Enabling Fine-Grained Incremental Builds by Making Compiler Stateful

Ruobing Han
Georgia Institute of Technology
Atlanta, USA
hanruobing@gatech.edu

Jisheng Zhao
Georgia Institute of Technology
Atlanta, USA
jisheng.zhao@cc.gatech.edu

Hyesoon Kim
Georgia Institute of Technology
Atlanta, USA
hyesoon@cc.gatech.edu

***Abstract*—Incremental builds are commonly employed in software development, involving minor changes to existing source code that is then frequently recompiled. Speeding up incremental builds not only enhances the software development workflow but also improves CI/CD systems by enabling faster verification steps.**

Current solutions for incremental builds primarily rely on build systems that analyze file dependencies to avoid unnecessary recompilation of unchanged files. However, for the files that do undergo changes, these build systems simply invoke compilers to recompile them from scratch. This approach reveals a fundamental asymmetry in the system: while build systems operate in a stateful manner, compilers are stateless. As a result, incremental builds are applied only at a coarse-grained level, focusing on entire source files, rather than at a more fine-grained level that considers individual code sections.

In this paper, we propose an innovative approach for enabling the fine-grained incremental build by introducing statefulness into compilers. Under this paradigm, the compiler leverages its profiling history to expedite the compilation process of modified source files, thereby reducing overall build time. Specifically, the stateful compiler retains dormant information of compiler passes executed in previous builds and uses this data to bypass dormant passes during subsequent incremental compilations.

We also outline the essential changes needed to transform conventional stateless compilers into stateful ones. For practical evaluation, we modify the Clang compiler to adopt a stateful architecture and evaluate its performance on real-world C++ projects. Our comparative study indicates that the stateful version outperforms the standard Clang compiler in incremental builds, accelerating the end-to-end build process by an average of 6.72%.

I. Introduction

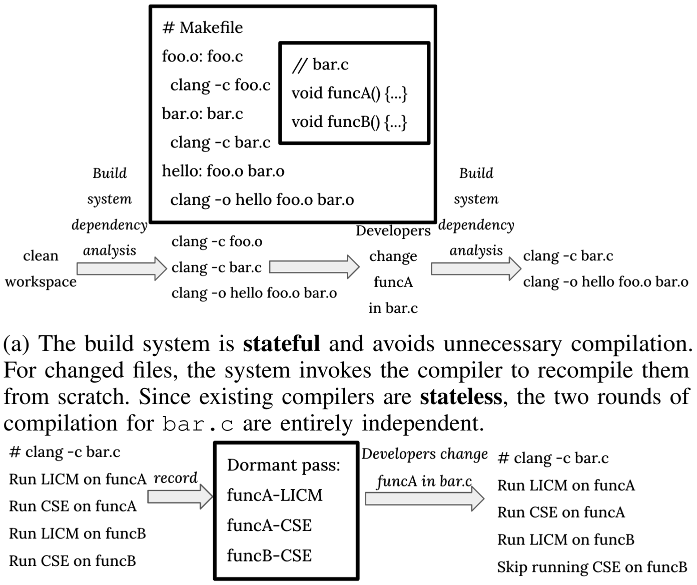
In modern software development workflows, developers often add new features or fix existing issues in already-established codebases. A typical development cycle starts by integrating a new patch into the current codebase. This is followed by the build and execution phases. The results of these runtime tests then guide further code revisions. In this setup, projects go through multiple build cycles featuring incremental code changes. Additionally, Continuous Integration and Continuous Deployment (CI/CD) systems, which are essential parts of today's project management frameworks, also rely on incremental builds [22]. These CI/CD systems add new commits to existing codebases, and then compile and run the code to make sure that these changes do not negatively affect current functionalities. Incremental builds are also used in other scenarios. For example, [26], [25] use incremental builds to explore the configuration space of Linux kernels, while [34] introduces an on-demand instrumentation framework that speeds up the compilation process through incremental builds.

TABLE I: The recent commits in the LLVM project. For the files changed by these commits, only a few functions have been changed. These **unchanged functions** in the **changed files** are compiled from scratch during the incremental builds.

Commits	Changed Functions	Unchanged Functions in Changed Files	Changed C++ Files
96e1914	1	53	1
26f230f	2	40	2
967d953	8	178	1
aca23d8	1	98	2
edb9fab	4	25	1
f09360d	9	20	1
6012fed	2	117	1
8dd8c4a	1	59	1

In the build process of modern C++ projects, the compilation phase generates multiple object files from source code. These object files are then linked together in the linking phase to create executable files or libraries. A key feature of incremental builds is that only the changed files are recompiled. This characteristic is managed by build systems like Make [32], Ninja [33], and Bazel [30]. These systems monitor changes in source files and use dependency analysis to only recompile the affected files. Figure 1(a) shows an example C project that uses the GNU Make build system. This project has three targets: `foo.o`, `bar.o`, and `hello`. In the first round of the build process, all three targets are built from a clean workspace. If developers later modify the source code of `bar.c`, the build system, recognizing that `foo.c` has not changed, would issue only two compilation commands. Importantly, these build systems are `stateful`; they retain information from past builds to optimize subsequent incremental builds.

Conversely, compilers are `stateless`. In the example involving `bar.c`, two separate rounds of compilation (`clang -c bar.c`) are entirely independent of each other. This means that the second round of compilation cannot benefit from the first. During incremental build process, the changed files are required to compile from scratch. We analysis the



```
# Makefile
foo.o: foo.c
  clang -c foo.c
bar.o: bar.c
  clang -c bar.c
hello: foo.o bar.o
  clang -o hello foo.o bar.o
```

```
// bar.c
void funcA() {...}
void funcB() {...}
```

```
clang -c foo.o
clang -c bar.c
clang -o hello foo.o bar.o
```

```
clang -c bar.c
clang -o hello foo.o bar.o
```

```
# clang -c bar.c
Run LICM on funcA
Run CSE on funcA
Run LICM on funcB
Run CSE on funcB
```

```
# clang -c bar.c
Run LICM on funcA
Run CSE on funcA
Run LICM on funcB
Skip running CSE on funcB
```

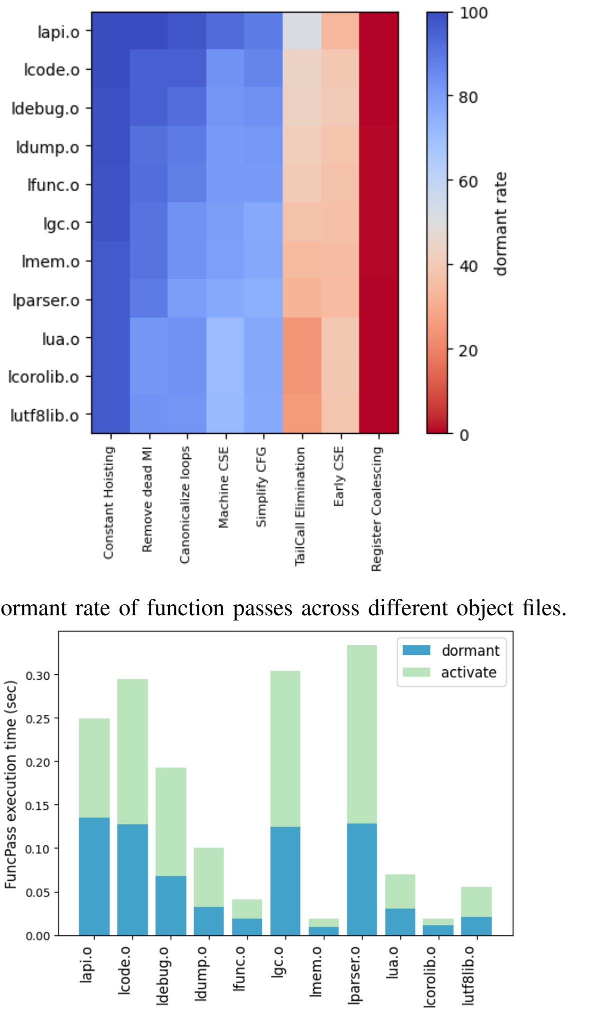
(a) The build system is **stateful** and avoids unnecessary compilation. For changed files, the system invokes the compiler to recompile them from scratch. Since existing compilers are **stateless**, the two rounds of compilation for `bar.c` are entirely independent.

(b) The improvement brought by the stateful compiler. The stateful compiler records dormant passes during the first compilation. In the second compilation, if only `funcA` is changed, the dormant information of `funcB` can be used to skip those dormant passes applied on `funcB`.

Fig. 1: An example C project consisting of three targets: two object files and one executable file.

recent commits for LLVM project in Table I and find that **most functions in the changed files are unchanged**. These unchanged functions can **NOT** get any benefit from the incremental build, but to be compiled from scratch, even though they are not changed. In other words, current incremental builds operate solely at a **coarse-grained** level, focusing on source-file-level modifications rather than adopting a **fine-grained** approach that takes into account changes at the level of individual code sections within files.

Modern compilers often apply hundreds of passes on programs. This process constitutes a significant portion of the overall compilation time. However, not all passes result in modifications to the programs. Passes that leave the programs unchanged are termed `dormant`, while those that modify the programs are called `active`. For example, if a program has no loops, any loop-related passes would be dormant, as their applications would not change the program. We measured the dormant rate $\left(\frac{\text{number of dormant passes}}{\text{total number of passes}}\right)$ of multiple LLVM function passes during the compilation of LUA project [4]. Figure 2(a) displays the dormant rates for different object files and passes. For instance, the Register Coalescing pass, which aims to minimize variable-to-variable move operations, is always active. On the other hand, the Constant Hoisting pass, applying only to a limited subset of functions containing expensive constants that can be hoisted, is often dormant. Despite being dormant, these passes still consume a substantial amount of computational resources, as shown in Figure 2(b). We also measure the processes of building LUA, Git, and CPython using a single process and found that 27.2%, 13.27%, and 21.02% of the overall build time were spent on dormant function passes, which do not change the programs. Thus, we claim that by wisely skipping the dormant passes, we can save up to approximately 25% of the build time for these projects.



(a) Dormant rate of function passes across different object files.

(b) Execution time taken by dormant and active function passes.

Fig. 2: Analysis of the LUA build process reveals that a considerable amount of time is spent on dormant passes.

It's worth noting that predicting which passes will be dormant is a complex task, influenced by specific characteristics of both the program and the compiler passes [19], [14], [20].

By making compilers stateful, incremental compilations can be speeded up by skipping the dormant passes. This is illustrated in Figure 1(b). A stateful compiler could record all dormant passes during the first round of compilation and use this information in the second round to expedite the process.

In summary, existing build systems generally perform incremental builds at the **coarse-grained** level of source files. In contrast, the stateful compiler introduced in this paper offers additional speedup by enabling **fine-grained** incremental builds at the level of code sections (i.e., functions, loops). We summarize the key contributions of this paper as follows:

- Introducing the concept of the fine-grained incremental build;
- Describing the workflow that leverages dormant pass information to apply fine-grained incremental builds;
- Identifying the prerequisites for converting conventional stateless compilers into stateful versions;
- Adapting the Clang compiler to a stateful compiler and assessing its performance on real C/C++ projects.

II. Background

For the purposes of discussion, and without loss of generality, this paper uses terminology and concepts from the LLVM framework [21]. However, it should be noted that similar concepts are prevalent in other modern compilers as well.

A. LLVM Hierarchy

The hierarchical structure of LLVM is designed to analyze and optimize programs at various levels of granularity. The three most common levels of this hierarchy are modules, functions, and loops.

A module is essentially a unit of compilation, roughly equivalent to a source file (e.g., a '.cpp' file) in C/C++ compilation. Importantly, build systems manage the incremental build at the module level, which is the coarsest level of granularity, compared with the following two levels

Functions represent discrete procedures contained within a module. A function comprises a sequence of instructions, input arguments, return types, and locally defined variables.

Given the pervasive nature of loops in programming, LLVM compiler treats loops as distinct entities to enable specialized optimizations. A loop refers to a repetitive section of code iteratively executed over a collection of elements. This level of abstract allows LLVM compiler to perform intricate analyses of loop characteristics and independent loop optimizations apart from other code sections.

In LLVM, all these hierarchies are represented using LLVM Intermediate Representations (IRs).

B. LLVM Passes

A *pass* acts as a unit of analysis or transformation that engages with various types of IRs. LLVM passes can be categorized into two types: *analysis passes*, which only analyze the IRs without making any modifications, and *transformation passes*, which utilize the analyzed data to modify the IRs. Given that LLVM PassManagers automatically trigger the necessary analysis passes for any given transformation pass, this paper **exclusively** focuses on transformation passes.

The hierarchy of LLVM passes mirrors that of LLVM IRs, with module, function, and loop passes being the most prevalent. Each type of pass operates on its corresponding level of IR. Moreover, passes operating on a given level of the hierarchy are **independent** of IRs at the same or higher levels. For instance, a function pass applied to one function IR will not be influenced by any other function IRs or module IRs. This independence allows for parallel application of passes across different IRs.

In this paper, we particularly emphasize the speed up brought by the stateful compiler based on the build systems. For this reason, our stateful compiler records only dormant function and loop passes. We opt to ignore module passes because, in the context of incremental builds, it is generally assumed that the modules under compilation will always contain semantics-level changes after being filtered by the build systems. This eliminates the possibility of reusing module-level information from previous builds.

C. PassManager

The PassManager manages the applications of passes. It automatically takes care of any analysis passes that are needed for transformation passes. This makes it simpler for developers, as they only need to think about the transformation passes during the design of compilation pipelines.

To extend the conventional stateless compiler to stateful compiler, we need to update the PassManager to make it record the dormant information as profiling data, and utilize the profiling data to check the dormant status and skip the dormant passes during incremental builds.

III. Stateful compiler design

Our design focuses on intra-procedural optimization passes that rely on intra-procedural analysis. Specifically, the stateful compiler focuses on optimizing two kinds of intra-procedural passes: the function passes and loop passes.

A. The Basic Assumption

Before discussing the design of the stateful compiler, we first establish the fundamental assumption critical to the validation of the stateful compiler. The assumption is that the passes are **functional**; as long as the input to the passes remains unchanged, the passes' dormant status will also remain unchanged. Although most modern compilers meet this assumption, we have discovered that implementing a fully valid stateful compiler in practice presents challenges. These challenges stem from the numerous factors that can influence the dormant status of compiler passes. To ensure correctness, the compiler must account for all these factors.

We propose two kinds of equivalence [12] to better assess the validity of the stateful compiler. **Binary Equivalence** signifies that the stateful compiler generates binary files that are exactly identical to those produced by a conventional compiler. A less stringent form of equivalence is **Semantic Equivalence**, where the stateful compiler generates programs with correct semantics. For LLVM, all passes in the optimization pipelines (e.g., loop unroll, LICM, DCE) do not alter the semantics of IRs. Therefore, an imperfect stateful compiler that erroneously skips active passes will not breach semantic equivalence. Due to the complexity of modern compilers, achieving binary equivalence in all cases is challenging. In Section VII-B, we examine various scenarios that could compromise binary equivalence in our current implementation. However, we have found that violations of binary equivalence are relatively rare in practice. In our evaluation, the stateful compiler attained binary equivalence in all tested cases.

B. The workflow of the stateful compiler

To speed up the incremental build with the stateful compiler, two kinds of build processes are involved: *profiling build* and *fine-grained incremental build* (Figure 3). The stateful compiler during profiling build behaves the same as the conventional stateless compiler, except it also records the dormant information and stores this information into storage. The profiling build is only required for the first time. For the following build processes, they are fine-grained incremental builds, in which the stateful compiler loads the dormant information from storage and uses it to skip the dormant passes to speed up the compilation. In our implementation, incremental builds do not update the profiling data. The trade-offs associated with updating profiling data on the fly are discussed in Section VII-A.

To implement the stateful compiler, we need to solve two challenges: 1) how to represent the dormant information; and 2) how to utilize the dormant information during the incremental build process. We discuss the first challenge in this section, as the dormant information is a general concept and should be compiler-independent. In the next section, we introduce several implementation details for the second challenge.

C. Representing dormant information

In general, the dormant information are triples of code sections, passes, and flags. For example, a triple (LICM, FuncA, 'licm-control-flow-hoisting=false') means FuncA (code section) will not be changed by the LICM (pass) when the control flow hoisting option is turned off (flag). We acknowledge that there are other factors (discussed in Section VII-B) that can affect the dormant status. However, in practice, we have found that these three factors are sufficient for our evaluated cases.

To explicitly represent the dormant information and store it in storage, the stateful compiler needs to represent each element in the triple with a data structure. For the pass, the stateful compiler directly hashes the pass names (text strings) into scalar values. It applies the same solution to hash the flags (text strings) into scalar values. This solution requires that the compiler will not be updated between the profiling build and the fine-grained incremental build. Since modern compilers usually have stable versions that do not update frequently, we believe this requirement is satisfied in most cases. If the developers upgrade the compiler, they have to erase all existing dormant information in the storage and re-execute the profiling build to maintain correctness.

The code sections (e.g., function, loop) are hard to hash. The compilers generally represent code sections as IRs, which contain control flow graphs (CFGs) and properties (e.g., return type, arguments for function IRs). To implement the stateful compiler, we need to build a hash function that accepts IRs as input and outputs scalar values. Although there are tree-structured data hashing solutions [29], [36], [35], [17], they only hash the topology structures. For the incremental build, it is quite common that, after the code update, the topology remains the same, but the dormant status changes. For example, changing the binary instructions in a function will not change the topology in the CFG but may affect the dormant status of passes (e.g., constant folding). Thus, the hash values should reflect not only the topology structures but also other properties that may affect the dormant result.

In summary, the ideal hash function for the stateful compiler should be **lightweight** and **sensitive** to all dormant status-related factors in IRs. To fulfill the requirements of the stateful compiler, we propose a BFS-based hash solution (Algorithm 1) to hash two common kinds of IRs, Function IRs and Loop IRs, to scalars. The algorithm contains several hierarchical hash functions. The basic hash function maps an instruction to a scalar value. This basic hash function is used to apply another higher-level hash function, which maps a block to a scalar value. Since the instructions in a block have a sequential order, the hash function continuously updates the hash value with each instruction, making the order of instructions affects hash values. The third hash function maps a function IR to a scalar value. It applies a BFS traverse on the CFG, and the starting point is the function's entry block. By strictly pushing the unvisited blocks into the queue in the order of the operands in the branch instructions, the output hash values are affected by the order of the edges. The hash value is initilized with the function properties before the BFS traverse, to make the hash value also reflect function properties. We also introduce the hash function for the loop IRs. Since both loop IRs and function IRs contain CFGs as the major component, the loop IR hash function is similar to the function IR hash function.

D. Organizing dormant information in storage

Dormant information consists of triples that represent code sections, passes, and flags. As modern projects often involve millions of code sections during compilation, the organization of this dormant information in storage is a critical design consideration, which directly impacts the latency involved in loading, storing, and querying the dormant information.

In our implementation, the stateful compiler organizes dormant triples on a **per-object-file** basis. Specifically, all dormant information triples generated during the compilation of an object file are grouped together. These triples are then divided into two categories: one for function passes and another for loop passes, each stored in separate files. This *per-object-file* organization is motivated by the observation that each compilation command targets a unique object file. As a result, only the dormant information specific to that particular object file is needed for its compilation.

IV. Implementation

To demonstrate the feasibility of our approach, we develop a proof-of-concept stateful compiler based on the LLVM/Clang framework. The LLVM compiler architecture comprises three main components: the frontend, the optimization pipeline, and the CodeGen. The frontend is responsible for parsing the source code and generating IRs. These IRs undergo passes in the optimization pipeline, which includes both hardware-independent and machine-specific optimizations. Finally, the transformed IRs are used to generate binary code in the

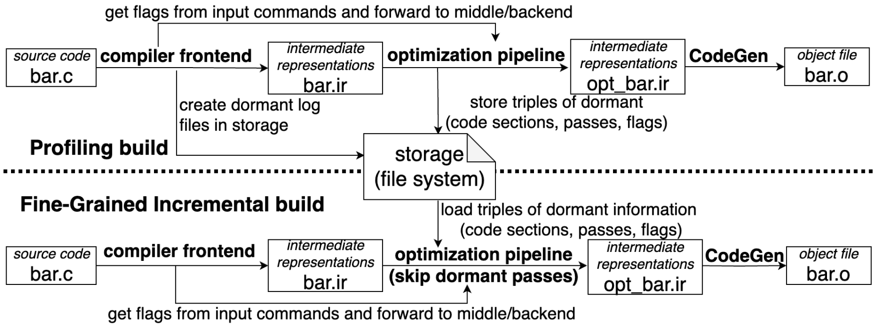

get flags from input commands and forward to middle/backend
source code
bar.c
compiler frontend
intermediate representations
bar.ir
optimization pipeline
intermediate representations
opt_bar.ir
CodeGen
object file
bar.o
create dormant log files in storage
store triples of dormant (code sections, passes, flags)
Profiling build
storage (file system)
Fine-Grained Incremental build
load triples of dormant information (code sections, passes, flags)
source code
bar.c
compiler frontend
intermediate representations
bar.ir
optimization pipeline (skip dormant passes)
intermediate representations
opt_bar.ir
CodeGen
object file
bar.o
get flags from input commands and forward to middle/backend

Fig. 3: The workflow of applying the stateful compiler in the build process. The profiling build records the dormant information and stores it in storage. The fine-grained incremental build loads the dormant information and utilizes it to skip dormant passes.

CodeGen phase. To transform LLVM compiler into a stateful version, we introduce modifications to both the frontend and the optimization pipeline.

A. Frontend

As depicted in Figure 3, the frontend of the stateful compiler carries out two additional tasks compared to a conventional stateless compiler. First, during the profiling build, the frontend creates files for storing dormant information, which are later accessed by the optimization pipeline. Second, it analyzes user inputs to extract flags relevant to the optimization pipeline and forwards this information to the optimization pipeline.

We enhance Clang with a plugin that reads from the Linux command-line interface (`/proc/pid/cmdline`) to gather compilation flags. The plugin filters out irrelevant flags, retaining only those prefixed with '-mllvm,' as these flags affect the optimization pipeline. The filtered flags are stored in an environment variable named `OBJECT_FLAGS`, subsequently accessed by the optimization pipeline.

Additionally, the plugin creates empty log files in the file system to store dormant information. Each object file is associated with two log files: one for recording dormant information related to Function Passes and another for Loop Passes. The paths to these log files are stored in two environment variables: `FUNC_DORMANT_PATH` and `LOOP_DORMANT_PATH`.

B. Optimization pipeline

In LLVM, all passes in optimization pipelines are managed by PassManagers. The primary modification lies in the FuncPassManager and LoopPassManager class. The updated Function PassManager is presented in Algorithm 2. The same updates is also applied to the LoopPassManager.

1) Profiling Build: The stateful compiler records all dormant triples in $dormant\_triples$ and stores this information to the file system. The storage path `FUNC_DORMANT_PATH` is set by the Clang plugin in the frontend.

2) Fine-Grained Incremental Build: The PassManager first checks whether $dormant\_triples$ is empty. If it is, the dormant information is loaded from storage. In LLVM, all FunctionPassManager instances are derived from the same class. By making $dormant\_triples$ a **class static member**, the dormant information is shared among all FunctionPassManager instances and initilized (loaded from storage) only once. The loaded information is then used to check the dormant status during the execution of passes. For each pass, the FuncPassManager examines whether the corresponding triple is recorded in the dormant information. If it is, the pass is directly skipped. Otherwise, the pass is applied as it would be in a stateless compiler. To minimize latency, the hash function is invoked only prior to the optimization pipelines and whenever the function IRs are modified by passes.

LLVM's Link Time Optimization (LTO) also utilizes the PassManager class. Consequently, developers can apply analogous modifications to bypass dormant passes in LTO.

C. Extending Stateless Compilers to Stateful Versions

The preceding discussion on implementation primarily focuses on the LLVM infrastructure. However, the insights gained from stateful compilers can also be applied to extend other conventional stateless compilers. In this section, we outline the ideal features that existing compilers should possess to facilitate their transformation into stateful versions.

Separated Analysis and Transformation: Generally, IR passes can be categorized into two types: analysis passes, which gather information without modifying the IRs; and transformation passes, which change the IRs based on the information analyzed. Analysis passes are inherently dormant but cannot be skipped. It is essential for stateful compilers to distinguish between analysis and transformation passes to prevent inadvertently skipping any analysis phases.

Explicit Dormant Information: An ideal compiler should offer a straightforward method for reporting whether a pass changes the IRs. In recent versions of LLVM pass managers, each pass returns a set of preserved analyses rather than a boolean value indicating changes, as was the case in legacy LLVM PassManagers. Although this new design avoids re-executing redundant analyses, it complicates the task of identifying dormant passes in the context of stateful compilers.

Functional Passes: Passes should solely depend on their inputs (code sections and flags) and be unaffected by external factors. In terms of dormant information, this implies that the dormant status of a pass should remain constant as long as its inputs are unchanged. Moreover, passes should be free of side effects. If a pass alters analysis results without changing code sections

Algorithm 1 Hash functions for mapping LLVM IRs to scalars.

```
Input: VHASH: The normal scalar-to-scalar hash function
function HASHINST(inst)
    hash_val ← VHASH(0xdeadbeef, inst.opcode)
    for operand ∈ inst do
        if $operand$ is $instruction$ then
            hashed_inst ← HASHINST(operand)
            hash_val ← VHASH(hash_val, hashed_inst)
        else
            hash_val ← VHASH(hash_val, operand.str)
        end if
    end for
    return hash_val
end function
function HASHBLOCK(block)
    hash_val ← $0xdeadbeef$
    for inst ∈ block do
        hash_val ← VHASH(hash_val, HASHINST(inst))
    end for
    return hash_val
end function
function HASHFUNCTION(func)
    hash_val ← $0xdeadbeef$
    for arg ∈ func.args do
        hash_val ← VHASH(hash_val, arg.type)
    end for
    hash_val ← VHASH(hash_val, func.ret_type)
    visited_blocks ← ∅
    queue ← initialize queue with func.entry
    while queue is not empty do
        current_block ← queue.dequeue()
        if current_block ∈ visited_blocks then
            continue
        end if
        visited_blocks ← visited_blocks ∪ current_block
        hashed_block ← HASHBLOCK(current_block)
        hash_val ← VHASH(hash_val, hashed_block)
        queue.push(current_block.successors())
    end while
    return hash_val
end function
function HASHLOOP(loop)
    hash_val ← $0xdeadbeef$
    visited_blocks ← ∅
    queue ← initialize queue with loop.header
    while queue is not empty do
        current_block ← queue.dequeue()
        if current_block ∈ visited_blocks then
            continue
        end if
        if current_block ∉ loop.blocks then
            continue
        end if
        visited_blocks ← visited_blocks ∪ current_block
        hashed_block ← HASHBLOCK(current_block)
        hash_val ← VHASH(hash_val, hashed_block)
        queue.push(current_block.successors())
    end while
    return hash_val
end function
```

Algorithm 2 The updated function in LLVM FuncPassManager.

```
procedure RUNONFUNCTION(Func)
    build_type ← os.env("STATEFUL_MODE")
    flags ← os.env("OBJECT_FLAGS")
    dormant_path ← os.env("FUNC_DORMANT_PATH")
    changed ← False
    hashed_func ← hash_function(Func)
    $static$ dormant_triples ← []
    if build_type = "fine_grained_inc_build" then
        if dormant_triples is empty then
            dormant_triples ← load(dormant_path)
        end if
    end if
    for pass ∈ passes do
        if build_type = "stateless" then
            changed ← changed ∨ pass(Func)
        else if build_type = "profiling_build" then
            local_changed ← pass(Func)
            changed ← changed ∨ local_changed
            if not local_changed then
                dormant_triples.append(
                        (hashed_func, pass.name, flags))
            else
                hashed_func ← hash_function(Func)
            end if
        else if build_type = "fine_grained_inc_build" then
            if (hashed_func, pass.name, flags)
                    ∈ dormant_triples then
                continue
            else
                local_changed ← pass(Func)
                changed ← changed ∨ local_changed
                if local_changed then
                    hashed_func ← hash_function(Func)
                end if
            end if
        end if
    end for
    if build_type = "profiling_build" then
        store(dormant_triples, dormant_path)
    end if
    return changed
end procedure
```

in a way that could impact other passes, it possesses a side effect and thus cannot be considered dormant and skipped.

V. Evaluation

A. Evaluation Setup

To mimic the real-world software development process, we perform incremental builds by applying real git commits to the codebase. Specifically, for a given project, we first check it out to an earlier commit and execute a profiling build on the source code. Subsequently, we apply later git commits and carry out the incremental build process.

We choose six popular C++ applications for our evaluation. LUA [4] is a lightweight, embeddable scripting language and currently leads in the realm of gaming scripts. Git [28] is a distributed version control system extensively employed for tracking changes in source code during software development. CPython [2] serves as the reference implementation of Python, offering a high-performance interpreter along with a comprehensive standard library. PostgreSQL [6] is a robust open-source relational database management system, providing reliable and scalable solutions for structured data storage and retrieval. OpenCV [7] is an open-source computer vision library featuring a comprehensive set of tools and algorithms for image and video processing. Lastly, LLVM [21] is a compiler infrastructure project that facilitates the development of compilers, programming languages, and associated tools. The specifics of these projects are detailed in Table II.

To implement our stateful compiler, we modify Clang-14. We employ GNU Make [32] as the build system, as it serves as the default build system for these selected projects. All experiments are conducted on a server equipped with 64 Intel CPUs (Intel Xeon Gold 6226R) and 400 GB of memory.

TABLE II: The projects used for evaluating the build process.

Name	Latest Commit	Make -j	Build Time (sec)	Number of C++ Files
LUA	64431851	1	5.931	40
Git	9748a682	8	8.243	565
CPython	e1d45b8e	8	12.07	357
Postgres	29cf61ad	32	38.905	1458
OpenCV	8839bd57	32	160.365	2384
LLVM	d954d975	64	312	38855

B. Speedup Evaluation

We first evaluate the speedup achieved by the stateful compiler. The formula for calculating speedup is given by: $100\% \times \frac{\text{baseline\_time}-\text{stateful\_time}}{\text{baseline\_time}}$.

We measure two types of time. The *compilation time* refers to the total time taken by the compiler, assuming all targets are compiled with a single CPU processor. This time metric directly reflects the speedup achieved by the stateful compiler. We also consider the *End-to-End time*, which is the total time taken to execute the 'make' command. This includes the time spent on non-compilation tasks, such as linking and folder management. For all projects except LUA, we execute the build process using multiple processors. While these factors may introduce bias when illustrating the speedup afforded by the stateful compiler, we include the End-to-End time to demonstrate that the stateful compiler does, in fact, improve the overall elapsed time.

We evaluate the speedup on these four relatively lightweight projects by applying the 20 most recent commits for incremental builds. We display the distribution of speedup across these 20 build processes for each project and each optimization pipeline (O1, O2, and O3) in Figure 4. Commits that do not modify C/C++ files are excluded from this evaluation. On average, across all commits and projects, the stateful compiler achieves a 6.72% speedup in end-to-end build time compared to the stateless compiler (Clang-14).

The speedup demonstrates considerable variability across projects, optimization pipelines, and individual commits. This variability is attributed to several factors. First, the effectiveness of the stateful compiler is contingent on the number of dormant passes it can skip, a quantity that varies significantly depending on the project and the chosen optimization pipeline. Second, within the same project, different commits necessitate varying degrees of recompilation. For instance, commits involving changes to header files (`.h`) generally trigger more recompilation than those solely modifying source files (`.cpp`).

We use LUA as a case study, since it can be built using a single processor. We measure the execution time for all function passes and calculate the dormant ratio (Figure 5). Establishing a formulaic relationship between the dormant rate and Function Pass speedup is challenging because it is contingent on specific passes and functions. However, we observe that, generally, targets with a high dormant rate tend to experience significant speedup when compiled with the stateful compiler.

The speedup achieved in LUA, Git, and CPython falls short of the upper bound speedup discussed in Section I, primarily for two reasons. First, stateful compilers incur additional overhead in checking the dormant status. Second, there are **false positive** instances where the stateful compiler unnecessarily applies dormant passes. Some code modifications do not change the dormant status of passes but change the IR hash values, thus precluding the use of dormant data. False positives also occur when code modifications make active passes dormant.

C. Overhead

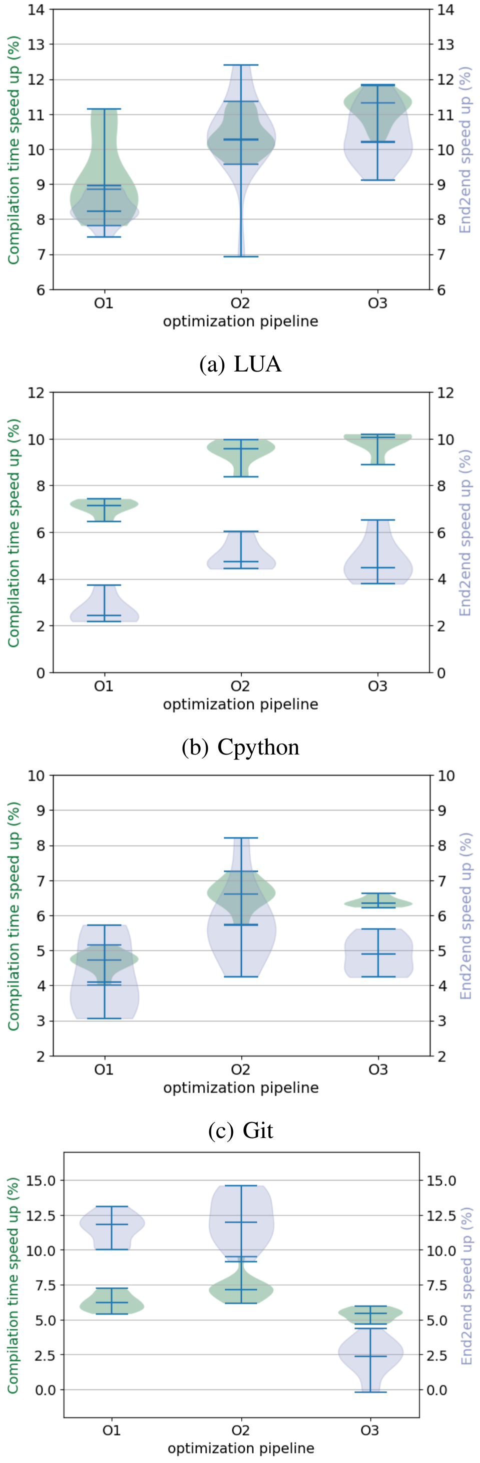
1) Profiling Build: Compared to conventional stateless compilers, the stateful compiler necessitates an initial profiling build to record dormant information and store it in storage. The end-to-end execution time for the profiling build across five projects is depicted in Figure 6. The build time for LUA is less than one second and is therefore not shown in the figure. The execution time is measured as the median of seven rounds of execution. Across all projects, the end-to-end build time remains consistent between the normal build and the profiling build. The most significant difference is observed in LLVM, where the normal build takes 313.46 seconds, while the profiling build requires 315.78 seconds. This results in a mere 0.73% difference. We also measure the size of the dormant information for six projects, as shown in Table III. Since all dormant information is stored in storage, the storage overhead is negligible for modern computers.

TABLE III: The size of dormant information. The dormant information is stored in storage.

	LUA	CPython	Git	Postgres	OpenCV	LLVM
Size (MB)	0.72	10.75	9.62	18.92	151.24	259.89

2) Fine-Grained Incremental Build: Although the stateful compiler can skip dormant passes, it also introduces overhead for loading dormant information, hashing the IRs, and checking the dormant status. All these factors are related to the program: the overhead for loading dormant information and checking the dormant status depends on the number of dormant triples recorded during the profiling build. Meanwhile, the overhead for hashing the IRs depends on the size of the IRs.

We profile the compilation time in four projects and measure the total amount of time used for executing the passes,



(a) LUA

(b) Cpython

(c) Git

(d) Postgres

Fig. 4: The compilation/end2end speedup brought by the stateful compiler. Each project is evaluated 20 times by sequentially applying the 20 most recent commits.

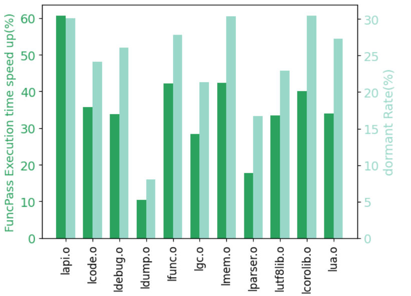


Fig. 5: Correlation between FunctionPass execution speedup and FunctionPass dormant rate in the LUA project. Higher dormant rates generally result in greater speedup when using the stateful compiler.

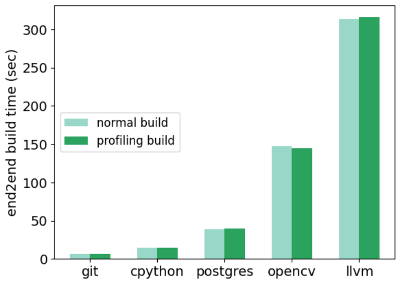


Fig. 6: The end-to-end execution time required to build the projects using 64 parallel processes. The profiling build does not exhibit a slowdown when compared to the Clang build process. Execution times are measured as the median of seven rounds of executions.

loading dormant information, hashing the IRs, and checking the dormant status (Figure 7). In all projects, more than 90% of the total execution time is spent on executing the passes. When it comes to the overhead brought by the stateful compiler, checking for dormant status costs the most, making up 77% of the total overhead. Hashing also contributes to 20% of the total overhead. As the profiling data are loaded only once, the loading overhead is negligible.

D. Stateful Compiler + CCache

CCache [31] is a popular solution for speeding up incremental builds. It maintains a cache in the file system to store executed C/C++ commands and the generated object files. CCache can be regarded as a wrapper of the compilers. For any upcoming call to the compiler, CCache first checks whether the command has been executed before and, if so, whether the input files have semantics modifications. If CCache finds that the upcoming call would generate the same object files as a previous call (*CCache hit*), CCache directly loads the corresponding object files from the cache without invoking the real compiler. Otherwise, CCache invokes the real compiler to generate the object files and stores the generated files into the cache. The hit/miss rate of CCache is the most critical metric for measuring its effectiveness. CCache is included in the build

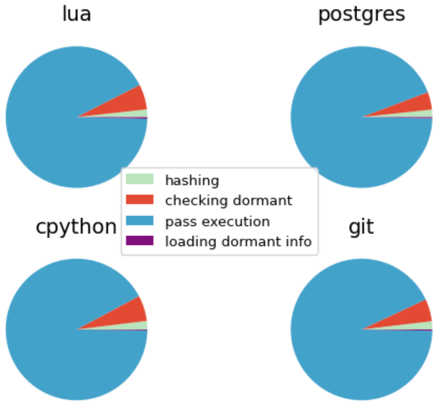


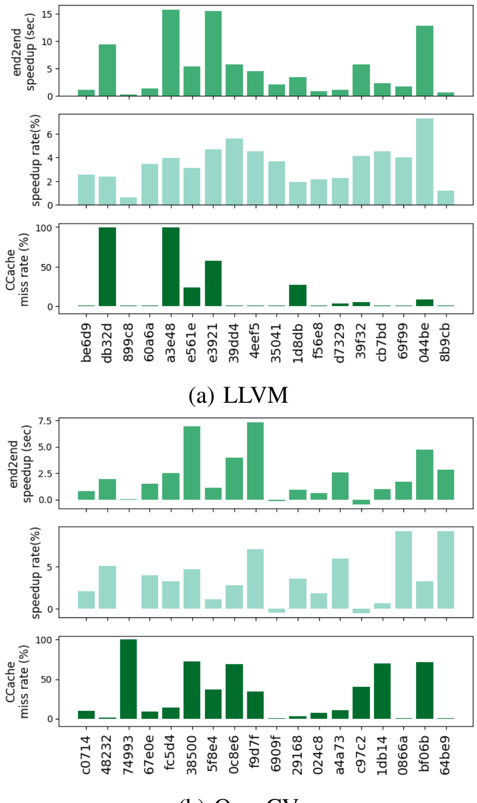
Fig. 7: Profiling data of compilation time during the fine-grained incremental build process. The overhead introduced by the stateful compiler is relatively small compared to the time spent on executing passes. The pass execution time accounts from 91.98% to 93.95% of the total time in the four projects.

processes of some well-known projects, such as LLVM [21], Linux kernel [3], and Mozilla Firefox [5].

The stateful compiler works **orthogonally** with CCache. During an incremental build, when multiple source files are updated, CCache can filter out the compilation of those files that have not undergone semantic changes. For the remaining changed files, the stateful compiler is used for compilation.

In this section, we integrate CCache with the stateful compiler to measure the improvement on two projects with large code bases: OpenCV [7] and LLVM [21]. The end-to-end time for both projects includes both compilation and linking time. To mitigate the bias introduced by the linking time, we adjust the configuration settings to reduce the amount of linking. For example, we disable the `LLVM_BUILD_TOOLS` option in LLVM's CMake files to reduce the time spent on linking for generating executable files. Additionally, instead of using sequential commits, we test the incremental builds using commits at intervals for two reasons: first, some commits do not contain meaningful modifications, such as those fixing typos in comments, which would not trigger any recompilation; second, by gathering modifications from multiple commits, we can increase the number of files changed in the incremental build, thereby increasing the time spent on compilation.

As shown in Figure 8, using the stateful compiler in addition to CCache further accelerates incremental builds. The extent of the benefit provided by the stateful compiler varies across commits. We find that there is no consistent relationship between the CCache miss rate and the speedup achieved through the use of the stateful compiler. We discuss these four cases in more detail below.



(a) LLVM

(b) OpenCV

Fig. 8: The speedup achieved by integrating the stateful compiler with CCache, compared to using CCache alone. Generally, the integration of the stateful compiler with CCache further accelerates the incremental build process.

High CCache miss rate and High stateful compiler speedup: In Figure 8(a), for the commits between 60a6a and a3e48, the incremental build experiences a high CCache miss rate (96.38%) but also benefits significantly from the stateful compiler (3.96%). This is attributable to a git commit (23ed601) that merely changes a typo in a function name (from *explicitly_convertable* to *explicitly_convertible*) without altering the functionality of any code. However, this commit modifies the source code of `llvm/ADT/iterator_range.h`, a header file used by almost all other files. For example, it is referenced by LLVM's String and BFS visiting implementations. Consequently, this causes a substantial regression in the CCache hit rate. Although these files must be recompiled, the modification doesn't affect the hash value of the original functions (as the stateful compiler doesn't hash function names). Thus, these compilations benefit from the stateful compiler.

High CCache miss rate and Low stateful compiler speedup: For the commits between c97c2 and 1db14 in OpenCV, a large amount of code requires recompilation (69.57% CCache miss rate). The changes in this code are so significant that they make it difficult for the stateful compiler to utilize the profiling data effectively. In this scenario, the stateful compiler behaves much like a conventional stateless compiler and achieves only a negligible speedup (0.67%).

Low CCache miss rate and High stateful compiler speedup: In the OpenCV project, between commits bf06b and 64be9, there are only minor modifications to the source code, resulting in just 12 targets requiring recompilation. As a result, the

CCache miss rate is as low as 0.73%. Despite this, the overhead to rebuild these 12 targets is quite large, involving files with substantial size (e.g., *opencl_kernel_dnn.cpp* contains around 5.5k lines). Since most of the source code in these files remains unchanged, the stateful compiler is able to accelerate the compilation process for these large files, leading to a 9.26% speedup in the end-to-end build time.

Low CCache miss rate and Low stateful compiler speedup: In the LLVM project, the commits between db32d and 899c8 introduce only minor changes to the C++ source code. As a result, the CCache miss rate is low at 0.87%, and only 5 targets require recompilation. However, there are 22 linking commands following the compilation. Consequently, the majority of the build time is spent on linking rather than compiling. Therefore, the stateful compiler has a minimal impact on the end-to-end build time, achieving only a 0.63% speedup.

In summary, the relationship between the CCache miss rate and the speedup achieved by the stateful compiler largely depends on the overhead associated with recompiling the missed targets, as well as the overhead of other parts of the build process (e.g., the linking process). However, generally speaking, employing the stateful compiler alongside CCache can consistently benefit the incremental build process. On average, compared to using CCache alone, the combination of CCache and the stateful compiler yields a 3.50% and 3.45% speedup for OpenCV and LLVM, respectively.

VI. Related Works

Incremental build is a common practice in software development, where similar code bases are frequently recompiled with different patches. Therefore, leveraging build history to accelerate incremental building is crucial. Instrumentation represents another scenario that stands to benefit from incremental building. Odin [34] is a framework that performs on-the-fly instrumentation, targeting only the code sections of interest for instrumentation. To expedite the process of incremental compilation when adding instrumented instructions, Odin subdivides the program into several smaller modules. As a result, rather than recompiling the entire program, only the affected modules need to be recompiled.

A. Build System

Researchers have developed various build systems to achieve high-performance incremental builds. These systems aim to avoid the compilation of unchanged objects through dependency analysis. GNU Make [32], one of the earliest and most widely known build systems, uses a declarative approach where users define dependencies and actions in a Makefile. Make tracks the timestamp of each file to identify those that have not been updated, thus avoiding their recompilation. While Make is simple and widely supported, its dependency-tracking mechanism can lead to unnecessary recompilation and longer build times for complex projects. Ninja [33] is another popular build system. Compared with Make, Ninja offers higher performance and better scalability in dependency analysis. It maintains an explicit dependencies graph, thereby avoiding redundant dependency analyses during incremental builds. Bazel [30] focuses on distributed build processes. It employs remote caching to deliver fast and reproducible builds, making it well-suited for complex projects on cloud servers.

For all build systems, the most critical task is to analyze dependencies among compilation targets. OMake [13] extends the Make build system to enhance its dependency-tracking mechanism. Pluto [11] provides a cooperative environment of builders that inform the build system the required and produced files during runtime, thus enables dynamic dependency analysis that can be utilized for making incremental build dynamically. [18] builds the dynamical dependency analysis that accepts file changes and transitively detects the affected build tasks for execution/re-execution during runtime. Shake [23] provides more flexible dependency descriptions, capable of handling dependencies involving files generated at build time. Riker [8] is a build system that automatically discovers dependencies to make the build process both correct and efficient.

B. Compiler wrapper

Most build systems are designed to accommodate multiple frontend languages and therefore do not perform specific analyses for source code semantics. Given the popularity of C/C++ in software projects, there are several projects aimed at addressing the incremental compilation problem specifically for C/C++ projects. CCache [31] serves as a compiler wrapper for C/C++. Before invoking the real compiler (e.g., gcc, clang), the wrapper checks whether the flags and input files have previously been built. If they have, the wrapper fetches the stored output files and returns them, avoiding the need to call the real compiler. Clcache [24] is a lightweight Python script designed for Microsoft Visual Studio that implements a similar mechanism. Distcc [10] takes this approach a step further by distributing the mechanism, allowing for the sharing of cached object files among users.

These projects typically use a hash function (e.g., checksum) to treat C/C++ source code as plain text for the purpose of detecting modifications. This approach does not utilize the semantic information in the programming languages. ABC [37] employs def/use analysis to detect the usage of header files. Consequently, if a function in a header file undergoes a change, only the files that actually make use of the changed function need to be recompiled, not all files that import the header. CHash [9] proposes an alternative hash function based on AST analysis, ensuring that non-semantic modifications [12] (e.g., code formatting) do not alter the hash value.

These works apply **coarse-grained** incremental build analysis at the source-file level, which does not utilize information at the function or loop levels. Montana [16], [27] proposes splitting large code files into several smaller ones to expedite the incremental build process. This approach can be considered fine-grained, where modifications at the function level are translated to the source-file level, assuming each function is contained in its own file.

These solutions concentrate on the semantics of C++ and maintain independence from specific compilers. In contrast, our approach necessitates modifications to the compiler. By delving into the compiler, it offers more opportunities for optimization. For example, if a user adds dummy instructions (e.g., ‘a = a‘) within a function, none of the previous projects can recompile the corresponding file by leveraging information from a previous build. Conversely, our solution can expedite the compilation process: after the dummy instructions are optimized out during compilation (e.g., dead code elimination), the stateful compiler can recognize that the same function has been compiled before and leverage previous building results.

Zapcc [1] is another stateful compiler. Zapcc treats the compiler and compilation process as a client-server architecture. Thus, the compiler is always an active process and can use memory to record the history of previous compilations. Compared with our solution, Zapcc utilizes memory instead of storage to keep cache data, which can potentially yield higher performance. Zapcc is an ambitious project and represents a heavily modified version of Clang, which poses challenges in keeping pace with the latest LLVM versions. As indicated on its GitHub project, Zapcc's last merge with LLVM 6 occurred in 2018. On the other hand, our approach is lightweight, involving only updates to the LLVM PassManager and the addition of a new plugin for Clang. This allows for more seamless integration with the latest LLVM versions. Additionally, while Zapcc focuses on frontend optimization to reduce the overhead of template instantiations and parsing headers, our solution is centered on middle-end optimization.

PASH-JIT [15] is a compiler that parallelizes POSIX script fragments. It utilizes a stateful compilation server to detect independent, parallelizable shell-script sections during runtime.

VII. Discussion

A. The Decay of Dormant Profiling Data

In the current design, dormant profiling data is not updated during the incremental build process. Consequently, as development progresses, the dormant profiling data decays, leading to the stateful compiler gaining fewer advantages as it detects fewer dormant passes. While it is feasible to update the dormant profiling data on-the-fly during the incremental build process, the size of the dormant profiling data would increase monotonically with each build, thereby slowing down the query time for the dormant status. To maintain a stable size of the dormant data, the stateful compiler also needs to discard any dormant data associated with IRs that have been modified, since this data will no longer be utilized. However, this additional step introduces extra overhead that can decelerate the compilation process. A potential solution could involve enabling the stateful compiler to track the hit rate of dormant data and to automatically regenerate profiling data when the hit rate falls below a certain threshold.

B. The Validation of Dormant Profiling Data

In our implementation, the compiler records IRs, flags, and pass names as dormant profiling data. However, given the complexity of modern compilers, there are additional factors that also affect the dormant status.

Optimization Pipeline: The optimization pipelines pass flags to passes implicitly. For instance, in LLVM, the `LoopUnswitch` pass is applied with both `O2` and `O3` optimization levels. However, the `NonTrivial` flag is enabled only at the `O3` level.

PGO Files: The PGO files supply metadata that influences the behavior of passes. For instance, these files offer insights into which code sections are 'hot' or 'cold', information that can significantly affect pass decisions.

Outer-Level IRs: LLVM permits an IR pass to be influenced by outer-level IR analyses. For example, certain function passes depend on module-level analysis. Consider a module containing two functions, A and B. Altering function B may affect the results of the module analysis, which could, in turn, change the dormant status when the function pass is applied to unchanged function A, since it relies on the module analysis outcome.

The most straightforward way to address these challenges is to record all relevant factors in the dormant profiling data. However, considering the complexity of modern compilers, detecting and recording all factors involves considerable effort.

A trade-off between performance and engineering effort is to exclude passes that involve PGO or higher-level IR analysis from the dormant profiling data. Consequently, during incremental builds, the compiler will not find any related dormant information for these passes and will apply them as traditional compilers do.

It is worth noting that **as long as compiler passes do not change the semantics of programs**, the omission of these factors in the profiling data results only in the violation of binary equivalence, while still maintaining semantic equivalence.

VIII. Conclusion

To accelerate the incremental builds, developers employ build systems to avoid recompiling unchanged files. For the changed files, build systems invoke compilers to generate new object files from scratch. This approach, which operates at the source-file level, is referred to as coarse-grained incremental build. In this paper, we introduce a stateful compiler designed to further expedite the incremental builds. Specifically, the stateful compiler retains dormant information and utilizes it to skip dormant passes during incremental builds. This approach facilitates a fine-grained incremental build process that leverages previous information at the code-section level.

We implement our approach using the LLVM compiler and evaluated its performance on real C++ projects. Our evaluation shows that the stateful compiler can improve the end-to-end incremental build process by 6.72%. Moreover, we integrate the stateful compiler with a state-of-the-art incremental build solution, CCache, and test it on two large C++ codebases. The results indicate that the stateful compiler can further accelerate the build process already optimized by CCache.

Acknowledgments

The authors are indebted to the CGO reviewers and the shepherd of this paper for their many useful suggestions and comments. The authors also wish to thank Vivek Sarkar for his comments on earlier drafts of this paper.

References

[1] "Zapcc," 2018. [Online]. Available: https://github.com/yrnkrn/zapcc

[2] "Cpython," 2023. [Online]. Available: https://github.com/python/cpython

[3] "Linux kernel," 2023. [Online]. Available: https://github.com/torvalds/linux

[4] "Lua," 2023. [Online]. Available: https://github.com/lua/lua.git

[5] "Mozilla firefox," 2023. [Online]. Available: https://www.mozilla.org/en-US/firefox/

[6] "Postgres," 2023. [Online]. Available: https://github.com/postgres/postgres

[7] G. Bradski, "The opencv library." *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[8] C. Curtsinger and D. W. Barowy, "Riker:{Always-Correct} and fast incremental builds from simple specifications," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 885–898.

[9] C. Dietrich, V. Rothberg, L. Füracker, A. Ziegler, and D. Lohmann, "{cHash}: Detection of redundant compilations via {AST} hashing," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 527–538.

[10] distcc Team, "distcc," 2021. [Online]. Available: https://www.distcc.org/

[11] S. Erdweg, M. Lichter, and M. Weiel, "A sound and optimal incremental build system with dynamic dependencies," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, J. Aldrich and P. Eugster, Eds. ACM, 2015, pp. 89–106. [Online]. Available: https://doi.org/10.1145/2814270.2814316

[12] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–38, 2021.

[13] J. Hickey and A. Nogin, "Omake: Designing a scalable build process," in *Fundamental Approaches to Software Engineering: 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings 9*. Springer, 2006, pp. 63–78.

[14] T. Jayatilaka, H. Ueno, G. Georgakoudis, E. Park, and J. Doerfert, "Towards compile-time-reducing compiler optimization selection via machine learning," in *50th International Conference on Parallel Processing Workshop*, 2021, pp. 1–6.

[15] K. Kallas, T. Mustafa, J. Bielak, D. Karnikis, T. H. Dang, M. Greenberg, and N. Vasilakis, "Practically correct,{Just-in-Time} shell script parallelization," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 769–785.

[16] M. Karasick, "The architecture of montana: an open and extensible programming environment with an incremental c++ compiler," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 6, pp. 131–142, 1998.

[17] D. A. Kim and S.-K. Lee, "Efficient change detection in tree-structured data," in *Web and Communication Technologies and Internet-Related Social Issues—HSI 2003: Second International Conference on Human. Society@ Internet Seoul, Korea, June 18–20, 2003 Proceedings 2*. Springer, 2003, pp. 675–681.

[18] G. Konat, S. Erdweg, and E. Visser, "Scalable incremental building with dynamic task dependencies," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 76–86. [Online]. Available: https://doi.org/10.1145/3238147.3238196

[19] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast searches for effective optimization phase sequences," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 171–182, 2004.

[20] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Practical exhaustive optimization phase order exploration and evaluation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 6, no. 1, pp. 1–36, 2009.

[21] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[22] G. Maudoux and K. Mens, "Bringing incremental builds to continuous integration," in *Proc. 10th Seminar Series Advanced Techniques & Tools for Software Evolution*, 2017, pp. 1–6.

[23] N. Mitchell, "Shake before building: Replacing make with haskell," *ACM SIGPLAN Notices*, vol. 47, no. 9, pp. 55–66, 2012.

[24] F. Raabe, "clcache," 2019. [Online]. Available: https://github.com/frerich/clcache

[25] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher, "Towards incremental build of software configurations," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2022, pp. 101–105.

[26] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher, "Pyrobuilds: Enabling efficient exploration of linux configuration space with incremental build," 2023.

[27] D. Soroker, M. Karasick, J. Barton, and D. Streeter, "Extension mechanisms in montana," in *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*. IEEE, 1997, pp. 119–128.

[28] D. Spinellis, "Git," *IEEE software*, vol. 29, no. 3, pp. 100–101, 2012.

[29] S. Tatikonda and S. Parthasarathy, "Hashing tree-structured data: Methods and applications," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 2010, pp. 429–440.

[30] B. Team, "Bazel," 2023. [Online]. Available: https://bazel.build/

[31] C. D. Team, "Ccache," 2023. [Online]. Available: https://ccache.dev/

[32] G. Team, "Gnu make," 2023. [Online]. Available: https://www.gnu.org/software/make/

[33] N. Team, "Ninja," 2023. [Online]. Available: https://ninja-build.org/

[34] M. Wang, J. Liang, C. Zhou, Z. Wu, X. Xu, and Y. Jiang, "Odin: on-demand instrumentation with on-the-fly recompilation," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 1010–1024.

[35] Z. Xu, L. Niu, J. Ji, and Q. Li, "Structure-preserving hashing for tree-structured data," *Signal, Image and Video Processing*, vol. 16, no. 8, pp. 2045–2053, 2022.

[36] R. Yang, P. Kalnis, and A. K. Tung, "Similarity evaluation on tree-structured data," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 754–765.

[37] Y. Zhang, Y. Jiang, C. Xu, X. Ma, and P. Yu, "Abc: Accelerated building of c/c++ projects," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 182–189.