IEICE TRANS. INF. & SYST., VOL.E104–D, NO.1 JANUARY 2021

PAPER

Native Build System for Unity Builds with Sophisticated Bundle Strategies

Takafumi KUBOTA[†a)], *Nonmember and* **Kenji KONO**[†], *Member*

SUMMARY Build systems are essential tools for developing large software projects. Traditionally, build systems have been designed for high incremental-build performance. However, the longer build times of recent large C++ projects have imposed a requirement on build systems: i.e., *unity builds*. Unity builds are a build technique for speeding up sequential compilation of many source files by bundling multiple source files into one. Unity builds lead to a significant reduction in build time through removal of redundant parsing of shared header files. However, unity builds have a negative effect on incremental builds because each compiler task gets larger. Our previous study reported existing unity builds overlook many better bundle configurations that improve unity-build performance without increasing the incremental-build time. Motivated by the problem, we present a novel build system for better performance in unity builds. Our build system aims to achieve competitive unity-build performance in full builds with mitigating the negative effect on incremental builds. To accomplish this goal, our build system uses *sophisticated bundle strategies* developed on the basis of hints extracted from the preprocessed code of each source file. Thanks to the strategies, our build system finds better bundle configurations that improve both of the full-build performance and the incremental-build performance in unity builds. For example, in comparison with the state-of-the-art unity builds of WebKit, our build system improves build performance by 9% in full builds, by 39% in incremental builds, and by 23% in continuous builds that include both types of the builds.

key words: *build systems, unity builds, C++*

1. Introduction

Build systems [1]–[10] are considered essential to developing large software projects. They orchestrate thousands of order-dependent commands including compiling source files, linking object files, and testing output binaries, to ensure that target programs are built correctly. Since developers execute a build system whenever they modify the source code, the performance of build systems directly affects the efficiency of developing cycles.

Build systems provide *incremental builds* [5] for high efficiency when rebuilding software projects with small updates. Incremental builds mean small partial rebuilds, where few source files need to be recompiled. For efficiency of incremental builds, the build systems calculate the minimum amount of commands necessary to generate a target binary that reflects the changes made to the source code. For example, when one source file is to be updated, the build system recompiles only that source file; it does not recompile any other source files.

However, since the code size of software projects has increased, the full build times and even incremental build times have become longer and problematic [11]–[14]. *Full builds* mean building the whole of target projects, in which many source files are compiled. For example, WebKit [15], which is the web browser engine used by Safari, takes 56 minutes to full-build with 18 threads on our Dell Power Edge R430 server. Chromium [16], an open-source browser project that forms the basis for the Chrome web browser, takes two and a half hours to build. Furthermore, incremental build times have become longer as well because of the many recompilations needed for updating the source and header files. According to the logs of the build bot of WebKit [17] collected from March 17 and August 3 in 2018, build tasks taking more than 15 minutes were submitted 164 times on 61 out of 75 days.

To accelerate builds, large C++ projects have started to apply a new build technique, called *unity builds* [18]–[24]. In unity builds, multiple source files are bundled into one *unity* file like in Fig. 1. Then, the unity file is passed to the compiler instead of the bundled source files. As a result, the compiler can eliminate the redundant processing of the header files commonly included in bundled source files. In so doing, the build times of large C++ projects can be dramatically reduced; for example, full build times can be decreased by 53% in WebKit and by 63% in Chromium.

However, unity builds negatively impact incremental builds [14]. They enlarge each compiler task by bundling multiple source files. For example, in unity builds, unity files are compiler tasks that are independently passed to the compiler as its input. However, since a unity file `#include`s multiple source files, not only updated source files but also non-updated source files are recompiled in incremental builds with unity builds. For example, when `API/JSBase.cpp` in Fig. 1 is updated, seven other source files have to be recompiled as well. These unnecessary

```
#include "API/JSBase.cpp"
#include "API/JSCTestRunnerUtils.cpp"
#include "API/JSCallbackConstructor.cpp"
#include "API/JSCallbackFunction.cpp"
#include "API/JSCallbackObject.cpp"
#include "API/JSClassRef.cpp"
#include "API/JSContextRef.cpp"
#include "API/JSHeapFinalizerPrivate.cpp"
```

Fig. 1 Example of unity files in WebKit.

Manuscript received May 8, 2020.
Manuscript revised August 5, 2020.
Manuscript publicized October 15, 2020.

[†]The authors are with the Department of Information and Computer Science, Keio University, Yokohama-shi, 223–8522 Japan.
a) E-mail: takafumi@sslab.ics.keio.ac.jp

DOI: 10.1587/transinf.2020EDP7105

recompilations result in large slowdowns in incremental builds.

In existing unity builds [18]–[24], they simply bundle a fixed number of source files in alphabetical order, as shown in Fig. 1. However, this simple strategy ignores characteristics of the source files to be bundled (e.g., which header files are `#included` in the source files) so that undesired source files are likely to be bundled together. In our previous study [14], we investigated the unity builds of the WebKit and revealed some better bundle strategies that improve both the full-build performance and the incremental-build performance in unity builds.

Based on the results, this paper presents a novel build system that supports unity builds using the sophisticated bundle strategies. It automatically collects hints for finding better bundle configurations during the build. Then, it bundles source files based on the sophisticated bundle strategies by using the collected hints. As a result, our build system achieves better build performance in large C/C++ projects including WebKit, LLVM [25], and the Mesa 3D library [26], compared with existing unity builds such as WebKit-unity builds [18], CMake-unity builds [27], and Meson-unity builds [20].

This paper makes the following contributions.

1. It proposes a new build system for unity builds using sophisticated bundle strategies.
2. It evaluates the build system on three large C/C++ projects, showing that it outperforms existing unity builds.

The rest of this paper is organized as follows. Section 2 provides background information. Section 3 describes the bundle strategies used in our build system. Section 4 outlines the design of our build system. Section 5 demonstrates the build performance. Section 6 covers related work. Section 7 concludes this paper.

2. Background

2.1 Build Systems

Build systems [1], [2], [4]–[7], [9], [10], [28]–[31] automate the execution of order-dependent tasks to generate target programs. A build system takes source files and build rules as inputs, constructs a dependency graph from the rules, and appropriately executes build commands.

A dependency graph consists of *nodes* and *edges*. Each node represents files and internal states of build systems. Edges show input and output dependencies among nodes. Build commands are set to edges. They include not only compiling and linking but also other commands such as changing directories and archiving files. When a build system tries to process a build command for an edge, it first confirms all input nodes are ready, then executes the command.

Figure 2 shows an example of the dependency graph for generating an executable file of `main.exe`. Here, normal rectangles show nodes of the files (i.e., header, source, object, and executable files). Double-lined rectangles indicate nodes of the internal states in the build system for issuing build commands. Then, arrows are the edges to specify dependencies and build commands.

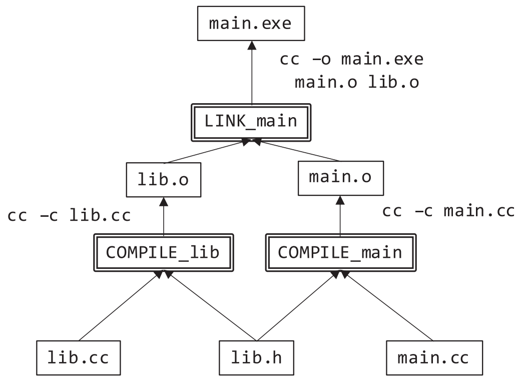


Fig. 2 Example of the dependency graph in build systems.

To understand how the build system deals with this dependency graph, let us discuss a case, when a user runs the build system specifying `main.exe` as the desired output without any previous compile result. The build system first finds the linking edge from `LINK_main` as the required task. However, it figures out the two dependent object files (`main.o` and `lib.o`) are not ready for this task. So, it iterates back in the dependency graph to find the edges for creating these object files. Then, it eventually finds the two compiling edges from `COMPILE_main` and `COMPILE_lib`. It confirms that all dependent input files exist and starts to issue the build commands at first. As a result, it will first build `lib.o` and `main.o`, in any order since these tasks are independent, and then build `main.exe`.

Build systems are essential tools for developing large software systems. Whenever developers modify the source code, they have to re-execute a build system to synchronize their modification to the target programs. This build method means the performance of build systems directly affects the productivity of typical developing cycles. As a result, the build systems must have high performance and efficiency [32].

2.2 Incremental Builds

Traditionally, build systems have been designed for high efficiency when rebuilding target programs with small changes. *Incremental builds* [5] are a build technique to avoid unnecessary build tasks when rebuilding programs. In incremental builds, build systems track dirty source files and calculate the minimum amount of build tasks from the dependency graph. This build method reduces the build time when a developer updates a small number of source files, because most of the build tasks can be skipped in incremental builds.

Many of the existing tools and studies aim at improving the performance of incremental builds. For example, Ninja [7] is a state-of-the-art build system that focuses on very fast incremental builds by forbidding users to express complex build rules. Compile caching tools such as CCache [33] and cHash [34] enhance incremental build performance by avoiding redundant compilations via textual hashing or AST hashing. Precompiled headers [35]–[37] are useful for saving unnecessary header recompilations.

2.3 Long Build Times of Large C++ Projects

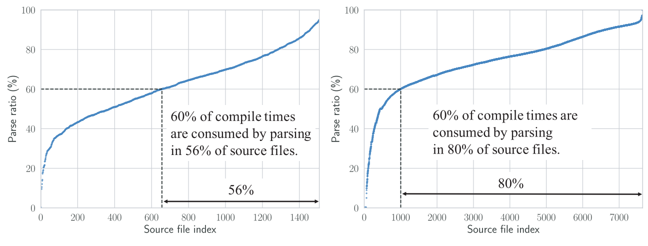
However, the build times for large C++ projects have become unacceptably long [11]–[14]. System software has continuously and rapidly evolved, and code sizes have reached thousands of source files and millions of lines of code (LOC). This evolution has increased the build time for full and incremental builds in C ++ projects. For instance, a full build of WebKit [15] takes 56 minutes with 18 threads on our DELL Power Edge R430 server. Chromium [16] takes two and a half hours to build on the same server. Such long build times are problematic for individual developers in open source projects who compile and build their projects on a standard laptop [38]. The build times of incremental builds are also problematic because source-code modifications are no longer small in large open-source projects. For example, a recent study [14] has pointed out that build tasks taking more than 15 minutes were submitted to build bots of WebKit almost every day from March 17 and August 3 in 2018. In such build tasks, 1,030 source files were recompiled on average.

To confirm that the problem is not limited to WebKit, we analyzed the logs of build bots of LLVM [25], [39], which is an open-source compiler framework written in C++, from March 9 to August 1 in 2018. The number of build tasks submitted during the period was 1,146. Build tasks consuming more than 15 minutes happened 227 times on 74 out of 81 days when the build bot was active. Thus, long build times for large C++ projects are commonplace.

Note that the build bots of both projects mentioned above use CCache. CCache is a compile caching tool, which saves and reuses previous compiled results indexed by the hash of the preprocessed code. However, the benefit of the compile caching tool is limited because many developers change different source and header files at a time in large open-source projects. Such updates easily change the preprocessed code so that CCache cannot reuse previous compiled results. As a result, many recompilations happen. Precompiled headers do not work well because header files are often updated in such long builds. For example, according to the build bot logs of both projects, the average number of modified header files was 11 in LLVM and seven in WebKit. Updating even one header file makes the precompiled headers obsolete so that they have to be recompiled.

2.4 Unity Builds

The problem of long build times highlights a new requirement for build systems that need to speed up sequential compilation of many source files.



(a) LLVM (b) WebKit

Fig. 3 How much time does the compiler spend parsing? Here, the x-axis shows source-file indexes for each project. The y-axis shows parse ratios for each source file. The parse ratio indicates the occupancy of parsing time during the compile time. The times are measured by using the timer report functionality of GCC (-ftime-report). The x-axis is sorted in ascending order of the parse ratio.

One of the main reasons why build times are so long in large C++ projects is redundant header processing [11]–[14]. In large C++ projects, much of the compilation time is spent parsing. For example, Fig. 3 a and Fig. 3 b show the distribution of times in which the compiler performs parsing. According to these figures, parsing occupies a majority of the compilation time (60%>) in many source files (56% in LLVM and 80% in WebKit). This high ratio stems from shared header files that are included in multiple source files. The compiler repeatedly reads the shared headers, parses them, and instantiates the same template bodies across different source files.

To eliminate redundant header processing in multiple source files, developers have started to use a new build technique, called *unity builds*; there are several project-specific names for this technique such as *unified builds* in WebKit [18]. Unity builds are designed for achieving high build performance in large C++ projects. In a unity build, a bunch of source files are `#include`'d in a single unity file (as shown in Fig. 1) which is then compiled. In current unity builds, the bundle strategy, which is how to select source files to be bundled together, is simply bundling a fixed number of source files in alphabetical order. The limit of *bundle size*, which is the number of `#include`'d source files, is project-specific value. However, it is often set to eight based on developers' experience [11], [18], [27]. In unity builds, *bundle configurations*, which are rules of source files bundled together, are decided by the bundle strategy and the maximum bundle size.

Unity builds accelerate both compiling and linking. Compilers parse and compile commonly included headers among the bundled source files only once and make fewer instantiations of the same templates among the source files. Linkers also receive the benefits of unity builds by reducing redundant processing in linkers. For example, linkers do not have to remove $N-1$ copies of the same weak symbol defined in a shared header included in N source files (N is the bundle size). As a result, unity builds have dramatically reduced build times of large C++ projects. For instance, they have been shown to reduce the full build times in WebKit by 53% (56 minutes → 28 minutes) and in Chromium by 66% (2 hours 34 minutes → 52 minutes).

However, unity builds have the disadvantage of increasing incremental build times. Since unity builds enlarge each compiler task by bundling multiple source files, an incremental build of one updated source file involves $N - 1$ non-updated source files bundled into the same unity file. Developers have recognized this downside, but they underestimate its negative effect. The WebKit project reports that the worst slowdown is 20% in a typical scenario where one or two files are touched [40]. However, our previous study exposed a slowdown in incremental builds of one source file of 479% (19 seconds → 110 seconds) in the worst case [14].

3. Bundle Strategies in Proposed Build System

3.1 Problem in Naive Bundle Strategy

While unity builds are considered as useful for reducing the build times of C++ projects and are being used in real projects [18], [19] and build systems [20]–[24], all of their approaches are inadequate. In particular, the current bundle strategy is simple and ignores the characteristics of the bundled source files. This means that bundle configurations that may achieve higher build performance and cause less incremental build slowdown are likely to be overlooked. To address this problem, we use sophisticated bundle strategies introduced by our previous study [14].

3.2 Sophisticated Bundle Strategy

Before we describe the bundle algorithm, let us describe our bundle strategies. In our previous study, we investigated the unity builds of WebKit and suggested some bundle strategies that improve the build performance of the unity builds [14]. However, in the previous study, we just investigated the potentiality of how much sophisticated bundle strategies improve unity-build performance and showed an empirical study only in WebKit. We did not integrate the strategies into build systems, did not discuss whether they are able to be integrated or not, did not show the overhead for applying them, and did not evaluate them on other projects. In this paper, our build system uses two of the four strategies proposed in that study. Here, we introduce the employed strategies.

Strategy: bundling source files with high header-file similarity. Since unity builds cut off the redundant processing of shared header files, they work well when most of the included headers overlap among the bundled source files. To quantify the overlap of the included header files, the Jaccard index [41] is used to gauge the similarity of two sets. It is defined as the size of the intersection divided by the size of the union of the two sets; here, each set indicates the header files included in a source file.

Here, unlike existing unity builds, we bundle source files regardless of the directories they are in. WebKit bundles source files only in the same directory. The reasons why WebKit developers made this decision are not explained, but it is speculated as follows. Developers often place related source files in the same directory whose name reflects the usage. For example, according to Fig. 1, source files for implementing APIs are in the same directory. As a result, source files often include common header files where basic classes and function prototypes are defined. So, the header-file similarity seems to be high among source files in the same directory. However, we found many source files in different directories have high header-file similarities. Based on the result, our build system tries to bundle source files in different directories if the header-file similarity is high.

Strategy: bundling source files with similar compile time. The compile time of each bundled source file is sensitive to the unity-build performance. The imbalance in the compile times of bundled source files causes significant performance degradation in incremental builds. For example, when we bundle source A whose compile time is one second with source B whose compile time is 10 seconds, the build time of unity builds cannot be under 10 seconds because the compile time of source B becomes a bottleneck. In such a situation, the slowdown of the incremental build of source A will be dreadful. The minimal overhead is estimated to be 9 seconds (10x slowdowns), which is definitely unacceptable for any developer updating source A.

However, this strategy is hard to be applied straightforwardly, because accurate compile times are not available before the build. We initially tried to use a machine learning technique to estimate compile times from the preprocessed code. However, we eventually gave up this approach because most of the techniques use information obtained by parsing source files [42]. Parsing the source files takes up a significant portion of the compilation time and adds overhead, as described in Sect. 2. Thus, we decided to use the word count of '`;`' to estimate compile times. Figure 4 shows the relationship between the number of '`;`'s and the compile times in LLVM and WebKit. We performed polynomial regression on these data and calculated an approximate expression that our build system uses to estimate the compile times. Note that although this estimation is simple and not accurate completely, it is still effective in increasing the unity-build performance as described in Sect. 5.

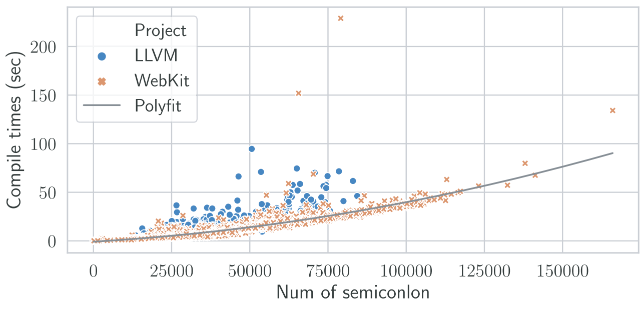


Fig. 4 Compile time estimation

4. Design of Our Build System

In this section, we describe the design of our build system, which uses sophisticated bundle strategies to achieve better unity-build performance.

4.1 Meta Build System vs. Native Build System

When we embarked on the design of our build system, we made an essential design choice, as follows.

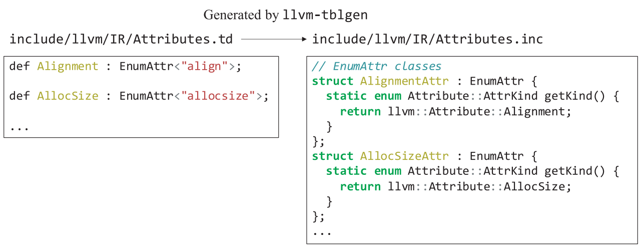
Current build systems can be classified into two types. Native build systems (e.g., Make [5], Ninja [7], llbuild [4], and so on) automatically issue build commands to generate target programs. On the other hand, meta build systems (e.g., CMake [3], Meson [6], Waf [24], and so on) do not build directly, but instead, generate build rules to be used by a native build system.

Unity builds are usually implemented as a module of a meta build system or a configuration of an integrated development environment (IDE) [18]–[24]. Despite this, we decided we should take the native build system approach, for the following reason. To apply sophisticated bundle strategies, our build system has to extract hints for bundling from the preprocessed code of each source file. However, it is difficult to complete the preprocessing of all source files when meta build systems are run. This is because not all of the header files are prepared before the build.

In large projects, developers sometimes use simple descriptions for defining similar data structures. The actual header files are generated from the descriptions during the build by using specific tools (e.g., `llvm-tblgen` in LLVM) or script languages (e.g., `ruby` in WebKit), as shown in Fig. 5. Thus, during the execution of the meta build system, preprocessing fails for some source files because of the missing header files.

To deal with this problem, our build system needs a scheduling feature to run the preprocessor at the appropriate scheduling point during the build. To clarify this requirement, we describe the build behavior using the dependency graph depicted in Fig. 6. Here, our build system tries to bundle `Attributes.cpp` that include `Attributes.inc`. `Attributes.inc` is a header file dynamically generated from `Attributes.td` by `llvm-tblgen`. In short, our build system runs the preprocessor for `Attributes.cpp` just before it processes the compiling edge for `Attributes.o` (the highlighted arrow in the figure). This is because, when it starts to deal with the edge, it has already confirmed that all inputs (i.e., `Attributes.cpp` and `Attributes.inc`) are ready. Thus, the preprocessing does not fail due to the lack of the dependent header file.

Note that our build system does not run the preprocessor just after the creation of `Attributes.inc`, because the number of dynamically generated header files is not always one. Waiting until the processing of the compilation edge begins is a promising way to confirm all required input nodes are ready.



```
def Alignment : EnumAttr<"align">;

def AllocSize : EnumAttr<"allocsize">;

...
```

```
// EnumAttr classes
struct AlignmentAttr : EnumAttr {
  static enum Attribute::AttrKind getKind() {
    return llvm::Attribute::Alignment;
  }
};
struct AllocSizeAttr : EnumAttr {
  static enum Attribute::AttrKind getKind() {
    return llvm::Attribute::AllocSize;
  }
};
...
```

Fig. 5 Example of the header files dynamicallygenerated by `llvm-tblgen`

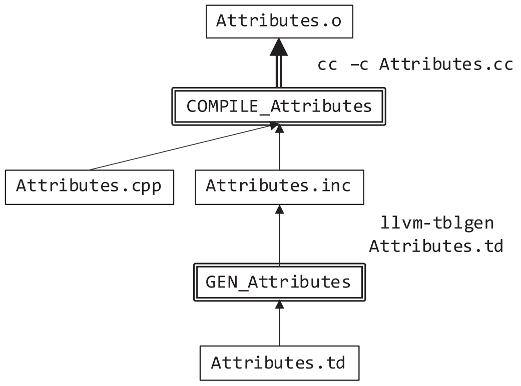


Fig. 6 Dependency graph including the dynamically generated header file

4.2 Overview

An overview of our build system is shown in Fig. 7. The design and implementation are based on a state-of-the-art native build system called Ninja [7]. To clarify our contributions, they are highlighted in the figure with double-lined boxes.

First, our build system takes source files and build manifests as its inputs and generates the dependency graph, just as Ninja does. Next, it analyzes the dependency graph for tracking updates to the file organization. It tracks additions and removals of source files by comparing the analysis result with the previous one. When it detects any change that affects the bundle configurations, it invalidates the affected configurations and reconfigures them in subsequent bundling steps.

After the dependency graph analysis, our build system starts its bundling steps. The bundling steps consist of two parts: ahead-of-time (AOT) and just-in-time (JIT). In AOT bundling, it bundles source files whose dependent header files exist before the build. It runs the preprocessor for the source files and gets hints for bundling from the preprocessed code. After bundling the source files, it modifies the compile commands for compiling the unity files. Then, it starts to schedule the build tasks.

When our build system starts the compilation of the source file whose dependent header files are not ready before the build, it interrupts the processing and performs JIT bundling. At this moment, it is confirmed that all header

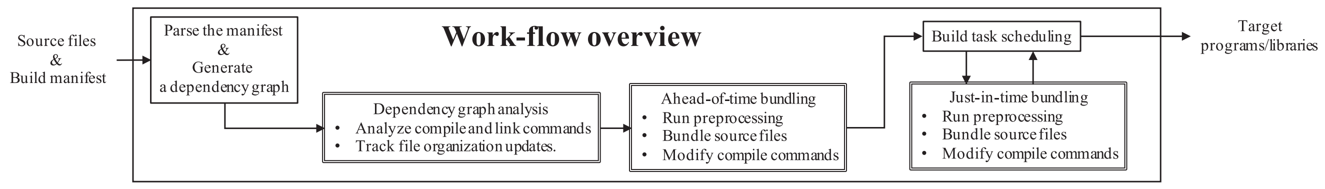


Fig. 7 Overview of the work-flow

files for this compilation are ready so that it can execute the preprocessor and bundle the source file. After it bundles the source files and modifies the compile commands, it resumes the build schedule. Once it finishes a unity build, it saves the bundle configurations for reuse in future builds.

4.3 Dependency Graph Analysis

During software development, source files are added and removed. To reflect such updates to the file organization, our build system keeps track of the updates and modifies the previously created bundle configurations. It accomplishes these tasks by tracking changes in the dependency graph.

Our build system analyzes the compiling edges and linking edges in the dependency graph and determines sets of source files that can be bundled together. In this paper, we call such source-file sets *bundle-sets*. For example, source files that are compiled with different flags or that are linked to different target programs cannot be bundled together. When a developer adds a new source file, it analyzes the corresponding edges for the new source file and registers the source file to an appropriate bundle-set. As a result, it can bundle the source file in subsequent unity builds. It does not bundle a new source file immediately because changing the bundle configurations will cause additional recompilations of other source files or unity files. It only reconfigures the bundle configurations when all of the source files in the bundle-set are to be compiled.

When a source file is removed, our build system excludes the source file from the bundle-set where the source file is. It also checks whether an existing unity file includes the removed source file. If it does, the source file is removed from the unity file and from the bundle configuration. It also makes sure the unity file is recompiled to avoid inconsistencies in the output programs.

4.4 Bundling Source Files

Now, let us describe how our build system bundles source files. As mentioned above, it performs two types of bundling: ahead-of-time (AOT) and just-in-time (JIT). After the dependency graph analysis, it gets information on the source file set that can be bundled together (i.e., bundle-set). When all of the source files in a bundle-set are ready for preprocessing before the build, it bundles the source files by using AOT bundling. On the other hand, if the set contains source files that include dynamically generated header files, it performs JIT bundling on the set. In the JIT bundling, our build system waits until all source files in the set are ready for preprocessing; then it bundles the source files. In particular, our build system waits for all of the inputs for compiling edges in the dependency graph are ready. Although AOT and JIT bundling have different timings for bundling the source files, their bundlings consist of three common steps.

First, our build system runs the preprocessor for the source files in order to extract two pieces of information. To compute the header file similarity, it collects the dependency of header files by using a general compiler feature (e.g., `-M`). To estimate the compile time for each source file, it counts the number of '`;`'s and calculates the compile times by using the approximate expression as we described in Sect. 3.2. We show the overhead of this additional preprocessing in Sect. 5.1. This overhead typically happens only at the first unity builds because our build system saves generated bundle configurations and reuses them in subsequent builds. Bundle-hints are managed by timestamps of their creation and will be updated only when reconfiguring bundle configurations.

Once our build system finishes gathering bundle-hints of all the source files that can be bundled together, it starts to bundle the source files. Note that bundling source files into unity files with the highest header-file similarity and with the highest compile-time similarity is a knapsack problem which is NP-hard. So we decided to use a simplified algorithm to avoid large overhead in computing which source files are bundled together.

Algorithm 1 shows an algorithm overview. First, to bundle the source files with high header-file similarity, it calculates the similarities among the source files by analyzing the file dependency and saves the result as a sorted-list (line 1-2). Then, it bundles each two source files in order of those with the highest header-file similarity (line 3-15). To keep the header-file similarity in unity files high, it stops bundling source files if the similarity falls below 90% (line 4). To bundle source files with similar compile times, it does not bundle source files whose compile times have a 50% difference, even the header-file similarity is high (line 5). These thresholds are set according to the results of the previous study [14].

There are three ways to bundle the two source files. First, if both of these source files have not been bundled yet, our build system bundles them into a new unity file (line 6-9). Second, if one of the source files has not been bundled and the other one has already been bundled in a unity file, our build system checks whether the unbundled source file can be added to the unity file. In particular, it checks that the bundle size does not exceed the maximum bundle size

Algorithm 1: Bundling source files

```
Data: $S$: source file set, $H$: list of header-file similarity data
      ($src_i$, $src_l$, $similarity$)
Result: $U$: Unity files
1  $H \leftarrow$ calculate header-file similarities among $S$;
2  Sort $H$ by the similarity;
3  foreach Entry ($src_i$, $src_l$, $similarity$) $\in H$ do
4      if $similarity < 0.9$ then break ;
5      if Compile times have a huge difference then continue ;
6      if Both of the source files are not bundled then
7          Allocate a new unity file $u$;
8          Bundle $src_i$ and $src_l$ to $u$;
9          Add $u \rightarrow U$;
10     else if Only one of the source files is bundled then
11         Bundle the unbundled source file, if can;
12     else if $src_i$ and $src_l$ are bundled in different files then
13         $u_i, u_l \leftarrow$ get unity files of $src_i$ and $src_l$;
14         Merge the two unity files, if can;
15 end
```

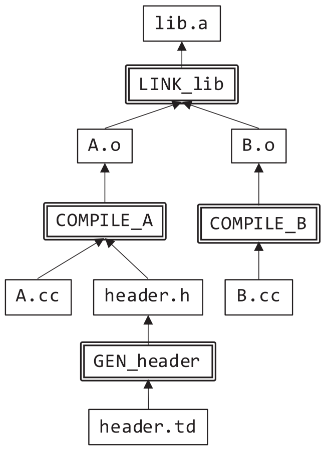


Fig. 8 Compilation of `B.cc` will be finished before `A.cc` and `B.cc` are bundled

and whether the compile times of all bundled source files are similar. If both checks pass, it adds the source file to the unity file (line 10-11). Here, the similarity of header files is not checked. This is because, even if the header-file similarity is unchecked, the header-file similarity of unity files is kept high. Our build system finishes the bundling when if the similarity falls below 90% (line 4). As a result, even if unchecked source-file pairs are in unity files, the header-file similarity is kept high; the average similarity of header files for each unity file is never below 87% in our evaluation. Third, when two source files are bundled in different unity files, our build system merges these two unity files after checking the maximum bundle size and the differences in compile times (line 12-14).

After bundling the source files, our build system modifies compile commands to compile unity files instead of bundled source files. In JIT bundling, it sometimes finishes compiling source files before bundling them because it waits for dependent header files to be generated for other source files. For example, Fig. 8 shows two order-independent compilations of `A.cc` and `B.cc`. Based on the dependency graph analysis result, our build system tries to bundle the two source files. Since `A.cc` includes `header.h` that is a dynamically generated header file, our build system has to wait for bundling until the header file is created.

However, the compilation of `B.cc` does not depend on the header creation. As a result, the compilation of `B.cc` may be done before `A.cc` and `B.cc` are bundled. In such cases, our build system just creates a bundle configuration and does not modify the compile commands to avoid double compilations, which cause symbol redefinition errors.

5. Evaluation

We evaluate our build system with existing unity builds of CMake, Meson, and WebKit, focusing on the build performance in continuous builds, the incremental-build performance, and the full-build performance. This evaluation shows that:

- Our unity builds have better build performance in continuous builds (Sect. 5.1).
- Our build system decreases the overhead in incremental builds (Sect. 5.2).
- Our build system achieves competitive full-build performance (Sect. 5.3).

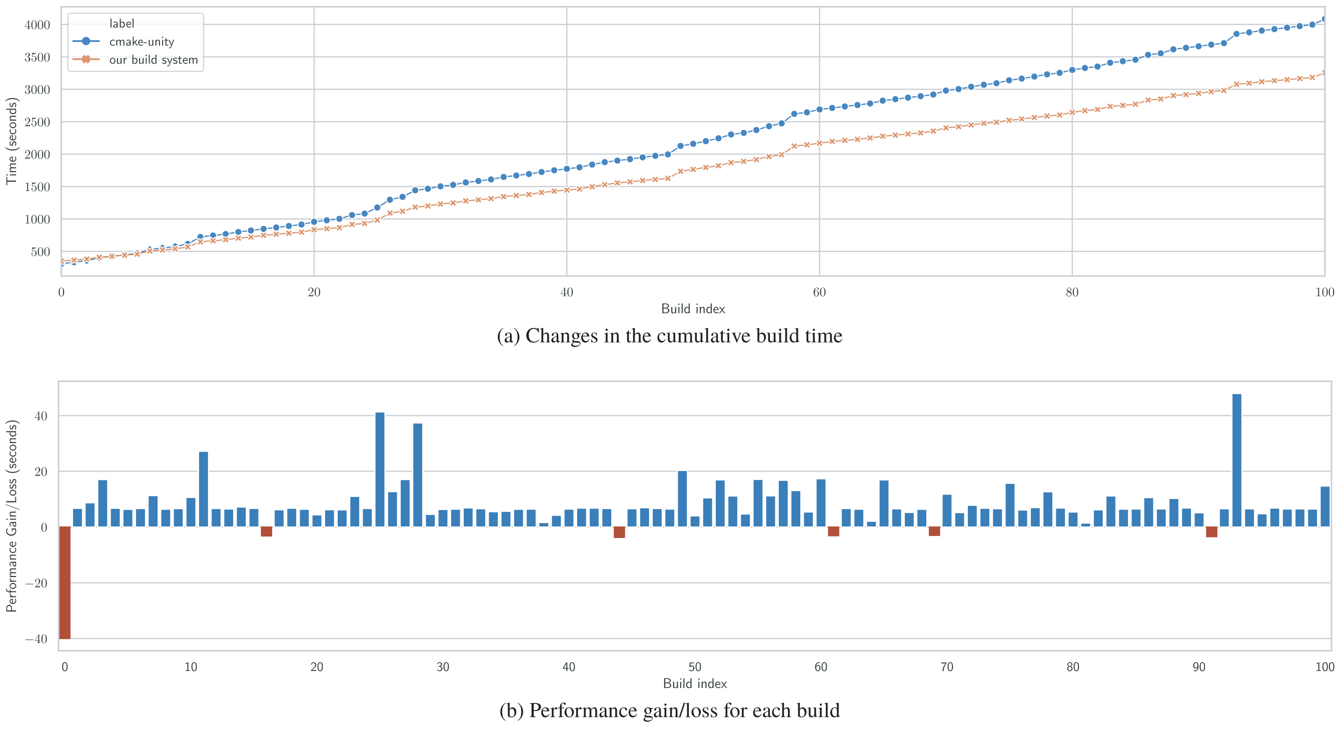
We perform the evaluation on our DELL Power Edge R430 server that has a Fedora 29 server installed. The server consists of 8-core 2.1 GHz Xeon E5-2620V4 processor, 128 GB of RAM, and a 1TB SATA HDD disk. All source files are resident in the disk.

We choose three real C/C++ projects for evaluating our build system, including LLVM (git-commit-id: `ca1e713fdd4`), WebKit (git-commit-id: `ec4eb02a9e2`), and Mesa 3D library (v18.3.6) [26]. In the evaluations, we use GCC v8.3.1 and LLVM/Clang v9.0.0 as compilers but only show the results of the GCC, because the results of the Clang are similar to those of the GCC. The linker is GNU ld v2.31.1.

We evaluate the unity builds of WebKit [18], CMake-unity builds [27], and Meson-unity builds [20] for comparison. The maximum bundle size of each unity file is set to eight, which is the default value of the unity builds of WebKit and CMake-unity builds. Note that, there is no limit on the bundle size in Meson-unity builds. However, this unlimited bundle size causes a terrible overhead in incremental builds (Sect. 5.2).

5.1 Build Performance in Continuous Builds

To evaluate how well unity builds of our build system perform in continuous builds, we analyze the build times of sequential git commits in LLVM and WebKit. The build performance in continuous builds is important in using continuous integration and automated test tools, such as buildbot [17], [39]. In this evaluation, we use CCache because it



(a) Changes in the cumulative build time

(b) Performance gain/loss for each build

Fig. 9 Build performance during 101 builds of real git commits: LLVM

is enabled in the build bots of the two projects.

First, we discuss the results of LLVM. Here, we use the git commit `ca1e713fdd4` (Feb. 6 2019) as the first full build and use the continuous 100 commits from `e9f465a6a80` to `2e390e6fde4` (Feb. 6-8 2019) for this evaluation. 27,552 LOC in 306 files were updated during the commits.

Figure 9 a shows changes in the total build time during the 101 builds. Figure 9 b depicts the performance gain/loss of our build system for each build, compared with the existing unity builds of the CMake. According to the figures, our build system has better build performance in many builds. This results in that our build system reduces the total build time by 20% and saves 13 minutes in total.

For example, our build system achieves the highest performance gain in the 92nd build. This is because our build system reduces unnecessary recompilations in unity builds. In unity builds, the number of source files to be recompiled is increased because of bundling multiple source files into one unity file. Our build system avoids bundling unrelated source files together by checking the similarities in included header files and compile times. This enables our build system to efficiently recompile the smaller number of source files in partial rebuilds. As a result, our build system excludes 176 unnecessary recompilations for unrelated source files (404 files → 228 files).

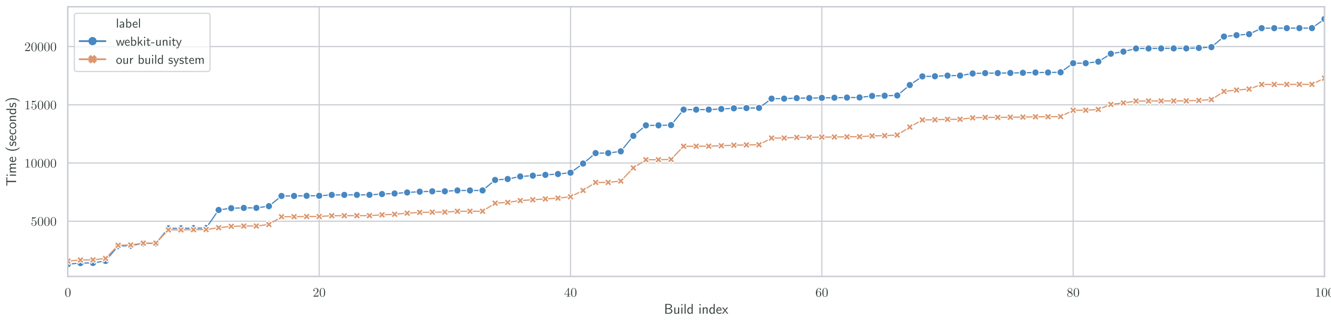
The reason why our build system takes a longer time in the first full-build than the CMake-unity build is the overhead in bundling source files. Since our build system requires the additional preprocessing to extract hints for bundling source files, the first full-build time includes the overhead. The overhead of the additional preprocessing is 10% (33s) in LLVM. However, this overhead is only included in the first full-build because our build system reuses the bundle configuration for subsequent builds.

Figure 10 shows the results of WebKit. Here, the git commit `ec4eb02a9e2` (Feb. 8 2019) is used for the first full-build and the continuous 100 commits `85f3eaeb98f` to `f7a11be17a9` (Feb. 6-12 2019) are used for this evaluation. 15,515 LOC in 624 files were changed during the git commits.

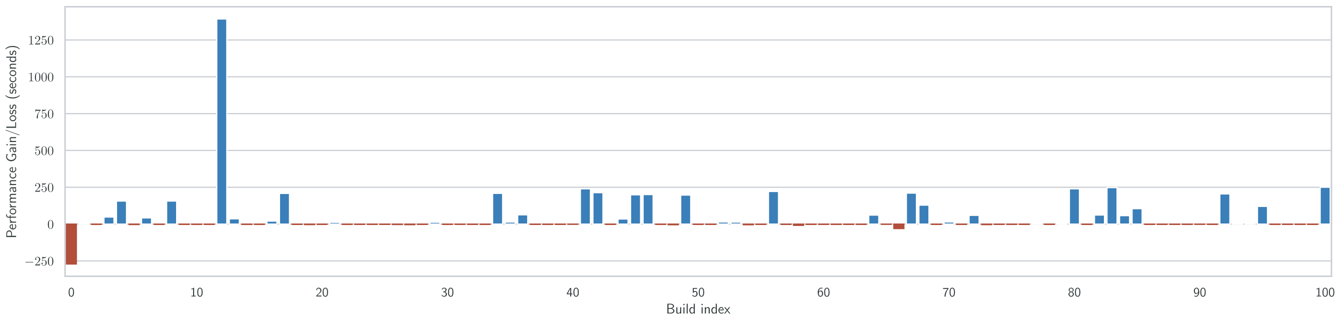
The figure shows similar results to LLVM. Our build system achieves large performance gains in many builds and improves the total build time by 23% and saves one hour and 24 minutes in total. The overhead of the additional preprocessing in the first full-build is 33% (6m 29s).

In particular, our build system achieves the highest performance gain in the 12th build. This is because the git-commit (`404c727821a`) corresponding to this build includes a file addition and an update on a configuration file of WebKit-unity builds for the added files. As a result, many unity files are regenerated and recompiled. However, our build system avoids such bursts of recompilation by lazily updating the bundle configuration when source files are added. It only updates the configuration only when all of the source files in a bundle-set are to be compiled, as described in Sect. 4.3. Thus, our build system avoids the update on bundle configuration due to the file addition and reduces unnecessary recompilations.

In summary, unity builds with the sophisticated bundle strategies and our native build system approach succeed in reducing the total build time of the continuous builds, which include both large partial rebuilds (like full builds) and small



(a) Changes in the cumulative build time



(b) Performance gain/loss for each build

Fig. 10 Build performance during 101 builds of real git commits: WebKit

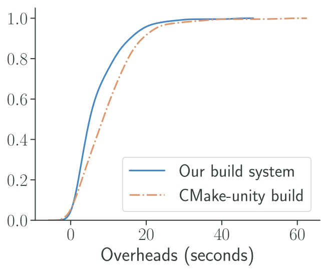
partial rebuilds (like incremental builds). The unity-build performance in continuous builds is important when unity builds are employed in continuous integration, automated testing, and daily development.

5.2 Incremental-Build Performance

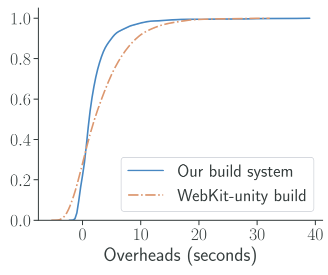
Now let us discuss the incremental-build performance of our unity builds. To evaluate the overhead of incremental builds caused by unity builds, we measured the slowdowns of rebuilding the projects when one source file was updated. The reason why we used this methodology is that it emphasizes the disadvantage of unity builds. The slowdowns are evaluated by the absolute difference of the build times rather than the ratio. This is because the ratio does not represent the developer's response to the build times. For example, doubling a slowdown of 0.5 second to 1 second is more acceptable than a 50% slowdown of 10 seconds to 15 seconds. Here, we did not use CCache in this experiment. This is to facilitate the recompilation of each source file. In other words, we only update source-file timestamps to trigger recompilations.

Figure 11 shows the overall results: CDFs of the incremental-build overheads. Table 1 shows notable numbers in the overall results. In summary, our unity builds have better incremental-build performance in terms of the average and the 90th percentile slowdowns for these projects. This is because our build system utilizes the similarity of compile times in its bundle strategy and mitigates the negative effect on incremental builds in unity builds.

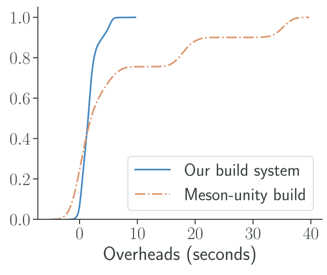
For example, the average slowdown is reduced from



(a) LLVM



(b) WebKit



(c) Mesa

Fig. 11 CDF of incremental-build overheads (seconds) caused by unity builds. The x-axis shows the overheads in seconds for all figures. The y-axis shows the cumulative probability.

Table 1 Notable results on incremental build overheads (seconds). The results of existing unity builds are shown in parentheses.

	mean	median	90th percentile	the worst case
WebKit	1.91 (3.11)	1.09 (1.97)	4.98 (9.17)	38.0 (29.64)
LLVM	6.97 (9.62)	4.83 (8.49)	15.77 (18.60)	44.25 (57.14)
Mesa	1.69 (6.96)	1.56 (1.40)	3.93 (19.85)	8.80 (35.71)

```
--- a/Source/JavaScriptCore/Sources.txt
+++ b/Source/JavaScriptCore/Sources.txt
@@ -392,9 +392,10 @@ dfg/DFGSSAConversionPhase.cpp
 ...
 dfg/DFGSnippetParams.cpp
-dfg/DFGSpeculativeJIT.cpp
-dfg/DFGSpeculativeJIT32_64.cpp
-dfg/DFGSpeculativeJIT64.cpp
+// These files take a long time to compile so we do
↪  them individually.
+dfg/DFGSpeculativeJIT.cpp @no-unify
+dfg/DFGSpeculativeJIT32_64.cpp @no-unify
+dfg/DFGSpeculativeJIT64.cpp @no-unify
 dfg/DFGStackLayoutPhase.cpp
 ...
```

Fig. 12 Patch (git-commit-id: `418f6e0edf4`) in order not to bundle source files taking a long compilation time in WebKit.

3.11 seconds to 1.91 seconds (by 39%) in WebKit. The 90th percentile slowdown is also improved by 46% (from 9.17 seconds to 4.98 seconds). In the case of the worst overhead, our unity builds improve the overheads in the LLVM and the Mesa, except for the WebKit. The reason why existing unity builds of WebKit achieves the smaller overhead is that the developer of WebKit excludes the source files that cause unacceptable overhead in incremental builds, by applying a patch as shown in Fig. 12. Here, the `@no-unify` is an attribute for CMake to exclude a source file from unity builds.

According to Fig. 11 c, the Meson-unity build significantly worsens the overhead of incremental builds for many source files. This is because there is no limit in the maximum bundle size in the Meson-unity build. For example, 108 source files are bundled together in the Meson-unity build. As a result, the unlimited bundle size causes a terrible overhead in incremental builds so that setting the maximum bundle size is necessary for reducing the incremental-build overhead in unity builds.

Note that some of the incremental-build slowdowns are below zero. This means that for some source files, unity builds outperforms non-unity builds even in incremental builds. This is because unity builds can accelerate link processing because the input size of link operations is reduced. As a result, unity builds constantly improves the I/O performance of the link processing even in incremental builds. For example, when the static library libWebCoreGTK.a is created in WebKit, 3,667 object files are linked in non-unity builds. By comparison, only 580 object files are processed in unity builds of our build system.

5.3 Full-Build Performance

Here, we shall focus on the full-build performance to evaluate the impact of our bundle strategies. In this evaluation, we exclude the overhead of bundling source files (such as computing header similarities, estimating compile times, choosing source files that should be bundled together, and creating unity files) are excluded. We showed the pure impact of using sophisticated bundle strategies in terms of source files bundled together. Table 2 describes the overall results. In short, the unity build of our build system achieves competitive build performance, compared with existing unity builds. For example, the full build time is decreased by 9% (from 21m 51s to 19m 53s), compared with the unity builds of WebKit. In LLVM and Mesa 3D library, the build times are similar to existing unity builds. The reason why only the full-build performance for WebKit using our method is better than using the existing one is that bundle configurations become much different. Since WebKit contains more source files than the other two projects, the configuration of which source files are bundled together becomes more different when changing bundle strategies. As a result, the benefit of using sophisticated bundle strategies becomes larger.

Table 2 Full-build performance

	Our build system	Existing Unity builds
WebKit	19m 53s	21m 51s
LLVM	5m 16s	5m 9s
Mesa	2m 9s	2m 11s

Table 3 The number of unity files and bundled source files in our unity builds. The results of existing unity builds are shown in parentheses.

	# of unity files	# of bundled source files	Avg. bundle size
WebKit	748 (660)	4,959 (5,027)	6.6 (7.6)
LLVM	234 (349)	1,284 (1,735)	5.5 (5.0)
Mesa	142 (81)	818 (905)	5.8 (11.0)

To explore the impacts of using our bundle strategies, we examine the number of unity files and the number of bundled source files as shown in Table 3. According to the table, our unity builds do not increase the number of bundled source files, compared with all of the existing unity builds. However, the full-build performance of our unity builds is better or similar to the existing unity builds as shown in Table 2. In general, if the number of bundled source files is decreased, the full-build performance is degraded because fewer source files benefit from unity builds. However, our unity builds provide the competitive full-build performance thanks to our bundle strategies, even though fewer source files are bundled than existing unity builds.

6. Related Work

Zapcc: Zapcc [43] is a caching C++ compiler based on LLVM/Clang, designed to perform faster compilations. Zapcc has an in-memory compilation cache in the client-server architecture for remembering all of the compilation information (e.g., ASTs, IRs, and so on) between runs. As a result, Zapcc can reduce redundant header processing, as unity builds do.

Our build system with GCC outperforms Ninja with Zapcc in building LLVM. Note that we only discuss the result of building LLVM because Zapcc (git-commit-id: `09cb4f07d`) cannot build WebKit, as it lacks support for some compilation flags. The full build time when using Zapcc is 6 minutes 49 seconds; this compares with 5 minutes 49 seconds for our build system. Zapcc is slower because its clients randomly send compile requests to free servers. As a result, the compiling servers often fail to reuse compile caches, so that they clear the compile caches and restart the compiling tasks. For example, the full build of LLVM entails clearing caches 52 times.

C++ Modules: The module system for C++ [44] is designed to deal with the serious degradation of compile-time scalability incurred by header file inclusion. In C++ modules, a binary representation of the corresponding header is imported instead of copying and parsing the header code. Since the module is only compiled once, importing the module into a translation unit is a constant-time operation. To reduce the cost of template specialization, developers can pre-compute commonly used instantiations by exporting explicit instantiations in module definitions.

C++ modules and unity builds share certain motivations and goals. Both approaches try to reduce redundant compilation tasks caused by the header code. However, unity builds could be still effective even after C++ modules become standard. For example, when multiple source files import the same module, there are redundant module reads and deserializations. By bundling such source files into one file as unity builds, the compiling throughput is improved for the same reason that unity builds reduces the redundant header inclusion and parsing. In terms of template instantiation, unity builds could be still effective in C++ modules. To completely exclude the redundant template instantiation by C++ modules, developers have to explicitly instantiate all possibly used template bodies in module definitions [44]. This method is not practical and requires developers not to instantiate templates implicitly in source files. Unity builds can reduce the redundant implicit instantiations in each unity file.

Cross-Module Optimizations: Cross-module optimization (CMO) [45]–[52] is an effective way to improve runtime performance. Since unity builds accelerate the link processing, they also have a positive effect on CMO. For example, it decreases the first link-time optimization (LTO) time of LLVM is decreased from one hour 12 minutes to one hour six minutes. We plan to evaluate the incremental LTO times when using unity builds in future.

7. Conclusion

Unity builds have been identified as a useful technique to improve build times in large C++ projects. However, the downsides of unity builds have been underestimated and overlooked. This paper has presented a novel build system focusing on the trade-off between incremental builds and unity builds. We evaluated our proposal on three large projects: LLVM, WebKit, and the Mesa 3D library. We showed that it achieves better build performance in continuous builds. It also achieves the competitive full-build performance in unity builds and mitigates the slowdown caused by unity builds in incremental builds. We hope that our design and evaluations will serve as a basis of discussion on future build systems, compilers, and module systems that cooperatively generate efficient compiler tasks.

Acknowledgments

This work was supported in part by Japan Science and Technology Agency (JST CREST JPMJCR19F3).

References

[1] “Apache Ant - Welcome.” https://ant.apache.org/.
[2] “Buck: A fast build tool.” https://buckbuild.com/.
[3] “Cmake.” https://cmake.org/.
[4] “A low-level build system, used by Xcode and the Swift Package Manager.” https://github.com/apple/swift-llbuild.
[5] S.I. Feldman, “Make — A Program for Maintaining Computer Programs,” Software: Practice and Experience, 1979. https://www.gnu.org/software/make/.
[6] “The Meson Build system.” https://mesonbuild.com/.
[7] “Ninja, a small build system with a focus on speed.” https://ninja-build.org/.
[8] S. Erdweg, M. Lichter, and M. Weiel, “A sound and optimal incremental build system with dynamic dependencies,” Proc. 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015.
[9] “SCons: A software construction tool - SCons.” https://scons.org/.
[10] “Shake build system.” https://shakebuild.com/.
[11] M. Catanzaro, “On Compiling WebKit (now twice as fast!).” https://blogs.gnome.org/mcatanzaro/2018/02/17/on-compiling-webkit-now-twice-as-fast/, 2 2017.
[12] Jussi, “Automatically finding slow headers in C++ projects.” http://nibblestew.blogspot.com/2018/02/automatically-finding-slow-headers-in-c.html, 2 2018.
[13] S. McIntosh, B. Adams, M. Nagappan, and A.E. Hassan, “Identifying and understanding header file hotspots in C/C++ build processes,” Automated Software Engineering, 2005.
[14] T. Kubota, Y. Suzuki, and K. Kubota, “To Unify or Not to Unify: A Case Study on Unified Builds (in WebKit),” Proc. 28th International Conference on Compiler Construction (CC), 2019.
[15] “Webkit.” https://webkit.org/.
[16] “The chromium projects.” http://www.chromium.org/.
[17] “Webkit build central.” https://build.webkit.org/.
[18] K. Miller, “[webkit-dev] unified source builds: A new rule for static variables,” 2017. https://lists.webkit.org/pipermail/webkit-dev/2017-August/029465.html.
[19] “Jumbo / unity builds.” https://chromium.googlesource.com/chromium/src/+/lkgr/docs/jumbo.md.
[20] “Unity builds in meson.” http://mesonbuild.com/Unity-builds.html.
[21] “cotire.” https://github.com/sakra/cotire.
[22] “FASTBuild - Function Reference - Unity.” http://www.fastbuild.org/docs/functions/unity.html.
[23] O. Arkhipova, “Support for Unity (Jumbo) Files in Visual Studio 2017 15.8 (Experimental).” https://devblogs.microsoft.com/cppblog/support-for-unity-jumbo-files-in-visual-studio-2017-15-8-experimental/.
[24] “waf.” https://gitlab.com/ita1024/waf/tree/master.
[25] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” Proc. 2004 International Symposium on Code Generation and Optimization (CGO), Palo Alto, California, 2004.
[26] “The mesa 3d graphics library.” https://www.mesa3d.org/.
[27] “Unity (Jumbo) build (!3611) · Merge Requests · CMake / CMake.” https://gitlab.kitware.com/cmake/cmake/merge_requests/3611.
[28] “Bazel - a fast, scalable, multi-language and extensible build system.” https://bazel.build.
[29] “Dune - A composable build system.” https://github.com/ocaml/dune.

[30] R. de Levie, Advanced Excel for Scientific Data Analysis, Oxford University Press, 2004.

[31] "gittup/tup: Tup is a file-based build system." https://github.com/gittup/tup.

[32] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems a la carte," Proc. 23rd International Conference on Functional Programming (ICFP), 2018.

[33] "ccache — compiler cache." https://ccache.samba.org/.

[34] C. Dietrich, V. Rothberg, L. Füracker, A. Ziegler, and D. Lohmann, "cHash: Detection of Redundant Compilations via AST Hashing," Proc. 2017 USENIX Annual Technical Conference (ATC), 2017.

[35] "Precompiled header files." https://docs.microsoft.com/en-us/cpp/build/creating-precompiled-header-files?view=vs-2019.

[36] "Using the gnu compiler collection (gcc): Precompiled headers." https://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.

[37] "Precompiled header and modules internals — clang 9 documentation." https://clang.llvm.org/docs/PCHInternals.html.

[38] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," 2018 USENIX Annual Technical Conference (USENIX ATC 18), Boston, MA, pp.601–614, USENIX Association, July 2018.

[39] "Welcome to the Buildbot for the LLVM project!." http://lab.llvm.org:8011/.

[40] G. Garen, "[webkit-dev] growing tired of long build times? check out this awesome new way to speed up your build… soon (hint: It's not buying a new computer)," 2017. https://lists.webkit.org/pipermail/webkit-dev/2017-August/029508.html.

[41] P.N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, Introduction to Data Mining, 2nd ed., Pearson, 2019.

[42] A.H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," ACM Comput. Surv., 2018.

[43] "Zapcc – A (Much) Faster C++ Compiler." https://www.zapcc.com/.

[44] G.D. Reis, M. Hall, and G. Nishanov, "A Module System for C++ (Revision 4)," 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0142r0.pdf.

[45] "Llvm link time optimization: Design and implementation." https://llvm.org/docs/LinkTimeOptimization.html.

[46] T. Johnson, M. Amini, and X.D. Li, "ThinLTO: Scalable and incremental LTO," Proc. 2017 International Symposium on Code Generation and Optimization (CGO), 2017.

[47] A. Ayers, S. de Jong, J. Peyton, and R. Schooler, "Scalable Cross-Module Optimization," Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI), 1998.

[48] S. Moon, X.D. Li, R. Hundt, D.R. Chakrabarti, L.A. Lozano, U. Srinivasan, and S.-M. Liu, "SYZYGY – A Framework for Scalable Cross-Module IPO," Proc. 2004 International Symposium on Code Generation and Optimization (CGO), 2004.

[49] P.W. Sathyanathan, W. He, and T.H. Tzen, "Incremental whole program optimization and compilation," Proc. 2017 International Symposium on Code Generation and Optimization (CGO), 2017.

[50] P. Briggs, D. Evans, B. Grant, R. Hundt, W. Maddox, D. Novillo, S. Park, D. Sehr, I. Taylor, and O. Wild, "WHOPR - Fast and Scalable Whole Program Optimizations in GCC," 2007. https://www.gnu.org/software/gcc/projects/lto/whopr.pdf.

[51] D.X. Li, R. Ashok, and R. Hundt, "Lightweight Feedback-Directed Cross-Module Optimization," Proc. 2010 International Symposium on Code Generation and Optimization (CGO), 2010.

[52] X.D. Li, R. Ashok, and R. Hundt, "LIPO - Profile Feedback Based Lightweight IPO." https://gcc.gnu.org/wiki/LightweightIpo.

Takafumi Kubota received his B.E. and M.E. degrees from Keio University in 2015 and 2017. He is currently a Ph.D. student in the Department of Information and Computer Science at Keio University. His research interests include operating systems, static analysis, compilers and software diagnosability.

Kenji Kono received the BSc degree in 1993, MSc degree in 1995, and PhD degree in 2000, all in computer science from the University of Tokyo. He is a professor in the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE, ACM, and USENIX.