suggested project managers account the build maintenance effort. [26], [27] presented experiences of build system migration. Our approach is portable for different build systems in theory, and evaluations on this feature is considered as a possible future direction.

Finally, we observed that the linking phase had significant impact on those incremental builds only involving few changes. If the project is statically linked, any changed object file will cause relinking of all the object files. The relinking reuses no information in previous builds, and regenerates everything from scratch. Though quite many efforts are done in optimizing the linking procedure (e.g., the new ELF linker *gold* [28]), design and implementation of an effective and efficient incremental linker is still an open problem.

VI. Conclusion

In summary of the paper, we found that redundant dependencies and unessential changes lead to unnecessary recompilations in incremental builds, substantially slowing down the build process. We thus developed the bypath compilation technique, and implemented the prototype tool *ABC*. *ABC* exploits compiler's ability to detect redundant dependencies and unessential changes, elides unnecessary recompilations and therefore accelerates the incremental build. We evaluated our approach on a set of real-world open source projects. The results showed that our approach detected that 83% ~ 97% of the recompilations are unnecessary and accelerated the incremental build up to 44.20%.

Finally, we also found that if we can slice the program via a fine-grained semantic differencing, only a small fraction of the code is required to be recompiled. We plan to develop such tool based on the *clang* compiler front-end.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472177, 91318301, 61321491) of China.

References

[1] S. I. Feldman, "Make - a program for maintaining computer programs," *Softw.: Practice and experience (SPE)*, vol. 9, no. 4, pp. 255–265, 1979.
[2] bazel. http://bazel.io/.
[3] Scons. http://scons.org/.
[4] ninja. http://martine.github.io/ninja/.
[5] waf. http://waf.io/.
[6] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos, "Improving the build architecture of legacy c/c++ software systems," in *Fundamental Approaches to Softw. Eng.*, ser. Lecture Notes in Computer Science (LNCS), M. Cerioli, Ed. Springer, 2005, vol. 3442, pp. 96–110.
[7] D. Spinellis, "Optimizing header file include directives," *J. of Softw. Maintenance and Evolution: Research and Practice*, vol. 21, no. 4, pp. 233–251, 2009.
[8] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *Proc. of the 37th Int. Conf. on Softw. Eng. - Volume 1 (ICSE)*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 123–133.
[9] K.-P. Vo and Y.-F. Chen, "Incl: A tool to analyze include files," in *USENIX Summer*, 1992, pp. 199–208.
[10] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, "Removing false code dependencies to speedup software build processes," in *Proc. of the 2003 Conf. of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 2003, pp. 343–352.
[11] W. F. Tichy, "Smart recompilation," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 3, pp. 273–291, Jun. 1986.
[12] C. Chambers, J. Dean, and D. Grove, "A framework for selective recompilation in the presence of complex intermodule dependencies," in *Proc. of the 17th Int. Conf. on Softw. Eng. (ICSE)*. New York, NY, USA: ACM, 1995, pp. 221–230.
[13] M. de Jonge, "Build-level components," *Softw. Eng., IEEE Trans. on (TSE)*, vol. 31, no. 7, pp. 588–600, July 2005.
[14] A. Telea and L. Voinea, "A tool for optimizing the build performance of large software code bases," in *Softw. Maintenance and ReEng., 2008. CSMR 2008. 12th European Conf. on*. IEEE, April 2008, pp. 323–325.
[15] D. Binkley, R. Capellini, L. R. Raszewski, and C. Smith, "An implementation of and experiment with semantic differencing," in *Softw. Maintenance, 2001. Proc. IEEE Int. Conf. on (ICSM)*. IEEE, 2001, pp. 82–91.
[16] Y. Yu, T. T. Tun, and B. Nuseibeh, "Specifying and detecting meaningful changes in programs," in *Automated Softw. Eng. (ASE), 2011 26th IEEE/ACM Int. Conf. on*. IEEE Computer Society, Nov 2011, pp. 273–282.
[17] ccache. http://ccache.samba.org/.
[18] H. M. Morgenstern, "An inconsistency management system," Master's thesis, Columbia University, 1987.
[19] R. Adams, W. Tichy, and A. Weinert, "The cost of selective recompilation and environment processing," *ACM Trans. on Softw. Eng. and Methodol. (TOSEM)*, vol. 3, no. 1, pp. 3–28, Jan. 1994.
[20] M. Dmitriev, "Language-specific make technology for the java programming language," in *Proc. of the 17th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, vol. 37, no. 11. New York, NY, USA: ACM, 2002, pp. 373–385.
[21] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proc. of the 33rd Int. Conf. on Softw. Eng. (ICSE)*. ACM, 2011, pp. 351–360.
[22] T. Krishnaswamy, "Automatic precompiled headers: Speeding up c++ application build times," in *Proc. of the 1st Conf. on Industrial Experiences with Systems Softw. (WIESS) - Vol. 1*. Berkeley, CA, USA: USENIX Association, 2000, pp. 57–66.
[23] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos, "Reducing build time through precompilations for evolving large software," in *Softw. Maintenance, 2005. ICSM'05. Proc. of the 21st IEEE Int. Conf. on*. IEEE, Sept 2005, pp. 59–68.
[24] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at google," in *Managing Technical Debt (MTD), 2012 Third Int. Workshop on*, June 2012, pp. 1–6.
[25] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proc. of the 33rd Int. Conf. on Softw. Eng. (ICSE)*. ACM, 2011, pp. 141–150.
[26] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits, "Automated migration of build scripts using dynamic analysis and search-based refactoring," in *Proc. of the 2014 ACM Int. Conf. on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA: ACM, 2014, pp. 599–616.
[27] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, and B. Adams, "An empirical study of build system migrations in practice: Case studies on kde and the linux kernel," in *Softw. Maintenance (ICSM), 2012 28th IEEE Int. Conf. on*. IEEE, Sept 2012, pp. 160–169.
[28] I. L. Taylor, "A new elf linker," in *GCC Developers' Summit*, vol. 2008, 2008, pp. 129–136.