Advanced Methods of Code Coverage Calculation in ClickHouse Using LLVM Compiler Infrastructure

Mike Kot
Faculty of Computer Science
Higher School of Economics
Moscow, Russia
to@myrrc.dev

***Abstract*—Code coverage is a technique that provides metrics showing which parts of the program source code are executed while running a specific set of tests. It helps to discover possibly error-prone places and to fix them. The task of building code coverage reports for projects with large codebases is non-trivial as we need to balance between execution speed and level of detail. The main goal of this paper is to develop a code coverage runtime (faster than the existing solution and producing per-test coverage data) for the ClickHouse database management system and integrate it into ClickHouse's CI pipeline.**

***Index Terms*—ClickHouse, code coverage, clang, test relevance**

I. Introduction

A. Acknowledgment

The author is grateful to his thesis supervisor Alexey Milovidov for his passionate participation. The author would also like to thank Elena Nikolaeva and Arina Stepanova for their continuous help and encouragement.

B. Glossary

- `std::unordered_map<Key, Value>` is later referred to as `hashtable<Key, Value>`.
- `std::unordered_set<Value>` is later referred to as `hashtable<Value>`.
- Address *symbolization* is a process of obtaining source file path and line for an assembler instruction located at a given address. Code that performs this task is called a *symbolizer*.
- Name *demangling* is a process of converting a function name from an architecture-dependent *mangled* form (used as an assembler label [1]in a resulting binary) to its original *unmangled* form as written in a source file. Name *mangling* is the opposite.
- A *basic block* is a "straight-line code sequence with no branches in except to the entry and no branches out except at the exit" [1]. Code coverage tools usually instrument the code in terms of basic blocks as every line in the basic block would be executed if the first line has been.
- A *Control flow graph* (CFG) or a *basic block graph* is a representation of the program's execution paths state machine with respect to conditional and unconditional branching. Basic blocks represent vertices, branches represent edges.
- An *edge* or an *arc* is an edge in program's CFG.
- A *hit* means that some edge was executed while running a test. Also, *hit* or *hits* may be referred to as a number of times the edge was executed, for example, "five hits".
- *GIMPLE* [2] and *LLVM IR* [3] are low-level intermediate representations used by compilers. GIMPLE is used by `gcc` [4] (GNU), LLVM IR is used by `clang` [5] (LLVM). Both can be converted to a CFG form.
- A *sanitizer callback* is a data-accumulating function. Calls of this function are inserted into the program's intermediate representation during the compilation process. Usually, the developer is responsible for implementing the callback.
- *Code coverage runtime*, referred to as *coverage runtime* or simply *runtime*, is a set of sanitizer callbacks and classes aimed at solving code coverage task for a particular binary.
- A *submodule* or a *contrib* is an external library used by ClickHouse [6]. The first refers to a mechanism used to store such libraries (`git submodules`), the second – to a folder `contrib/` where libraries are stored.

C. Problem statement

While every project has errors in its source code that have not been discovered yet, the overall cost of fixing such errors in projects with large [2] codebases becomes extremely high.

Usually one needs to:

1) Find an error and create the corresponding issue/pull request to indicate the problem being worked on.
2) Fix the error.
3) Run the CI pipeline to ensure the error has been fixed.
4) Backport [3] the solution to older versions of the program.

Please note that the last step cannot be completely automated, so one may also need to involve engineers from the core team that can merge the solution.

[1]Unlike C, C++ allows function overloads (functions with different arguments but the same name), thus function name alone cannot be used as an assembler label. Therefore, additional function information: namespace (if present), class (if the function is a member function), arguments and return type as well as template arguments are used to obtain a mangled name that is unique for every function

[2]Containing more than one million lines of own code

[3]Most corporate clients use the so-called LTS (long term support) versions that include the patches created during the support period

Any optimization that could reduce the number of errors or provide early detection would save core developers' time and, ultimately, money.

D. Motivation

- Current coverage task runs about 9 hours (being the slowest in CI pipeline). The proposed solution lowers this time to about 3 hours thus making it possible to run it for every commit.
- Current coverage task generates an accumulated HTML report for all tests. Per-test coverage data are not preserved. The proposed solution provides a way to view per-test coverage data, which can be used to obtain tests that are the most relevant for a source file.
- The proposed solution describes a script that, having been given a `diff` for a commit and a coverage report for a previous commit, can reorder the tests according to their failure probability. Testing scripts can use this information to reduce testing time drastically. Developers' time is also saved as they get failed tests immediately instead of waiting for all to finish.
- The proposed solution describes another script providing a `diff`-like utility for two coverage reports. Such information may be used to force (in an automated way) the developer proposing changes to ClickHouse to write tests for the proposed changes (so the overall coverage metric would not worsen).

E. Goals

The main goals of this paper are to:

1) Create a runtime that accumulates code coverage data on a per-test basis, stores it efficiently, and converts it to format that can be further visualized on demand.
2) Research various storage formats, compression algorithms, code coverage variants, IPCs, and options provided by the infrastructure to achieve the best build time, external memory and space consumption, and performance compared to the clean ClickHouse server parameters.
3) Briefly describe tools operating on coverage reports, namely, tests reorderer and incremental coverage checker.
4) Integrate all the above tools into the Yandex ClickHouse CI pipeline.

The proposed runtime should not give a measurable slowdown. The data produced while testing should not exceed the disk quota ($\approx$ 30GB).

F. Structure

The "Code coverage software overview" section contains various solution types overview. The "Solutions provided by GNU compiler infrastructure" and "Solutions provided by LLVM compiler infrastructure" sections describe in detail infrastructure provided by compilers. The "Current solution" section describes the current solution. The "Proposed solution" section describes the developed solution and its details. The "Applications of CCR usage" section describes various scripts that can further improve the CI using artifacts of main coverage build runs. The "Results" section summarizes the anticipated results in order of their relevance. The "Conclusion" section gives an overview of the paper.

II. Code coverage software overview

Code coverage task itself may be viewed as a task of building a bijection between the set of program's own source code lines and a metric (for example, "binary – is line covered or not", "N – how many tests cover this line").

A. Provided by compilers

Examples: GNU coverage sanitizer and source coverage with gcov, LLVM SanitizerCoverage and SourceBasedCodeCoverage with lcov.

Such solutions operate on compilers' intermediate representations. Compilers find places that should be instrumented (typically, finding a minimal spanning tree for the program's CFG) and insert calls to sanitizer callbacks there.

The developer is responsible for implementing callbacks (that should store data and write it to disk after binary has been tested).

B. Tools extending infrastructure provided by compilers

Examples: covtool, llcov.

They work similarly to the above solutions but act as compiler plug-ins mostly, extending provided functionality or improving build/testing time. Such solutions may implement better spanning tree finding algorithm, richer data accumulation metrics, compression for data stored in the binary itself, or introduce other optimizations.

C. Tools operating on information already contained in the binary

Example: kcov.

The previous two solution types modify the program's intermediate representation while compiling (so one needs the program's source code to use these). Tools like `kcov` operate only on the information that is contained in debug sections in a given binary. These are useful in general when sources are unavailable but are considered slow and featureless in comparison with the above.

D. Unit-testing frameworks

Examples: Google.Test, Catch2, including all CI pipelines using these like GitHub.Actions.

These tools mostly rely on tests written in the same language as the project itself, so they do not solve code coverage task in terms of "CFG edges coverage", but are useful in tracking functions coverage, namely "Is some function covered?". Function in such terms is thought to be covered if there exists a test calling this function.

E. Sampling software

Such tools may work similar to `perf`: once in a specified interval (for example, 5000 times per second) a script would obtain ClickHouse's binary current execution state (instruction pointer and all of its symbolized data) and accumulate statistics. Nonetheless, issues with tracking overall coverage percent would arise due to lack of CFG information.

F. Using calculated metrics (Clickhouse-specific)

ClickHouse calculates a wide range of metrics (instantly calculated metrics, periodically calculated metrics, events). These metrics could be used to track, for example, functions coverage.

G. Requirement-based coverage (Clickhouse-specific)

In such coverage type each test would need to specify which ClickHouse features it would attempt to use. Implemented in the Testflows CI check. However, one would need to rewrite all 2500 stateless tests according to Testflows standard.

The vast majority of use cases are solutions provided by compilers, so they are described in detail.

III. Solutions provided by GNU compiler infrastructure

`gcc` offers two main ways of tracking code coverage:

- The simplest (in terms of efforts) pipeline that does not require writing any code. The runtime provides reports that can be further visualized. Let's call it the *Source-based* pipeline by analogy to the LLVM (covered later).
- A more customized (though harder to implement) approach using user-defined callbacks. Let's call it the *SanitizerCoverage* pipeline.

A. Source-based pipeline

This pipeline consists of 3 steps:

- Building the binary with `gcc`'s profiling flags (and running the program itself).
- Indexing and merging reports to a single report with `lcov`.
- Visualizing the report using `genhtml`.

1) Finding places to insert instrumentation code: For each function

- Split cycles of abnormal [4] nodes by adding fake edges between function's entry and node's source and between node's destination and function's exit to keep the graph acyclic.
- Add fake exit edges for each `call` and `asm` statement since they may not return.
- Sort edges according to their frequencies [5]
- Call the subroutine calculating the spanning tree.
- Call the subroutine inserting an edge profiler for every critical [6] edge in the spanning tree (covered in III-A2).

Routine that finds the spanning tree for a function:

- Add fake exit edge between function's end and start [7].
- Add edges that point to function's end edge so we would not insert profiling code after function **`return`** statements.
- Add fake edges so we could instrument, for example, contents of a **`while`**`(true)` block.
- Add abnormal edges. Those are added as they cannot be split into multiple nodes due to their "undefined nature".
- Add calls to edges pointing to a non-local transition (exceptions, `setjmp/longjmp`, non-local `gotos`).
- Add all the rest edges. Edge list was previously sorted according to frequencies so we would produce the minimal spanning tree [8].

2) Inserting the instrumentation code:

- We build (listing 1) a GIMPLE node of an integral type with value 1 (which will be added to the counter).
- If counter update must be made atomically, we create a call to an intrinsic function (`f`, its exact type is determined by the node type) using the created pointer to the edge `edgeno` (legit as each edge has its unique number), created "one" node, and `MEMORY_MODEL_RELAXED` as we do not care about reordering.

```
__atomic_fetch_add(
  &counter, 1, MEMMODEL_RELAXED);
```

- If we do not care about atomicity [9], we perform a read-modify-write operation with a temporary variable [10].

```
long counter = counter_ref;
counter += one;
counter_ref = counter;
```

[4] Those which "cannot be instrumented at the moment"

[5] Expected number of branch's executions scaled by the edge's hit probability. The frequency itself is used in many places

[6] Edge is thought to be critical if both of its nodes have more than one edge, in other words, this edge is not the only one connected to each of its nodes

[7] We do not insert nodes that may form cycles as we will be unable to count hits

[8] We will not duplicate the edges as node's `on_tree` flag is checked while the groups are unionized

[9] User can choose between counter update types (atomic, atomic preferred, non-atomic)

[10] We cannot just insert the `cnt += 1`-alike node due to the fact that GIMPLE satisfies SSA form, in which each variable must be assigned exactly once

```
void profiler(int edgeno, edge e) {
  tree one = build_int_cst(
    gcov_type_node, 1);
  if (flag_profile_update == ATOMIC) {
    tree addr = tree_coverage_counter_addr(
      GCOV_COUNTER_ARCS, edgeno);
    tree f = builtin_decl_explicit(
      ATOMIC_FETCH_ADD);
    gcall *stmt = gimple_build_call(
      f, 3, addr, one,
      build_int_cst(integer_type_node,
        MEMMODEL_RELAXED));
    insert_on_edge(e, stmt);
  } else {
    tree ref = tree_coverage_counter_ref(
      GCOV_COUNTER_ARCS, edgeno);
    tree gcov_type_tmp_var =
      make_temp_ssa_name(
        gcov_type_node, NULL, "counter");
    gassign *stmt1 = gimple_build_assign(
      gcov_type_tmp_var, ref);
    gcov_type_tmp_var =
      make_temp_ssa_name(
        gcov_type_node, NULL, "counter");
    gassign *stmt2 = gimple_build_assign(
      gcov_type_tmp_var, PLUS_EXPR,
      gimple_assign_lhs (stmt1), one);
    gassign *stmt3 = gimple_build_assign(
      unshare_expr (ref),
      gimple_assign_lhs(stmt2));
    insert_on_edge(e, stmt1);
    insert_on_edge(e, stmt2);
    insert_on_edge(e, stmt3);
    }
}
```

Listing 1: Simplified GIMPLE edge profiler

B. *Source-based pipeline example*

Consider the following C program (listing 2). Its CFG can be obtained with `-fdump-tree-cfg-raw` (listing 3).

```
int foo(int a, int b) {
  return a > b ? a : b;
}

int main() {
    int a, b;
    scanf("%d %d", &a, &b);
    printf("%d", foo(a, b));
}
```

Listing 2: A simple program

```
foo (int a, int b)
{
  int D.2883;

  <bb 2> :
  gimple_cond <gt_expr,
    a, b, NULL, NULL>
    goto <bb 3>; [INV]
  else
    goto <bb 4>; [INV]

  <bb 3> :
  gimple_assign <parm_decl,
    D.2883, a, NULL, NULL>
  goto <bb 5>; [INV]

  <bb 4> :
  gimple_assign <parm_decl,
    D.2883, b, NULL, NULL>

  <bb 5> :
gimple_label <<L2>>
  gimple_return <D.2883 NULL>
}
```

Listing 3: Program's CFG for `foo`

Although being quite intuitive, this representation can be visualized with graphViz (fig. 1). `gcc` can directly dump representation to `.dot` files with `-fdump-tree-all-graph`.

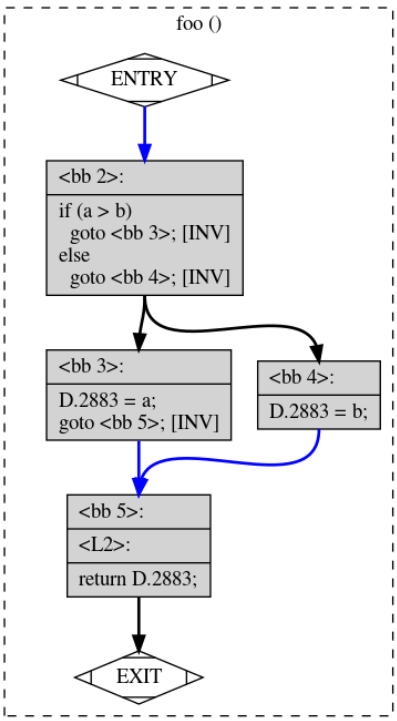

Fig. 1. Visualised program's CFG for `foo`

As you may see in listing 4, the profiler added two calls to counter increments: the first one when the program takes an `if` branch, the other – when `else` branch is executed as both edges are critical.

```
foo(int, int):
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
+   movq    __gcov0.foo(%rip), %rax
+   addq    $1, %rax
+   movq    %rax, __gcov0.foo(%rip)
    movl    -4(%rbp), %eax
    cmpl    -8(%rbp), %eax
    jle     .L2
+   movq    __gcov0.foo+8(%rip), %rax
+   addq    $1, %rax
+   movq    %rax, __gcov0.foo+8(%rip)
    movl    -4(%rbp), %eax
    jmp     .L3
.L2:
    movl    -8(%rbp), %eax
.L3:
    popq    %rbp
    ret
```

Listing 4: Without instrumentation / with `-fprofile-arcs` (highlighted)

By default, `gcc` stores counters data using static variables.

Please note that official documentation states that you should not access coverage files directly. That is the reason you cannot skip `gcov` run.

`-ftest-coverage` works similar to the former option but produces `.gcno` files [11] containing information for `gcov` (instrumented lines, function data, source files data).

This option is separated from the former one as `-fprofile-arcs` may be also used for profile-guided performance optimizations.

C. SanitizerCoverage pipeline

SanitizerCoverage pipeline consists of two steps:

- Writing callbacks that process actions of executing a critical edge or a function.
- Building the program along with these callbacks.

Relevant flags:

- `-fsanitize-coverage=trace-pc` inserts a call to `__sanitizer_cov_trace_pc()` into every critical edge.
- `-finstrument-functions` inserts callbacks into every function start and end.
- `-finstrument-functions-exclude-...` provide blocklists for instrumentation. Function exclude list operates on unmangled names.

[11] `.gcno` are obtained at compile time

Sanitizer coverage instrumentation is implemented via a separate pass.

The implementation is pretty straightforward (get GIMPLE tree representation for function, insert the callback) as it was in the Source-based pipeline.

D. SanitizerCoverage pipeline example

```
extern "C" void
[[gnu::no_instrument_function]]
__sanitizer_cov_trace_pc() {
        printf("%p\n",
      __builtin_return_address(0));
}
```

Listing 5: An example callback implementation

```
foo(int, int):
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rbx
    subq    $24, %rsp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    call    __sanitizer_cov_trace_pc
    movl    -20(%rbp), %eax
    cmpl    -24(%rbp), %eax
    jle     .L2
    call    __sanitizer_cov_trace_pc
    movl    -20(%rbp), %ebx
    jmp     .L3
.L2:
    call    __sanitizer_cov_trace_pc
    movl    -24(%rbp), %ebx
.L3:
    call    __sanitizer_cov_trace_pc
    movl    %ebx, %eax
    movq    -8(%rbp), %rbx
    leave
    ret
```

Listing 6: Instrumented function from the previous section

Compiler, as explained before, emitted a callback invocation at every critical edge.

IV. Solutions provided by LLVM compiler infrastructure

`clang` has a similar implementation of both SourceBased and SanitizerCoverage pipelines (in its source code and in exposed API), so only the differences for the latter are highlighted.

Relevant flags:

- `-fsanitize-coverage=trace-pc` inserts a call to `__sanitizer_cov_trace_pc()` on every critical edge.
- `-fsanitize-coverage=trace-pc-guard` inserts a similar callback invocation (that takes an edge

index pointer as an argument) on every critical edge (see IV-A).
- `-fsanitize-coverage=pc-table` provides a way to obtain all instrumented addresses (including function entries) at runtime.
- `-fsanitize-coverage-{blocklist, allowlist}` provide a way to exclude source files or functions from instrumentation.

A. Level of detail

Unlike `gcc`, `clang` provides two additional levels of coverage detail. By default, `edge` mode is enabled (which is the only one available for gcc).

```
void foo(int *a) { if (a) *a = 0; }
```

Listing 7: A simple function

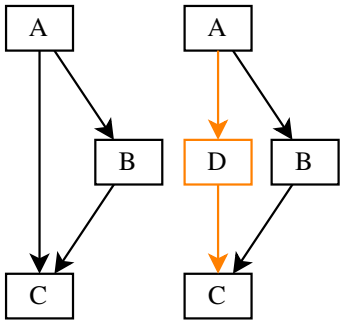

Fig. 2. Listing 7 CFGs

For an example function (listing 7) with CFG (fig. 2, on the left) if all blocks A, B, C were covered, the compiler cannot determine whether edge $A \to C$ was covered (as only edges $A \to B$ and $B \to C$ may have been executed). So it introduces another fake block D (fig. 2, on the right) and inserts instrumentation code to D (in addition to instrumenting A, B, and C).

If `bb` (basic blocks) mode is enabled, no fake blocks are introduced. Also, one may instrument only function entries with `func` option (`gcc` splits this option to `-finstrument-functions`).

B. PC table

```
void __sanitizer_cov_pcs_init(
  const uintptr_t*, const uintptr_t *)
```

Listing 8: PC table callback signature

PC-table is an experimental `clang` feature that provides a callback (listing 8) pointing to all instrumented addresses in binary. The array holds pairs of addresses and a flag "is this address a function entry" (as function entries may be processed separately). One can use this along with PC guards callbacks to get an address by its corresponding edge index.

C. PC guards

```
void __sanitizer_cov_trace_pc_guard_init(
  uint32_t *start, uint32_t *stop)
```

Listing 9: PC guard init callback

```
void __sanitizer_cov_trace_pc_guard(
  uint32_t *index)
```

Listing 10: PC guard callback

`clang` also provides a lightweight analog to `trace-pc` called *PC guards*.

The user has to implement two callbacks:

- The first (listing 9) receives an array of critical edges indices (from now on we presume that coverage sanitizer has `edge` option enabled). One can initialize indices to any non-zero value (for example, enumerating all indices from 1). Each index corresponds to an instrumented address.
- The second (listing 10) is invoked when a critical edge gets executed. Unlike `trace-pc`, the developer does not have to obtain function's return address as the address can be obtained from PC table. On `x86_64` it reduces the memory consumption to half of initial as `sizeof(void*)` (representing addresses) is two times bigger than `sizeof(uint32_t)` (representing edge index as stated by the compiler).

`gcc`, on the contrary, does not expose a public API to retrieve instrumented addresses. One would need to parse produced `.gcno` files.

D. Partially disabling instrumentation

Unlike `gcc`, `clang` does not have options to exclude files directly. It uses allowlist/blocklist mechanism instead.

Blocklist contains three types of items:

- Comments, starting with #, which are ignored.
- Source file item, starting with `src:`, which contains a path of a file that should be excluded.
- Function item, starting with `fun:`, which contains a name of a function that should be excluded.

The documentation states that you can also use the asterisk symbol $*$ to provide regular expressions. Asterisk will be expanded like in shell expansion. However, double expansion such as $*foo*$ works neither for functions nor for source files paths so one still needs to use mangled names or their prefixes.

There is another issue with mangled names' architectural dependency. On `x86_64`, for example, functions in `namespace std::` can have mangled prefixes `_ZNSt`, `_ZNKSt`, or `_ZNRSt` (as there is no standard mangling scheme, provided list is not complete) so one needs to write all possible options explicitly.

V. Current solution

Currently, ClickHouse uses `clang` SourceBased code coverage pipeline. Source code is compiled with appropriate flags and then the resulting binary is tested. After testing the binary an LLVM profile file (`.profraw`) is generated (in fact, multiple profile files are generated, for server and client, but they are merged into one).

After that `lcov` tool indexes the profile file and writes data back to disk (in `.info` format) where `genhtml` tool builds an HTML report out of it.

Coverage build is run once a day on the latest master commit. The report is displayed as an ordinary GitHub check.

VI. Proposed solution

A. Motivation for using compiler-provided tools

As *functional* (a pair of SQL request to server and desired answer) tests check the most of ClickHouse's functionality, the "unit-testing framework" solution is not applicable. As already said, tools that do not have access to source code are considered slow compared to the ones that do. Tools like `llcov` usually provide an out-of-the-box solution (which is untweakable), so the selected type is a solution bound to the compiler infrastructure.

Source-based tools provided by both supported compilers are also untweakable (in fact, both manual pages state they should be used "only if you need the visualization"), so SanitizerCoverage pipeline is going to be used.

B. Motivation for using clang instead of gcc

1) clang drawback in comparison with gcc: Inability to exclude files from instrumentation in a usable way. Allowlist does not solve the issue with headers (see VI-P). Moreover, using the mangled name is architecture-dependent.

2) gcc drawbacks in comparison with clang:

- Slower build time, worse binary size. `gcc` does not have ThinLTO, so contribs build slower, the resulting binary is about 0.5GB heavier.
- No PC table. If using `gcc`, one would need to operate on addresses instead of edge indices (that means using hashtables instead of vectors and no pre-caching of symbolized data). That introduces about an hour slowdown.

Coverage builds are running on `x86_64` Linux in CI, so using an architecture-dependent blocklist is fine. For speed and size reasons, `clang` was chosen.

C. Caches and data storage

Coverage runtime embraces various caches (all of which are read-only when tests start):

Filled when processing PC table:

- `vector<Addr>` cache of addresses (using edge indices as vector indices). PC table allows us to operate on edge indices instead of addresses which gives a substantial boost as edge indices may also be used as indices in other caches.
- `vector<bool>` "is address at index i a function entry". Function entries should be processed separately, as, unlike ordinary addresses, we also need to obtain their mangled names (ordinary addresses only need the source file and line).

Filled while merging symbolized data:

- `vector<SourceFileInfo>` of cached source files. By the time tests start we already know all instrumented source files (as we know all instrumented addresses from PC table and can symbolize each one), so we cache their data (absolute or relative source file path, instrumented function entries, and lines).
- `hashtable<Source file relative path, source file index>`

  Used when merging symbolized data.
- `vector<EdgeInfo>` of cached edges information. It is easier to unify information (such as source file, line index and optional function name, if the address is a function entry) into a single cache. In that case code writing data to disk can just invoke `operator[]` with edge index instead of branching before obtaining separate caches.

Runtime also embraces additional data structures:

- `vector<TestData>` holding information about each test:
  - Test name
  - Test index
  - `vector<SourceFile>` that contains two hashtables:
    - `hashtable<edge index, call count>` for instrumented functions. We cannot use a line as a key as multiple functions may get invoked in the same line, for example, `foo(bar(baz))`.
    - `hashtable<line, call count>` for instrumented lines. Multiple addresses may refer to the same line, for example, ternary operator `cond ? foo() : bar()` would produce two callbacks as it is an if-else block written in another form, but this information is useless for visualization purposes, so we ignore it.
- `vector<EdgeIndex>` holding data for a currently active test (such test that has started but has not finished yet).

D. Tasks queue

Coverage runtime uses a simple tasks queue (based on internal thread pool implementation). This tasks queue is used both in symbolization tasks (spawning all available threads) and in tests processing (using one external thread, acting more like a queue). It has

- A mutex.
- "job has finished" condition variable.
- "new job has been scheduled or class is shutting down" condition variable.
- A queue of scheduled jobs.
- A list of worker threads.

Jobs spawned are simply functions taking a `size_t` index of a worker thread running it. Functions

are stored in a standard type-erasing container `std::function<void(size_t)>`.

Tasks queue spawns N worker threads on startup (N is set in constructor).

First, obtaining an exclusive lock, each worker waits on a second condition variable until either the job queue is non-empty or an internal flag "we are shutting down" is set.

Then, if the job queue is non-empty, the worker pulls out the job, runs it, and notifies the first condition variable that the job has finished. The worker also updates an internal "scheduled jobs" counter which is used while waiting for all jobs to finish.

Tasks queue provides a way to wait until all currently scheduled jobs finish – it is done via waiting on the first condition variable until "scheduled jobs" is zero.

Unlike ClickHouse's ThreadPool, the tasks queue does not:

- free threads when no jobs are present (this saves a few comparisons on jobs schedule).
- use job priorities (all jobs are treated as equal, which allows using an ordinary queue instead of a priority queue).
- handle exceptions. Neither symbolization jobs nor test processing jobs are thought to throw, so no code handles this case.
- use metrics. Code coverage (as an external runtime) does not need to count its tasks in general metrics.

Also, some utility functions were made non-thread-safe (so this class is not intended for general use).

E. ClickHouse symbolizer

ClickHouse has an internal symbolizer called SymbolIndex. It acts as a faster replacement for `dladdr`.

For purposes of this thesis, we need to know only the fact that on startup (when instantiating the first SymbolIndex object) this class reads the ClickHouse binary as an ELF [12] file.

Coverage runtime on startup initializes a DWARF [13] reader from an ELF object. A utility function is provided that allows finding only desired information for a given address.

1) Obtaining a physical address out of a virtual one: ClickHouse binary gets loaded first, so (as it is built without PIC) its virtual offset is 0, and one can simply obtain the physical address by initializing `uintptr_t(virtual_addr)`.

2) Finding the .debug_info entry for a physical address: Done via looping over `.debug_aranges`, a section storing lookup table for mapping addresses to compilation units.

DWARF reader reads data in chunks.

Every chunk consists of:

- an aranges version.
- an address size variable (only `sizeof(uintptr_t)` addresses are supported).
- a segment size variable (only unsegmented architectures like `x86_64` are supported).
- padding according to DWARF standard.
- pairs (start, length) which represent address ranges.

If our given address fits in some range, we return chunk offset to the outer function.

3) Getting the compilation unit: First, we obtain a pointer to `.debug_info` section start and shift `offset` bytes from it.

Then we read section entry length (filling compilation unit's size) and fill other needed variables such as compilation unit's version, offset regarding the system machine word size (32 or 64), address sizes in the unit, and offset from section start.

4) Retrieving file and line information: Looping through all compilation unit attributes, we fill:

- Offset in `.debug_line` for the VM program compilation unit (`DW_AT_stmt_list`).
- Compilation directory (`DW_AT_comp_dir`).
- Compiled file name (`DW_AT_name`).
- Compilation unit base address (`DW_AT_low_pc` or `DW_AT_entry_pc`).

Then we get the main source file path (compilation directory concatenated with the compiled file name). Then we execute the LineNumberVM program with previously filled attributes to find the exact file and line [14].

LineNumberVM is an interpreter for line numbers bytecode VM. It has quite a complex algorithm so it is not going to be covered here.

F. Executing PC table callback

If we use non function entry address as is, SymbolIndex will not be able to find the line for our address.

```
; Instrumented address should
; be here (0x12dbca7a),
jmp    0x12dbca7f
; but is actually here (0x12dbca7f),
; where edge index is set for the callback
movabsq $0x15b4522c, %rax
addq   $0x4, %rax
movq   %rax, %rdi
; pc guard call
callq  0xb067180
```

Listing 11: Example of instrumented address range

The symbolizer (as well as `lldb` and `gdb` debuggers) thinks that instruction at `0x12dbca7f` (listing 11) is located at line 0 of the file.

So we need a way to get the previous instruction (`clang` uses internal architecture information to decrement the given address in default callbacks implementation). We can simply decrement given `uintptr_t` address [15].

[12] Executable and linkable format, the standard binary file format for Unix.

[13] Debugging data format

[14] DWARF reader also can search inlined functions, but for coverage build, ClickHouse's own source files are built in debug mode, so that code does not get executed

[15] The only reason for `uintptr_t` existence – integral operation on pointer types

G. Symbolizing instrumented data

When runtime has executed the PC table callback and filled two caches, it needs to symbolize all instrumented addresses and function entries to speed up test processing (addresses symbolization is the bottleneck here).

First, addresses are split into function entries and ordinary ones. It is better as in that case function spawning symbolization jobs can be made a function template with parameter "are we processing function entries". It allows writing less code.

Symbolization can be parallelized, so tasks queue spawns all available threads (`std::thread::hardware_concurrency`). During symbolization server does not do any other work (excluding lightweight tasks like updating DNS cache), so all CPU cores are dedicated to this job.

A worker thread can either

- Symbolize some addresses in a batch and then write them to global cache under exclusive lock, or
- Symbolize all provided addresses to local caches. Then main thread merges all local caches data into a global cache.

The second option was faster independent of batch size, so it was chosen.

Each worker thread processes a range of addresses. It symbolizes each address and writes obtained data into a local cache. For functions, it also looks up the function's mangled name.

Structures used:

- `LocalCaches<CacheItem>` is a vector (of thread pool size, hardware concurrency in our case) of `LocalCache<CacheItem>`. Each worker thread job, provided with a `size_t` worker index (see VI-D) writes to a separate `LocalCache` so no synchronization is needed.
- `LocalCache<CacheItem>` is a
  ```
  hashtable<Source relative path,
    CacheItem::container>
  ```
  It is templated as function entries and ordinary addresses need different containers.

The container used for addresses is a multi `hashtable<Line, EdgeIndex>` (that is, a hashtable allowing multiple values for a key).

Multiple addresses can be hit for a single line (the aforementioned ternary operator). We track line coverage, not address coverage, so we can hash local cache items by their unique lines. However, a test may register hits of both addresses, so we need to process all edge indices per line.

The container used for function entries is a vector of triples `(Line, EdgeIndex, std::string_view)`.

The function name is already contained in DWARF, so we do not need to perform any external memory allocations and can store a reference to its name in a string view. We cannot use a hashtable here as multiple functions may be invoked in the same line.

Another issue is with function templates: technically, multiple template instantiations for function `foo` are all invocations of `foo`, although they may do completely unrelated work depending on template arguments with C++17 `if constexpr` feature (or using SFINAE for older C++ versions). So they should all count hits for `foo`, but checking some names set being just template instantiations of a single function would need a demangler and some substring checking, so function templates are counted as separate functions.

H. Merging symbolized data

Data merging function loops through all caches filled by threads in the previous step.

They fill the utility

```
hashtable<Source file relative path,
  source file index>
```

from local caches hashtable.

For function entries, triples are directly written to the edges cache; for normal addresses they are inserted after checking for their single appearance in the `hashtable<line>` (as instrumented lines are stored in a vector, not in a hashtable, we need to check that no lines will be duplicated).

I. Why only one thread is used for tests processing

Each test runs at least 0.5-0.6 seconds, the whole coverage pipeline for this test takes about 0.1-0.2 seconds (including writing data to disk), so no more than one thread is needed.

J. Inter-process communication between ClickHouse server and tester

The testing script needs to notify the server about the test count and each test name. The server, in turn, needs to notify the tester about symbolization jobs end.

Solutions tried:

- Signals. Tester script sends a `SIGRTMIN + 1` signal with an attached integral value that represents the line in a test names file. The server, in turn, sends a `SIGUSR` signal when it has finished testing. The server processes signals from the tester script via `sigqueue`.
- Settings. Tester executes a `SET coverage_test_name='filename'` query after running every test. This special [16] setting is captured and a coverage runtime function is called. A bash script waits until a special message `Symbolized all addresses` appears in the log file and then starts the tester script.

The second solution was as fast as the first one but needed less code.

[16] Setting is not actually set as it is not used anywhere after, so we can save a bit of time

K. Starting and ending test coverage

When a callback is invoked, it means a test with a given name has finished.

Thus we allocate another vector of edge indices (of the same size with currently active one, filled with zeros) and swap it with the current one (it is the fastest way to get previous test data and clear the accumulation vector).

Then, a functor is created that moves the allocated vector (now containing data for the finished test) inside itself. This functor is scheduled in the tasks queue.

Special case: when testing ends, clickhouse-test executes a `SET coverage_test_name=''` query (the value is empty). When a callback processes this value, after scheduling a task it also calls a special `deinitRuntime` function which waits on all currently active jobs and finishes writing the report to disk.

L. Converting raw test data

When the scheduled job gets run, it resizes the given source files vector (already allocated in the tests global cache) to source files count.

Please note that tests global cache pre-allocates all TestData structures (test name, test index, two hashtables for hit functions and lines, and an internal logger pointer) for all tests as these structures do not take much space ($72B \times 2500tests \approxeq 175KB$) but save us time for zero-initializing in test processing functor.

However, we do not preallocate space for all source files as that invokes slowdown comparing to the solution where each test allocates space for its source files (slowdown is about 20 minutes for all tests).

Then the job iterates over all hits, skipping zero ones, and updates function and line hits. Only three indexing operations are performed:

- Obtaining hit for edge index.
- Obtaining edge info for edge index (to get edge line and determine whether it is a function entry).
- Updating or inserting data into internal hits hashtables.

First two can be optimized to just one-two assembler instructions as data start pointers and edge indices are known.

Then a function that writes data to disk in CCR format is called.

M. Writing data to disk

When processing test entries, we have three options of writing data to disk:

- Use a memory buffer with some reserved size, insert formatted data into it with help of `fmt::format_to(memory_buffer&)` and write everything to disk in the end.
- Use a pre-allocated buffer, resized to fit at least all desired data, and write to it with `fmt::format_to(Iter)`.
- Format individual lines and immediately write them to disk using `fmt::print(FILE*)`.

The first option resulted in $\approx$ 0.87s per test, the second took $\approx$ 0.83s per test, the third took $\approx$ 0.65s per test (tested on a subset of "fast tests" running no more than one second on a server without coverage). It is a decent approximation as running all 2500 tests with the third option (compared to the second one) lowered down overall execution time from 4h10m to 3h23m.

The first option does a lot of work on data insertion (checking if data would not fit in allocated memory and so on), so no wonder it was the slowest.

The second option has another drawback – one needs to monitor report sizes and change the compile-time variable of buffer pre-resized size. The third option does not need another memory buffer (`fmt::print` uses an internal one for each function call).

N. ClickHouse coverage report

ClickHouse Coverage Report (referred to as *CCR*) is a simple format for accumulating large coverage reports while preserving per-test coverage data. It can be easily converted to `.info` for `genhtml`.

```
/absolute/path/to/ch/src/directory
FILES <source files count>
<source 1 relative path> <functions> <lines>
<func 1 name> <start line> <edge index>
<func 2 name> <start line> <edge index>
<instrumented line 1>
<instrumented line 2>
<source 2 relative path> <functions> <lines>
```

Listing 12: CCR header (function names are mangled)

```
TEST
SOURCE <source id> <funcs count> <lines count>
<function 1 edge index> <call count>
<function 2 edge index> <call count>
<line 1 number> <call count>
<line 2 number> <call count>
```

Listing 13: CCR entry

```
TESTS
<test 1 name>
<test 2 name>
```

Listing 14: CCR footer

CCR consists of 3 sections: header (listing 12), entry (listing 13), and footer (listing 14).

The motivation for using CCR instead of `genhtml`'s `.info` was the latter's redundancy. `.info` was designed to contain information that could be discarded later [17].

Consider `.info`'s function entries:

```
FN:<function start line>,<function name>
FNDA:<call-count>,<function name>
```

[17]For example, function coverage that would be ignored when invoking `genhtml --no-function-coverage`

Function's mangled name gets duplicated twice, for general coverage and for function coverage. It is inappropriate for a project of ClickHouse's size.

`.info` stores absolute paths for every source file. This thesis's target was to instrument only source files in `src/` folder, so one could store relative paths from that directory.

Also, one would need to duplicate all information about all functions and lines for every test (including test-agnostic information like all instrumented functions in binary).

As a result, the `.gz` per-test data in `.info` (compressed with `zlib`) took ≈ 2.8GB on disk (≈ 1.1MB per a report). After introducing CCR we managed to lower this size to ≈ 180MB for an unzipped version.

O. Thread-safety and synchronization

We have two situations where a data race may occur:

- Calling `hit` from multiple threads.
- Copying data to process in other thread (calls to `hit` may occur).

1) Possible race in hit calls: `hit` may be called from multiple threads (as ClickHouse is a multithreaded application).

We have different options of synchronization:

- Using an exclusive lock for every hit (mutex), turned out to be too slow. For example, one of functional tests `00002_system_numbers` called `hit` about 10 000 000 times (and that is not the upper limit on calls). So a binary with such implementation would need to perform about 20M operations (acquire and release) on a mutex (and these operations are not free).
- Writing to static thread-local storage in batches, then copying them to global storage under an exclusive lock. Faster, but one cannot get the last batch of data as then one would need to invoke `hit` from all threads that previously invoked `hit`. The solution implementing this mechanism via thread identifiers was overengineered.
- Using atomics from `std::` for individual cache cells. This option frees us from using a global lock. The drawback here is the inability to resize a vector of atomics (or copy them at once, or swap them with another empty vector) as they are neither copy-constructible nor movable. As atomics are not copy-constructible, one cannot use a vector of atomics directly. A `vector<unique_ptr<atomic>>` may be used instead, but this introduces additional heap allocations for every edge for every test which is a lot. Please note the section where data are copied is somewhat critical as it is called from a `Settings::setSetting` function thus needing to be as fast as possible. Sadly, the vector of unique pointers of atomics showed about the same performance as a global lock, so it was not used.
- Using atomic-like objects. `clang` has a default implementation of coverage sanitizer callbacks. This implementation uses internal "atomics" which are default-constructible and copy-constructible. Such atomics consist of a pair of a needed type and a special volatile variable which is used to guarantee atomicity via inline assembler. However, this interface is not publicly exposed.
- Using compiler-provided macros for atomic operations on non-atomic types.

  ```
  __atomic_add_fetch(
      &edges_hit[edge_index], 1,
      __ATOMIC_RELAXED);
  ```

  This is the fastest, though perfectly valid solution, which was used.

2) Possible race in data copying: The function that copies hits data to process it in other threads is called under an exclusive lock (located up the stack in `Settings.cpp` when attempting to set a setting), so only one thread may copy test data, no external synchronization is needed.

The only issue here may be a `hit` call from some thread while the data are being copied, but it also is safe.

- hit gets optimized to something like

  ```
  addq $1, (%rax,%rbx,8)
  ```

  where `rbx` is the edge index passed, `rax` is a pointer to vector's data, namely, `vector.begin()`.
- Data copying (or, to be precise, data swap), excluding allocation checks which do not apply here, is two swaps (vectors' member pointers and vector sizes). `hit` does not care about vector size as `operator[]` only has a checking assertion in debug so it is just swapping member pointers. Pointer swap can be done using at most three instructions [18]. At any time point, the member pointer stays valid (and can be safely retrieved by `hit`), so the program's behaviour is defined.

We have two cases:

- `hit` gets called from the thread when the `v2` pointer was set to a new value. Hit gets counted as planned.
- `hit` gets called before the last swap operation. In that case, hit would count for the next test. As already said, we do not care about precise results, so that also is fine.

P. What should be put in the blocklist

```
#include <iostream>

static auto f = [] {
    std::cout << "Hello";
    return 1;
}();

int main() {}
```

Listing 15: Printing in static lambda

1) Global static variables initializers: Imagine a program (listing 15) having a static object that needs to log data to the standard output stream.

[18] Load `v1` to a temporary register, set it to `v2`, set `v2` from the temporary register

C++ has an <iostream> header providing global object std::cout. However, according to standard, you cannot use std::cout in your static objects as static objects do not have a specified initialization order in different translation units. This is known as static initialization order fiasco.

Different compilers solve this issue differently. clang's libcxx [19], processing each translation unit that includes iostream, constructs a special static object in the first instance.

One can lookup a symbol

```
_GLOBAL__sub_I_translation_unit_name.cpp
```

responsible for initializing global static variables in a translation unit. It invokes constructors, represented by symbols

```
// example: __cxx_global_var_init.8
__cxx_global_var_init.variable_index
```

for every static variable in a unit. Symbol

```
__cxx_global_var_init
```

is reserved for an iostream object initializer.

We do not need to instrument the iostream header, and instrumenting static variables initializer slows down both the symbolization (due to debug information lookup strategy) and tests (as contribs have lots of static variables).

```
set_source_files_properties(
  file.cpp
  PROPERTIES COMPILE_FLAGS
  "-O3 -finline-functions \
    ${WITHOUT_COVERAGE}")
```

Listing 16: Removing a .cpp file from instrumentation

```
#pragma clang flag push
  "-fno-sanitize-coverage=trace-pc-guard"
void foo();
#pragma clang flag pop
```

Listing 17: Desired clang flag application

```
set(COVERAGE_BLOCKLIST
  "blocklist.txt")
file(WRITE ${COVERAGE_BLOCKLIST}
  "fun:__cxx*\nfun:_GLOBAL_*\n")

function(build_files_without_coverage)
    foreach(arg IN LISTS ARGN)
        file(APPEND
          ${COVERAGE_BLOCKLIST}
          "src:${arg}\n")
    endforeach()
endfunction()

build_files_without_coverage(foo.h bar.h)
```

Listing 18: Dynamic blocklist generation

[19]Used as a standard library in ClickHouse regardless of the compiler

2) Submodules' namespaces: For non-header source files (e.g. .cpp, .hpp) one can apply a COMPILE_PROPERTY with a -fno- instrument flag. It can be done via cmake (listing 16).

However, ClickHouse also uses multiple header-only libraries like boost::header_only. Compile properties cannot be applied to header files as the latter are not passed to the compiler driver directly [20].

3) Alternative solutions tried:

- Using allowlist.txt.
  With a format similar to the blocklist, allowlist could have been used to limit the instrumented folders to src/ only, but it does not solve the issue with header files, as a file in src/ may include a header from a contrib, and allowlist would not catch that.
- Using a preprocessor macro.
  clang does not have options to insert a non-diagnostic flag onto a file or a function, so this imaginary snippet (listing 17) is invalid.
  A possible solution would be to append a .cpp postfix for every header file, apply compile options to it and create a header file with same name containing #pragma once; #include file.h.cpp.
  However,
  - Functions in multiple translation units would compile with or without instrumentation, and the result would entirely depend on the linker's weak symbols folding algorithm.
  - During compilation of such .cpp files no templates (needed in other files) would be instantiated.
- Generating the allowlist and blocklist dynamically.
  One could list all files that should not be instrumented in cmake and put them into the blocklist (listing 18). Unfortunately, this still does not solve the issue with headers.

VII. Applications of CCR usage

A. Tests reorderer

Testing scripts in CI run tests from newest to oldest (that means a test with 00001 prefix would be run last). Having the CCR file, we can use the contained information to find tests that are the most relevant for changes in a commit. Approximate algorithm:

- Obtain all changed files and all changed lines (for example, Python3 provides a unidiff module that can get and parse the diff for a given commit hash).
- Iterate over all changed [21] and removed files [22].
- For each *hunk* (a changed consecutive line set) in a file get old line numbers. For every line get a list of tests that cover this line (for every test compute a set difference

[20]One can check it in the compile_commands.json file

[21]Renamed files are thought to be changed

[22]We do not have coverage information for added files as coverage report is taken from the previous commit

between instrumented line ranges and hit line ranges and check if line is inside it).
- Create a global tests list via concatenating lists for individual lines for every hunk. Count unique items occurrences (for example, using Python's `collections.Counter`) in that list.
- Sort the tests using the occurrences as weights (the higher the occurrence is, the earlier the test will appear). If the test was not present in the previously mentioned list, it would not be reordered.

B. Incremental coverage

One can write a script that, provided with two CCR files, computes their difference. The simplest implementation calculates ratios ("hit lines count" / "instrumented lines count") for every file in each report and compares them (if files are present in both reports). This information may be used to obtain a global coverage ratio.

VIII. Results

- The ClickHouse's CI pipeline builds a coverage report for every commit. Each coverage run takes ≈ 3 hours (≈ 3 times speedup in comparison with the previous solution).
- Per-test data (CCR report) as well as the HTML visualisation are available on the corresponding GitHub check page.
- All the sources are located [7] in the ClickHouse GitHub repository, allowing other companies like Altinity to use this code in their ClickHouse testing pipelines.

IX. Conclusion

In this paper we described a fast code coverage runtime that preserves per-test values. We managed to gain a substantial testing speedup. Moreover, we also researched the instrumentation algorithms that compilers use to achieve the best runtime performance. Finally, we integrated the runtime into ClickHouse's testing pipeline.

X. References

[1] Basic block, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Basic_block. [Accessed: 15-May-2021].

[2] GIMPLE (GNU Compiler Collection (GCC) Internals). [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html. [Accessed: 15-May-2021].

[3] LLVM Language Reference Manual. [Online]. Available: https://llvm.org/docs/LangRef.html#id1308. [Accessed: 15-May-2021].

[4] GCC, the GNU Compiler Collection. [Online]. Available: https://gcc.gnu.org/. [Accessed: 15-May-2021].

[5] Clang C Language Family Frontend for LLVM. [Online]. Available: https://clang.llvm.org/. [Accessed: 15-May-2021].

[6] ClickHouse DBMS. [Online]. Available: https://clickhouse.tech/. [Accessed: 15-May-2021].

[7] Code coverage runtime by myrrc. [Online]. Available: https://github.com/ClickHouse/ClickHouse/pull/20539. [Accessed: 15-May-2021].