*c*Hash: Detection of Redundant Compilations via AST Hashing

Christian Dietrich*, Valentin Rothberg*, Ludwig Füracker+,
Andreas Ziegler+ and Daniel Lohmann*

+Friedrich-Alexander Universität Erlangen-Nürnberg
*Leibniz Universität Hannover

Abstract

Software projects that use a compiled language are built hundreds of thousands of times during their lifespan. Hence, the compiler is invoked over and over again on an incrementally changing source base. As previous work has shown, up to 97 percent of these invocations are redundant and do not lead to an altered compilation result. In order to avoid such redundant builds, many developers use caching tools that are based on textual hashing of the source files. However, these tools fail in the presence of modifications that leave the compilation result unchanged. Especially for C projects, where module-interface definitions are imported textually with the C preprocessor, modifications to header files lead to many redundant compilations.

In this paper, we present the *c*Hash approach and compiler extension to quickly detect modifications on the language level that will not lead to a changed compilation result. By calculating a hash over the abstract syntax tree, we achieve a high precision at comparatively low costs. While *c*Hash is light-weight and build system agnostic, it can cancel 80 percent of all compiler invocations early and reduce the build-time of incremental builds by up to 51 percent. In comparison to the state-of-the-art CCache tool, *c*Hash is at least 30 percent more precise in detecting redundant compilations.

1 Introduction

Software development for a project that uses a compiled language involves a (seemingly) endless number of compiler invocations. Typically, a developer edits some source files, builds the whole project, and then tests and debugs the resulting binary. In this process, which is repeated tens to hundreds of times a day by thousands of developers, the time taken for the (noninteractive) build step is a crucial property to developer productivity [23].

After many incremental modifications, software developers typically commit their changes into a larger project-wide repository. From there, the robots of a continuous integration platform might pull and merge them to perform automated build tests, which involves some additional thousand builds of the software. A prominent example is Linux and the Intel 0-day robot.[1] The robot monitors more than 600 development repositories to run tests on newly integrated changes, for which it builds more than 36 000 Linux kernels on an average day in order to provide kernel developers with quick feedback on integration issues. Again, the time of each build is a crucial property for the effectiveness of the system – the more builds it can handle each day, the more build tests can be performed.

1.1 Redundant Builds

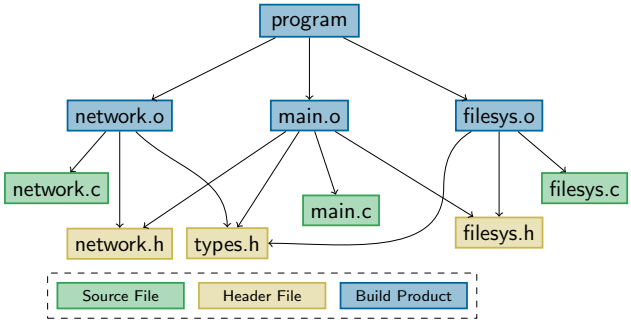
In both settings, the build process itself can often be performed as an *incremental build*: Compilation is generally considered to be an idempotent operation. Hence, only the source modules that are affected by a change or commit – either directly or transitively via a dependency – need to be recompiled into object files, while a large portion of unchanged object files can be reused from a previous build. In larger projects, build times thereby are reduced from hours and minutes for a full build to seconds and milliseconds for an incremental build.

The challenge here is to detect – in a reliable but fast manner – which source modules are part of the increment that needs to be recompiled. Ideally, a module becomes part of the increment only if its recompilation would lead to a program with a different *behavior* – which is undecidable in the general sense. Therefore, we technically reduce this to the decision if recompilation would lead to a different program *binary*. The respective test needs to be reliable in that it never produces a false negative (i.e., excludes some source module from the increment that is affected by a change). False positives do not harm reliability, but lead to costly *redundant builds* – which we wanted

[1] https://lwn.net/Articles/514278/

```
objects = network.o main.o filesys.o

program: $(objects)
        cc -o program $(objects)

main.o: main.c types.h network.h filesys.h
        cc -o main.o -c main.c

network.o: network.c network.h types.h
        cc -o network.o -c network.c

filesys.o: filesys.c filesys.h types.h
        cc -o filesys.o -c filesys.c
```

(a) Makefile describing the build process and its dependencies.



(b) The corresponding Make-internal build-dependency graph.

Figure 1: A minimal example of a software project written in C and built with Make.

to avoid with incremental building in the first place. However, the test itself also has to be fast – it gets executed for every source module on every build. If the overhead to decide which source modules are part of the increment outweighs the cost of false positives, incremental building also becomes pointless. In practice, the trade-off between precision and overhead is tricky – precision often does not pay off [1]. On the other hand, Zhang et al. [34] showed that with the common timestamp-based tests performed by Make [6] and other build systems, up to 97 percent of calls to the compiler for C/C++ projects are unnecessary and have to be considered as redundant builds.

1.2 About This Paper

We present *c*Hash, an approach and compiler extension for the Clang C compiler to quickly detect if a source module is affected by some change and needs to be recompiled. Our approach combines speed and precision by analyzing the effect of a change on the level of the (hashed) abstract syntax tree (AST) of the program. Compared to existing state-of-the-art techniques, such as CCache, we can significantly reduce the number of false positives (by 48.18 %) at only moderate extra costs, resulting in up to 23.16 percent shorter build times for incremental builds. In particular, we claim the following contributions:

- Efficient and reliable detection of semantic source-code changes on the language level.
- Open-source implementation of the *c*Hash concept as a plugin for Clang C compiler.
- Detailed evaluation on six open-source projects and comparison with the state-of-the-art CCache tool.

The remainder of this paper is structured as follows. In Section 2 we analyze the problem of redundant builds with a special focus on C projects. In Section 3, we describe the *c*Hash approach and discuss its implementation briefly in Section 4. We evaluate *c*Hash on a set of six open-source projects in Section 5 and discuss the previous work in Section 6. Besides a discussion of our results, we also elaborate on possible threats to the validity of our findings in Section 7 and conclude the paper in Section 8.

2 Problem Analysis

All modern build systems, whether they are implemented in Make [6] or use a more sophisticated toolchain [9, 4], try to reduce the number of redundant builds in order to achieve fast incremental rebuilds. However, the employed mechanisms often fail to detect non-essential changes precisely. For example, if a developer updates the modification timestamp of the file `convolute.h`[2] from the CPython source-code repository, the build system takes 15.9 s to rebuild the entire project on our server described in Section 5.3, about half of the time that is required for a fresh build. In this scenario, all build operations were redundant, so we should not have spent time on invoking the compiler at all. With *c*Hash, we can cut down the rebuild time in this particular case to 0.72 s, a decrease of 95.5 percent.

2.1 Modular Decomposition

Incremental rebuilds are enabled by the decomposition of software into modules, which is already around since the 1970s [18]. While modules are a necessary means for the separation of concerns on the logic level, they are also often physically separated into different files. Integral to modular decomposition is the export and import of interfaces to define whether others can use a particular field or data type and if they can invoke a specific functionality. For incremental builds, modularization entails the advantage that an interface is logically split into declaration, implementation, and invocation of the interface. Hence, an invoking module is only required to be recompiled if

[2] Full path: `Modules/_decimal/libmpdec/convolute.h`

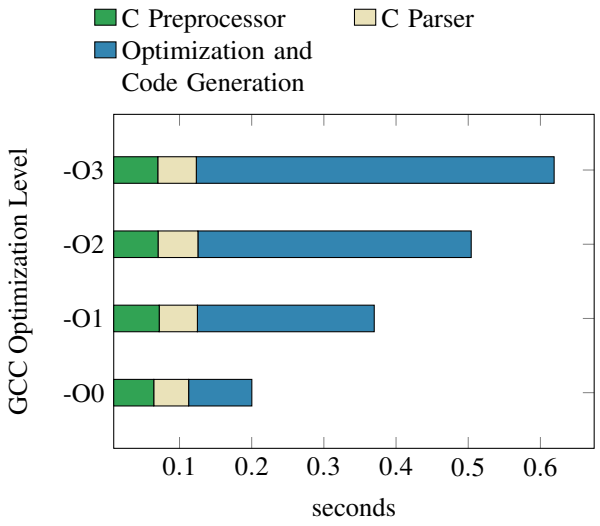


Figure 2: Average run time of the compiler phases. We compiled CPython with GCC on different optimization levels (-O0: no optimization, -O3: heavy optimization) and recorded the run time of the different compiler phases (n=850 compiler invocations per variant).

the declaration of an imported and used interface has been changed, which avoids many sources of redundant and costly recompilation [33, 30, 28].

While many languages, like Haskell or Rust, have built-in module support, the widely used C programming language lacks this feature. In C projects, modules are implemented purely idiomatically by means of the C preprocessor (CPP) and the file system. Importing another module's interface is realized via the `#include` directive of the CPP, which textually replaces `#include` directives with the content of the included file. Exporting an interface is realized via the file system by explicitly exposing the declarations (i.e., interfaces) in the corresponding header. Consequently, module dependencies in C can only be defined on granularity of files.

2.2 Build Systems and Dependencies

Since C and C-like programming languages are widely spread, their file-system–level implementation of modules heavily influenced build systems. For example, Figure 1 depicts the structure of a typical software project written in C and its build system implemented in Make [6]. Logically, the program is decomposed into the three modules `main`, `network` and `filesys`. On the file-system level, the modules are further scattered across different source files: For the `network` module, the interface declaration is located in `network.h`, while the actual implementation lives in `network.c`. Furthermore, the `network` module also imports the `types.h` definition file. From these source-code artifacts, build system and compiler generate the object file `network.o`, which is finally linked into the executable *program* file.

The developer describes all build products, their dependencies, and the production rules in the Makefile (see Figure 1a). During the build process, Make parses the Makefile and internally builds the dependency graph (see Figure 1b). The dependency graph is traversed bottom up and for each node Make checks whether the production rule has to be executed. For a clean build, all products are missing and, therefore, all rules must be executed, while for an incremental build Make examines the modification timestamp to detect out-of-date build products.

2.3 Detecting Redundant Compilation

With a *timestamp-based* method to detect redundant builds, like employed by Make, the build system compares the modification timestamp of the dependencies and the (already present) build product. If any prerequisite is newer, the production rule is re-executed to generate an updated build product, which again leads to the rebuild of all dependent build products. While this mechanism is reliable and fast, it leads to many false positives, as it is insensitive to the actual file contents. Thus, updating a file's modification time stamp suffices to cause a (cascading) recompilation.

A perfectly precise build system would only schedule an object file for recompilation, if the production rule will yield an altered binary representation. Modifying a comment, for instance, has no effect on the binary since the CPP removes comments from the token stream before compilation. However, even the introduction of a new identifier, such as a type or constant, will have no effect on the binary if the new element is not referenced by the module. Nevertheless, for such an ideal recompilation predictor, the whole compilation process would have to be done to the full extent in the same environment (e.g., optimization level). Since that would bear no benefit, various heuristics are used instead in practice, which we describe as follows.

Let's assume a given source file *C* uses a function *func* declared in header *H*. There are several possible heuristics to decide if *C* must be recompiled, each addressing a different abstraction level of the source module:

(1) The *metadata* of *H* or *C* has been changed

(2) *H* or *C* has been changed *textually*

(3) *H* or *C* has been changed *syntactically*

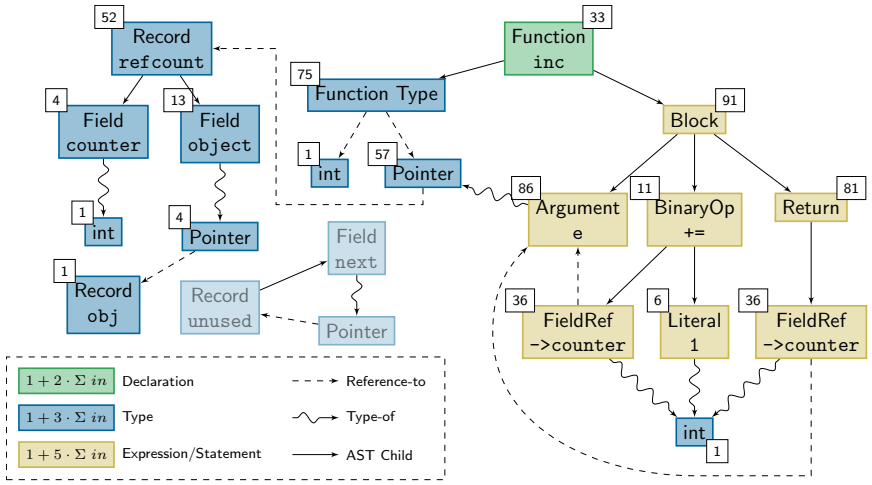
  (a) Syntactical change on the preprocessor level

  (b) Syntactical change on the language level

(4) The *declaration* of *func* in *H* has been changed

```c
struct unused {
    struct unused *next;
};

struct obj {};

struct refcount
{
    int counter;
    struct obj * ptr;
};

int
inc(struct refcount *e)
{
    e->counter += 1;
    return e->counter;
}
```

(a) C Source Code



(b) Simplified AST with annotated hash values.

Figure 3: The *c*Hash Approach. From the source module, the (standard) parser builds the abstract syntax tree and the semantic analysis establishes cross-tree references for types and variables. *c*Hash calculates a recursive hash over the AST to create a unique fingerprint of the program. In this example, we use three simplified hash rules depending only on the node class (represented by the node's color) and always take the modulo 100 from the result. Note that the `unused` record declaration is irrelevant for the resulting AST hash.

In this schema, Make and other build systems usually apply heuristic (1) by means of timestamp-based recompilation. In order to reach heuristic (2)-(4), we have to gather detailed information about the source module and its modifications. However, whether a more precise heuristic is desirable depends on the ratio of two run times: the time required to execute the redundant-build detection and the time saved for avoiding the compilation. A commonly used tool in this context is CCache [35], which uses heuristic (3a): CCache calculates a textual hash (i.e., MD4 hashes) over the preprocessed source code and uses this fingerprint as an index into an object-file cache of previous compilation results. The wide-spread adoption of CCache can be explained by the run-time proportions of the different compiler stages (see Figure 2). Compared to the whole C compiler invocation, the preprocessor takes up only 13.94 percent of the run time for the commonly used optimization level `-O2`.

In contrast to pure preprocessing, preprocessing and parsing takes only slightly longer (24.9 % of the whole invocation, `-O2`). Hence, with a more precise fingerprint of the parsed input we unlock potential higher build-time reductions for incremental compilation. In this paper, we present the *c*Hash approach, an incremental-build acceleration that applies heuristic (4) by means of hashing the abstract syntax tree within the compiler.

3 The *c*Hash Approach

Technically, *c*Hash operates similarly to the CCache tool. Both intercept the compilation process, calculate a hash value over the input, search in a cache for a previous compilation result associated with the same hash, and stop the compilation if the search was successful. Nevertheless, *c*Hash differs from CCache in two important regards: (1) *c*Hash calculates a hash over the abstract syntax tree (AST), while CCache hashes the preprocessed source code textually. (2) CCache only has to perform preprocessing, while *c*Hash must perform preprocessing, parsing, and semantic analysis. Since *c*Hash operates on the language-level instead of the CPP-syntactical level, we can easily avoid hashing of syntactic and semantic constructs (e.g., an additional declaration) that will surely not influence the compilation result. The underlying assumption is that the additional overheads of parsing and semantic analysis are only minor compared to the possible savings regarding redundant compilations, especially if a higher optimization level is chosen (see Figure 2) .

The abstract syntax tree is *the* central data structure compilers use during parsing and semantic analysis of a program. Its nodes represent the language entities (e.g., statements, expressions, types, . . .) that were identified by the parser, while the tree edges represent their syntactic nesting (e.g., statements within a function). After parsing, the semantic analysis checks the program for errors and introduces cross-tree references between semantically related nodes. For example, all variable-

definition nodes carry a reference to their respective type node. So, if we also consider these references, the AST effectively becomes a directed graph.

Figure 3a shows an example C source module with one function and three record definitions. Figure 3b depicts the corresponding (simplified) AST for the module (we omitted the root node and duplicated the `int` type). In our AST, three different classes of nodes are present: definition, statements/expression, and type nodes. For example, the function definition (`inc`) has a signature type, which itself references other types, and a compound block node that includes all statements from the function body. Furthermore, through the cross-tree references, cyclic structures can occur for recursive type definitions (see `struct unused`).

In a nutshell, *c*Hash operates directly within the compiler after the semantic analysis. In a depth-first search, we start from all top-level definitions and calculate a hash value over the semantically-enriched AST. For each node, we combine the hash values of all referenced nodes and all important node-local properties that influence the compilation into a new hash value. However, since we operate on a directed graph, nodes can be referenced more than once and two situations can arise: (1) If we have visited the node before and already calculated a hash value, we reuse it. (2) If we are currently visiting the node and encounter it again, we have detected a cycle and use a surrogate hash value instead to avoid an endless recursion. As a surrogate hash value, we use a textual representation of the type name in order to avoid collisions. This is necessary, since mutual referencing of types is possible:

```
struct x { struct y* link1; }
struct y { struct x* link2; }
```

If our surrogate value would be constant, the hash for the type `struct x` would be unchanged, if we make `link2` of type `struct *y`. In both ASTs, the depth-first search would visit the sequence (`struct x` $\rightarrow$ `link1` $\rightarrow$ `struct y` $\rightarrow$ *surrogate(link2)*). Therefore, the surrogate value must depend on the type of `link2`.

After the depth-first search, the hash is not only a fingerprint of the program semantics, but it also covers only elements that are reachable from the top-level definitions.

For illustration purposes, we executed a simplified version of the AST hashing (see Figure 3b) and annotated the intermediate hashes at the visited nodes. Here, we use a very simplistic hashing rule that incorporates only the node class (Declaration, Type, and Expression) as a node-local property. To keep the numbers small, we always calculate the modulo 100 of the result. For the top-level definition node (`inc`), the hash value calculates as follows: We add the hashes of all referenced nodes $(75+91)$, apply the node-class rule $(1+2*(166))$, and get a hash value of 33. However, not all nodes influenced this top-level hash: The record type `unused` and its children were never referenced and therefore are not covered by the top-level AST hash.

In practice, we have to be very careful when calculating the AST hash. If we omit an important property, two programs that are semantically different will end up having the same AST hash and, therefore, would be considered equal. Furthermore, we have to include all compiler flags that can influence the compilation result. In order to be on the safe side, we textually include all compiler flags into the top-level hash. If consistency between compiler upgrades is desired, we also must consider the compiler version. We also have to choose a sufficiently good hash function to avoid hash collisions. Since we are not defending against an evil attacker, we choose the efficient but non-cryptographic MurMur3 [2] hash function.

With the AST hash as a fingerprint of the source module, we can search for previous compilation results in a cache and abort the compilation process if we were successful, thereby avoiding the costly compiler phases of optimization and assembling. If the AST hash was not found, we continue the compilation process and copy the result to the cache for future invocations.

4 Implementation

We implemented the *c*Hash approach as a CLang [3] plugin for the C programming language. CLang is the C/C++/Objective-C/C++ front end of the LLVM [14] project. CLang only performs parsing and the semantic analysis of programs and hands the results, in form of LLVM intermediate representation (IR), over to LLVM for optimization and actual code generation.

The CLang plugin interface allows us to load a shared library into the compiler that is called during the compilation process. We instructed CLang to invoke *c*Hash after the semantic analysis, but right before the IR is generated and handed over to LLVM. At this point all cross-tree AST references are established, while optimizations and code generation are yet to come.

After an actual compilation, we store the AST hash for the source file and the object file for future compilations. On the next compiler invocation for the same source file, we calculate the AST hash again, compare it to the stored hash, and, in case of equality, hard link the last object file to the correct location and terminate CLang by calling `exit()`. The hard link is necessary as some build systems (CMake) remove the old object file before invoking the compiler. In contrast to the CCache tool, our current implementation stores only the last compilation result for every file instead of all previous ones. For an incremental application scenario, this has a minor influence if a change is reverted to a previous revision.

Project	Version	SLOC	Source Files		Build System	Opt. Level
			`.c`	`.h`		
LUA	5.3.4	18 k	35	26	Make	`-O2`
mbedTLS	2.4.1	56 k	123	83	CMake	`-O2`
musl	7597fc25	73 k	1322	16	Make	`-Os`
bash	4.4-p5	103 k	253	117	AutoConf	`-O2`
CPython	3.7-α1	403 k	324	325	AutoConf	`-O3`
PostgreSQL	e72059f	742 k	1152	747	AutoConf	`-O2`

Table 1: Summary of the evaluated C project repositories.

5 Experimental Results

For the evaluation, we validated the correctness of our implementation and quantified the influence of *c*Hash on the run time of incremental rebuilds. We chose six open-source projects to cover a wide range of possible application scenarios and applied *c*Hash in two typical usage scenarios. We compare our results to the CCache [35] tool (version 3.2.4) and quantify our absolute and relative prediction precision.

5.1 Evaluated Applications

We used a set of six real-world open source C projects (see Table 1) for our evaluation. This set of software projects covers a broad range of possible project properties, since they vary in size, application domain, and the employed build system.

The probed source-code bases range from small projects, like the LUA [11] language interpreter, which is mainly developed by one person, to large multi-decade, multi-person projects, like the CPython language interpreter [20]. They also differ in their usage of C language extensions: While some projects, like the musl C library [38], aim for portable and simple-structured code, others, like the mbedTLS SSL library [37] use compiler-specific features (e.g., vector types) to achieve a higher performance. The examined projects also develop at different speeds: While the PostgreSQL [19] repository lists 40 000 changes for 20 years of development, the bash command-line interpreter [7] reaches only 128 for the same period.

Furthermore, the projects employ different build systems: Small projects, like LUA and mbedTLS, often stick to plain GNU makefiles [8] and encode their dependencies manually. Larger projects often use configuration systems like CMake [4] or GNU AutoConf [9] that act as makefile generators. All examined build systems compare timestamps to detect compilation results that have to be rebuilt from the source files.

The musl C library deserves special mention, since their build system ignores some actual dependencies on purpose. Their manually encoded dependencies exclude all exported header files (515 files), since changing them would break the library's binary interface, which by definition is immutable. Hence, we exclude the public header files from our evaluation and treat them as unchangeable.

5.2 Validation of *c*Hash Implementation

As a first step, we validated the robustness of our *c*Hash implementation. In our targeted scenario, a robust implementation produces equal AST hashes for two inputs iff the compilation result is also equal.

For the validation, we built 2368 changes taken from the development history of the musl library independently. For every change and every object file, we recorded the AST hash, a textual hash of the object file, and a run-time report of the compiler-internal phases.

Over all examined changes, the compiler ran 5.68 million times and emitted 13 199 different object files. Our implementation proved to be correct and no AST hash was associated with more than one object file.

A perfectly precise predictor would exactly produce one fingerprint for every object file. For *c*Hash, we collected 55 829 different AST hashes over all changes, which results in a ratio of 1 object file : 4.23 AST hashes. Through manual investigation, we traced back 55.3 percent of the AST hashes to only 16 changes. While some of them included a major source-code reorganization, 11 changes caused a new AST hash for every source file, since they modified the compiler flags (e.g., enabled a new warning). Without these compiler-configuration changes, the ratio of object files and fingerprints drops to 1 : 2.5.

From the collected data, we could also confirm that the run-time impact of *c*Hash is minimal. On a 16-core Intel i7-2600 @ 3.4 Ghz, *c*Hash needed 9 ms on average for the AST-hash calculation. In comparison, the parser took an average 187 ms, while the rest of the compilation (i.e., optimizer, assembler) executed in 1082 ms.

5.3 Rebuild with Minimal Changes

Our first end-to-end evaluation resembles a typical scenario for an individual developer. In an already built working copy of the source code (all build products are up to date), the developer makes a *minimal change* and instructs the build system to update all products. The duration of this rebuild cycle is critical for the developer, since it is often hundreds of time a day.

We used two methods to introduce an artificial minimal change to a single file: (1) As *the* most minimal possible change, we set the modification timestamp to the current time to mimic the editor's save command. (2) To mimic a minimal textual change, we introduce a `#line 1` directive to the beginning of the file, which is left alone by the CPP and has no influence on the debug information. While both modification are surely artificial, they will

			Updated Timestamp			Textual Change		
Project	Initial Build	#files	Baseline	CCache	*c*Hash	Baseline	CCache	*c*Hash
LUA	2.03 s	61	1.09 s	−67.3 %	−59.5 %	1.10 s	16.4 %	−59.6 %
mbedTLS	3.57 s	204	1.33 s	−24.1 %	−4.1 %	1.33 s	18.9 %	−4.3 %
musl	14.29 s	1338	0.86 s	−20.6 %	−4.5 %	0.86 s	17.6 %	−4.7 %
bash	6.06 s	370	1.49 s	−70.9 %	−65.8 %	1.48 s	−9.2 %	−65.3 %
CPython	34.30 s	649	8.16 s	−77.7 %	−63.7 %	8.22 s	−24.7 %	−64.1 %
PostgreSQL	61.35 s	1891	3.16 s	−65.3 %	−42.2 %	3.12 s	8.6 %	−41.8 %

Table 2: Average rebuild duration after a minimal change. In a built working copy of the examined project, we repeated a modify–rebuild cycle for every file and measured the duration of the recompilation. Baseline shows the arithmetic average over the rebuild times, relative percentages are in respect to the baseline (n=#files).

result in an unchanged object file. Therefore, this is a best-case scenario for recompilation avoidance, since all compiler invocations are actually redundant.

We repeated the modify–rebuild cycle for every source file (headers included), measured the required rebuild time on a 48-core AMD Opteron 6180 system with 64GB of memory running Ubuntu 16.04.1, and calculated the arithmetic-average rebuild time. While our test system is an older server system, its performance is comparable to a modern developer work station. For all experiments, we instructed the build system to utilize all cores (`make -j48`). Since the initial build was done just before the actual experiment, all files were served from the main memory and disk contention was no issue.

The results are listed in Table 2. As an orientation, we also measured the time to build the project from a fresh checkout after the build system is set up. The second column holds the number of files for which the modify–rebuild cycle was executed.

For updated timestamps, CCache outperforms *c*Hash in all cases, since it only has to invoke the preprocessor and skips all subsequent compilation steps. In contrast, *c*Hash must at least wait for the parser and the semantic analysis to start calculating the AST hash.

If we introduce a textual change, CCache cannot detect the redundant rebuild and the build-time improvement diminishes or even turns negative, since CCache still has to pay the cost of maintaining the object cache. For the modification scenario, the improvements for *c*Hash remain stable and we achieve a maximum rebuild-time reduction of −65.3 percent for bash.

However, for two projects (mbedTLS and musl), we see a much smaller influence of CCache and almost no improvement by *c*Hash. Both projects have a very sparse dependency structure where a change to a source file often leads only to a single compiler invocation, while the majority of the time is spent in the linking process.

5.4 Rebuild with Commit-Sized Changes

Our second evaluation scenario resembles a usage pattern that is found in continuous-integration systems. Source-code changes are uploaded to a build server and automatically integrated into the mainline repository. For every incremental change, the build server verifies that the source code compiles and informs the developers about compilation errors. This scenario is distinct from the former one, since an uploaded change reflects the condensed editing effort of a single developer over a time period.

For this evaluation, we selected the last 500 (127 for bash) non-merge changes from the source-code repositories of the examined projects. We excluded all changes that were broken in the original repository and failed to compile. For every change, we set up the working copy to the previous (parent) change and built the project as a starting point. After applying the change, we measured the recompilation time on the same 48-core Opteron that was used for the previous scenario and calculate the arithmetic average over all non-failing changes. We repeated the evaluation for the unmodified baseline build system, CCache, *c*Hash, and a combined variant (CCache+*c*Hash). We recorded the build times, as well as the number of detected redundant builds, which we will call "hits" for brevity. The summarized results can be found in Table 3.

Our largest improvement for a single change occurred in the change `90d3da11c9` in PostgreSQL that fixes a spelling mistake in a comment located in a central header. Normally, the rebuilding of this change takes 15.6 s. While CCache correctly identifies the situation, its cache-maintenance overhead keeps the recompilation time at 3.5 s. With the compiler-internal approach of *c*Hash, we only require 2 s (−87.4 %) to rebuild.

Over all projects, and all examined changes, *c*Hash aborted the compilation in 79.75 percent of all invocations and decreased the average build time by −29.64 percent. For CPython, we even achieved an improvement of more than 50 percent. In contrast to that, CCache has a much

	Changes		Baseline		CCache		*c*Hash		CCache + *c*Hash	
	OK	Fail	Time	#Invoc.	Time	#hits	Time	#hits	Time	#hits
LUA	479	21	2.14 s	16765	−38.8 %	13761	−49.3 %	15748	−46.7 %	15748
mbedTLS	498	2	2.13 s	36654	−20.7 %	24124	−7.3 %	25750	−21.6 %	26764
musl	500	0	1.25 s	28655	−3.8 %	19587	0.7 %	19457	−3.2 %	23104
bash	108	19	2.88 s	1931	−11 %	326	−22.7 %	1281	−16 %	1354
CPython	500	0	8.27 s	20338	−46.4 %	14551	−51.4 %	19102	−53.7 %	17859
PostgreSQL	498	2	5.63 s	25934	−11 %	7184	−31.6 %	22209	−25.3 %	20909

Table 3: Rebuild time for the last 500 non-merge changes. For every change, we prepared a fully built working copy with the previous (parent) change. After applying the change, we measure the rebuild duration, as well as the number of detected redundant build operations. For the baseline, we give the arithmetic average over the time required to build one change and the number of compiler invocations. For the modified build processes, we give the change in average build time (n=#OK changes) and the accumulated numbers of detected redundant builds (#hits, higher is better).

lower hit ratio (61.05 %) and could decrease the average rebuild time by only −23.63 percent. From the total number of hits, we can quantify that *c*Hash's semantic fingerprint is at least 30.19 percent more precise than CCache's textual one.

Similar to the results of the previous scenario (Section 5.3), we see little influence of *c*Hash on the rebuild times of musl and mbedTLS. For musl, we even have a small decrease in performance due to its dependency structure. Since the developers intentionally omitted almost all dependencies on header files, the majority of compiler invocations is caused by a modified source file, which almost always involves a semantic modification.

We also combined CCache and *c*Hash by chaining their execution: First, CCache searches for the textual hash and, if unsuccessful, hands over the preprocessed code to *c*Hash. In the last two columns of Table 3, we see the result of the combined experiment. However, in our implementation, CCache interferes with the operation of *c*Hash such that the hit number decreases (CPython, PostgreSQL): CCache includes a deeper history of previous builds. If a compilation is aborted by CCache due to its cache, *c*Hash is not invoked and can, therefore, not fill its shallow cache, which is only one object file deep. If the respective source file is then modified in a way that the textual hash changes but the AST hash remains the same, *c*Hash is not able to detect the redundant build because its cache is empty. Nevertheless, the combination of CCache and *c*Hash comes close or even exceeds the best result of both methods if applied in isolation. With a combined caching strategy, this interference between CCache and *c*Hash could be avoided.

6 Related Work

As building software is an important part of the development process, attempts to reduce the build time are numerous and focus on different aspects and phases of the process. Since multiple C/C++ compilation units can be built independently, the process can be distributed over several machines. The free `distcc` [36] tool acts as a compiler wrapper and sends the preprocessed source code over the network for remote compilation. Microsoft's in-house build service CloudBuild [5] employs the same technique and distributes 20000 builds per day on up to 10000 machines and attaches to various build systems. CONCORD [21], which is Microsoft's internal alternative to CloudBuild, also uses distributed builds and speeds up the Windows build process by up to 100 times. Google's build infrastructure [12] relies on reproducible builds and distributes the work over thousands of machines.

A too coarse-grained module structure often leads to redundant builds if one of the central "God" modules are touched. Therefore, Yu, Dayani-Fard, and Mylopoulos [33] proposed a technique to refactor large header files into multiple smaller ones and thereby achieved a speedup of the compilation by nearly 32 percent. However, with higher optimization levels their speedup dropped to 12 percent. Furthermore, automatic restructuring of headers can be in conflict with the developers' intentions.

Morgenthaler et al. [17] proposed the CLIPPER tool, which automatically finds build system targets that are too coarse-grained and aids the developer in removing them. Vakilian et al. [30] examined build-system dependencies and found that nearly 50 percent of the 40000 build targets of a Google internal Java library are too coarse-grained and can be further refined. However, Miller [16] discusses that incomplete dependency graphs, which can stem from the usage of recursive GNU make systems, can yield incorrect compilation results and render incremental builds useless. Developers then often fall back to compile the software always from scratch.

Besides general build-system organization, several researchers proposed techniques to speed up the compiler invocation itself; *c*Hash being one of them. Often these propositions focus on a single compiler phase and are composable. Pre-compiled headers are an old technique

that was already used for Mesa [10, 26] and NeXT [15] and is still employed in industry [13]. For this technique, header files are translated to an intermediate format which then can be loaded faster by the compiler for all subsequent invocations.

The CCache [35] tool intercepts the compilation after the preprocessor and calculates a textual hash over the preprocessed code to detect redundant builds. For distributed build services the problem of redundant builds becomes especially severe, since many developers start compile jobs for very similar code bases. Therefore, both Microsoft and Google use textual hashing in their build services [32, 5]. However, the employed hashing method is orthogonal to the caching method and *c*Hash could act as a drop-in replacement for the textual hashes in these systems. Actually, Google's build system allows the integration of language-specific hashing methods.

For languages with a module system, Tichy [29] proposed the smart recompilation approach, which was extended by Schwanke and Kaiser [22] and Shao and Appel [24]. For smart recompilation, each exported module interface is annotated with a version stamp. Dependent modules only have to be recompiled if one of their imported interfaces has an updated version stamp. By incorporating only referenced declarations, *c*Hash achieves the same effect for languages without a proper module system. Furthermore, *c*Hash does not only include the called function signatures, but also ignores all syntactic changes that are removed by preprocessor and parser.

Zhang et al. [34] introduce the ABC tool to generate an additional unoptimized object file for each compilation unit. For each compilation unit, ABC invokes the compiler without optimizations and aborts the subsequent, but more expensive compilation process with optimizations if the unoptimized object remains unchanged. However, in case of a redundant build, *c*Hash pays only the price for parsing, while ABC has to finish the compilation.

Whaley [31] uses dynamic execution profiles to determine rarely executed code regions in Java programs, which can either be excluded from optimization or entirely from compilation. During the execution, the program falls back to using unoptimized code or even an interpreter solution. Suganuma, Yasue, and Nakatani [25] used a similar approach for dynamic compilation of Java software by focusing optimization efforts only on non-rare code paths.

7 Discussion

As we have shown in the evaluation, *c*Hash provides a significant speed up for realistic usage scenarios that occur during the development of software. In this section we want to discuss threats to the validity of our results, benefits, and give hints for future work.

7.1 Threats to Validity

One threat to the validity of our experimental results is the selection of software projects we used to evaluate the effects of *c*Hash. If their code organization and/or their change-recording policies were highly favorable for *c*Hash, our results would be overly optimistic. For example, if a project had one central header file that is updated in every single change to the repository, the time savings *c*Hash induces would be optimal. However, as discussed in Section 5.1, the chosen projects cover a broad range of properties and we have not encountered such a pattern. Furthermore, our evaluation scenarios would yield totally different results for a project with this pattern.

Another threat to our experimental validity is our implementation of *c*Hash. Although we have rigorously validated our implementation (see Section 5.2) it is not verified formally. However, the examination of 500 changes from several open-source projects and over 2000 changes for musl makes us confident in the reliability of our implementation.

Currently, we implemented the *c*Hash only for the C programming language. However, the general concept is suitable for every language that can be expressed as an AST with cross-tree references, even if it includes cyclic references. If a programming language cannot be expressed as such a structure, *c*Hash cannot be applied. One prominent example is the TEX programming language: In TEX, the program flow can influence and feedback data back to the lexer and, therefore, the actual structure becomes only visible during execution. However, such languages are rare and will only be executed in an interpreter.

Furthermore, *c*Hash is only usefully applicable if the compilation process is dominated by the middle- and back-end (optimizer, code generation) and the front end is considerably fast. However, since most modern languages offer a more expressive semantic than C (e.g., Haskell, Rust), the efficient code generation and the optimizations take longer. Therefore, we are confident that *c*Hash will perform even better for these languages than for C.

Another general impediment for a wide-ranged adoption of *c*Hash is its requirement to access internal compiler data structures. If a compiler does not provide an appropriate plugin structure or is developed as a closed-source project, *c*Hash cannot be applied. Furthermore, the AST hashing must be implemented for every programming language and for every compiler, a general implementation does not exist. However, C and C-style languages are still the most prominent compiled languages [27] and both widely used free-software compiler suites (clang, GCC) include a powerful plugin interface.

7.2 Advantages of *c*Hash

Besides the apparent benefits of faster rebuilds, *c*Hash is also build-system agnostic. Similar to the widely-adopted CCache tool, *c*Hash does not require modifications on the build-system level. We, therefore, think our unintrusive approach fosters a wide-spread adoption of *c*Hash. We demonstrated this property in the evaluation, where three different build systems were handled without any modification (see Section 5).

During the compilation process, the compiler always builds an abstract syntax tree of the program and holds it in memory. This availability allows *c*Hash to be a lightweight and self-contained mechanism that is easy to test in isolation. Furthermore, the calculation of an AST hash is computationally cheap, since only one depth-first search graph-traversal is required to calculate it. During the compilation process, such traversals are already executed dozens of times.

Since the AST is a semantic representation of the program, *c*Hash is able to detect various changes that do not lead to a changed compilation result. First of all, many syntactic modifications, like comments, whitespace-changes, or even the presence of braces are not present on the AST level.

Besides the syntactical modifications, *c*Hash is also able to ignore language-semantic changes to the compilation unit. Per default, we already ignore unused declarations and type definitions, which leads to a fine-grained dependency tracking on the symbol and the type level. As mentioned in Section 6, this property of *c*Hash brings the benefits of smart recompilation [29] to the C programming language, which in other respects lacks any module support. As C projects handle modules on the file-system level, the whole interface definition of another module is included if the header file is referenced (`#include`). With *c*Hash, we narrow this import down to the actually used interfaces and consequentially detect dependencies between modules more precisely.

7.3 AST Hash Precision

The predictive power of the AST hash in regard to detecting redundant builds is determined by two factors: (1) Which AST nodes are considered during the depth-first search. (2) What attributes of the visited nodes are included into the hash. Currently, *c*Hash is conservative in both dimensions in order to avoid false-negative compiler abortions.

On the node-selection part, *c*Hash currently ignores all AST nodes which are not referenced, directly or indirectly, from the top-level definitions of a compilation unit. However, with a more complex and compiler-aware strategy, *c*Hash could also ignore other AST nodes that will not lead to changed object file (e.g., defined but unused functions marked as `static`). A normalization step – like sorting the order of local-variable definitions – is also possible.

In regard to the AST-node fields, we ignore only fields that are known to not influence the resulting code, like origin line numbers in the source code (if correct debugging information is desired, this information should be included). In order to increase the predictive power, we could furthermore exclude variable and type-name fields. With a more relaxed equivalence relation for object files, we could additionally exclude modifiers (e.g., `inline`) if they are known to have no effect on the resulting program behavior. However, every introduction of in-depth compiler knowledge increases the complexity of the hashing mechanism, which, most probably, would make *c*Hash more fragile, especially in terms of changes in future compiler versions.

7.4 Future Work

We see several directions of future work: In order to allow co-evolution of the implementation and to do more validation, we will work on integrating *c*Hash into the CLang mainline repository. This effort also includes the integration of the *c*Hash approach into other open-source compilers (i.e., gcc).

Furthermore, we plan the implementation of the *c*Hash approach for more complex languages. As a first target, we will extend our CLang plugin to the C++ programming language. There, the usage of templates and their location in header files promises huge savings by *c*Hash.

Another direction of research is the possibility of *c*Hash to provide more fine-grained information about changed definitions and language constructs. With *c*Hash, a compiler can not only detect that the whole compilation process is redundant, but also that the compilation of a single function can be skipped. For such a partial-compilation scheme, we would start the depth-first search at the function level instead of the AST's root node.

8 Conclusion

The detection of redundant builds, which can increase the throughput of the development-testing cycle significantly, is a trade-off between precision and cost. In this paper we show how this trade-off can be optimized towards higher precision at low costs by applying language-level analyses directly in the compiler. The results for our *c*Hash approach show that on average 80 percent of all compiler invocations can already be canceled after the semantic analysis. For single projects, we speed-up the recompilation process by up to 51 percent, while single changes

even compiled up to 87 percent faster. In comparison to the state-of-the-art CCache tool, *c*Hash's AST hash fingerprinting is over 30 percent more precise.

Acknowledgments

The authors thank the anonymous reviewers and our shepherd Theodore Ts'o for their feedback. This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/3-1 and SFB/Transregio 89 "Invasive Computing" (Project C1).

The source code of cHash and the raw data for this paper are available at:
`https://gitlab.cs.fau.de/chash`

References

[1] Rolf Adams, Walter Tichy, and Annette Weinert. "The Cost of Selective Recompilation and Environment Processing". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (1994). DOI: 10.1145/174634.174637.

[2] Austin Appleby. *SMHasher: a testsuite for distribution, collission, and performance properties of non-cryptographic hash function*. https://github.com/aappleby/smhasher, accessed 6 Feb. 2017.

[3] *CLang: a C language family frontend for LLVM*. http://clang.llvm.org, accessed 7. Feb 2017.

[4] *CMake*. http://cmake.org, accessed 7. Feb 2017.

[5] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. "CloudBuild: Microsoft's Distributed and Caching Build Service". In: *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*. 2016. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889222.

[6] Stuart I. Feldman. "Make — A program for maintaining computer programs". In: *Software: Practice and experience* 9.4 (1979), pp. 255–265.

[7] *GNU Bourne Again SHell*. http://www.gnu.org/software/bash/, Git repository at http://savannah.gnu.org/projects/bash/, accessed 7. Feb 2017.

[8] *GNU Make*. http://www.gnu.org/software/make, accessed 7. Feb 2017.

[9] *GNU autoconf*. https://www.gnu.org/software/autoconf/autoconf.html, accessed 7. Feb 2017.

[10] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. "Early Experience with Mesa". In: *SIGSOFT Softw. Eng. Notes* 2.2 (Mar. 1977), pp. 138–. ISSN: 0163-5948. DOI: 10.1145/390019.808320.

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. "Lua—An Extensible Extension Language". In: *Software: Practice and Experience* 26.6 (1996), pp. 635–652. ISSN: 1097-024X.

[12] Christian Kemper. *Build in the Cloud: How the Build System works*. http://google-engtools.blogspot.de/2011/08/build-in-cloud-how-build-system-works.html, accessed 7. Feb 2017. [Online; posted 18-08-2011].

[13] Tara Krishnaswamy. "Automatic Precompiled Headers: Speeding Up C++ Application Build Times". In: *Proceedings of the 1st Conference on Industrial Experiences with Systems Software (WIESS)*. USENIX Association, 2000.

[14] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. (Palo Alto, CA, USA). 2004.

[15] Andy Litman. "An Implementation of Precompiled Headers". In: *Software – Practice and Experience* 23 (1993).

[16] Peter Miller. "Recursive Make Considered Harmful". In: *AUUGN Journal of AUUG Inc* (1998).

[17] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. "Searching for Build Debt: Experiences Managing Technical Debt at Google". In: *Proceedings of the Third International Workshop on Managing Technical Debt*. 2012.

[18] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* (1972).

[19] *PostgreSQL*. http://postgresql.org, Git repository at http://github.com/postgres/postgres, accessed 7. Feb 2017.

[20] *Python*. http://python.org, Git mirror at http://github.com/python/cpython, accessed 7. Feb 2017.

[21] Wolfram Schulte. "Changing Microsoft's Build: Revolution or Evolution". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. 2016. ISBN: 978-1-4503-3845-5.

[22] Robert W. Schwanke and Gail E. Kaiser. “Smarter Recompilation”. In: *ACM Trans. Program. Lang. Syst.* 10.4 (Oct. 1988), pp. 627–632. ISSN: 0164-0925. DOI: 10.1145/48022.214505.

[23] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. “Programmers’ build errors: a case study (at google)”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 724–734.

[24] Zhong Shao and Andrew W. Appel. “Smartest Recompilation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL. 1993. DOI: 10.1145/158511.158702.

[25] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. “A Region-based Compilation Technique for a Java Just-in-time Compiler”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. 2003. ISBN: 1-58113-662-5.

[26] Richard E Sweet. “The Mesa Programming Environment”. In: *ACM SIGPLAN Notices*. 1985.

[27] *TIOBE Index for January 2017*. http://www.tiobe.com/tiobe-index/, accessed 06 Feb. 2017.

[28] Alexandru Telea and Lucian Voinea. “A Tool for Optimizing the Build Performance of Large Software Code Bases”. In: *12th European Conference on Software Maintenance and Reengineering (CSMR)*. 2008.

[29] Walter F. Tichy. “Smart Recompilation”. In: *ACM Trans. Program. Lang. Syst.* 8.3 (June 1986), pp. 273–291. ISSN: 0164-0925. DOI: 10.1145/5956.5959. URL: http://doi.acm.org/10.1145/5956.5959.

[30] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. “Automated Decomposition of Build Targets”. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. 2015. ISBN: 978-1-4799-1934-5.

[31] John Whaley. “Partial Method Compilation Using Dynamic Profile Information”. In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2001. ISBN: 1-58113-335-9.

[32] Nathan York. *Build in the Cloud: Distributing Build Steps*. http://google-engtools.blogspot.de/2011/09/build-in-cloud-distributing-build-steps.html, accessed 7. Feb 2017. [Online; posted 23-09-2011].

[33] Yijun Yu, Homy Dayani-Fard, and John Mylopoulos. “Removing False Code Dependencies to Speedup Software Build Processes”. In: *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 2003.

[34] Ying Zhang, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Ping Yu. “ABC: Accelerated Building of C/C++ Projects”. In: *Asia-Pacific Software Engineering Conference (APSEC)*. 2015.

[35] *ccache – a fast C/C++ compiler cache*. http://ccache.samba.org/, accessed 7. Feb 2017.

[36] *distcc – a free distributed C/C++ compiler system*. http://github.com/distcc/distcc, accessed 7. Feb 2017.

[37] *mbedTLS*. http://tls.mbed.org, accessed 7. Feb 2017.

[38] *musl libc*. http://www.musl-libc.org, accessed 7. Feb 2017.