The Case for Multi-Version Experimental Evaluation (MVEE)

Simon Jörz
Johannes Gutenberg University
Mainz, Germany
joerzsim@uni-mainz.de

Felix Schuhknecht
Johannes Gutenberg University
Mainz, Germany
schuhknecht@uni-mainz.de

ABSTRACT

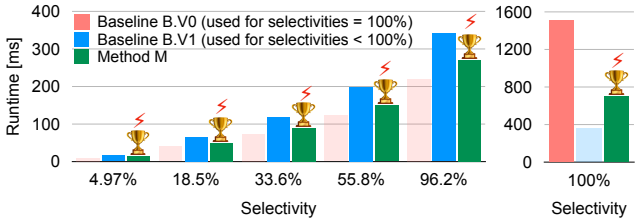
In the database community, we typically evaluate new methods based on experimental results, which we produce by integrating the proposed method along with a set of baselines in a single benchmarking codebase and measuring the individual runtimes. If we are unhappy with the performance of our method, we gradually improve it while repeatedly comparing to the baselines, until we outperform them. While this seems like a reasonable approach, it makes one delicate assumption: We assume that across the optimization workflow, there exists only a *single* compiled version of each baseline to compare to. However, we learned the hard way that in practice, even though the *source code* remains untouched, general purpose compilers might still generate highly different *compiled code* across builds, caused by seemingly unrelated changes in other parts of the codebase, leading to flawed comparisons and evaluations. To tackle this problem, we propose the concept of *Multi-Version Experimental Evaluation (MVEE)*. MVEE automatically and transparently analyzes subsequent builds on the assembly code level for occurring "build anomalies" and materializes them as new versions of the methods. As a consequence, all observed versions of the respective methods can be included in the experimental evaluation, highly increasing its quality and overall expressiveness.

PVLDB Reference Format:
Simon Jörz and Felix Schuhknecht. The Case for Multi-Version Experimental Evaluation (MVEE). PVLDB, 19(12): 4778 - 4781, 2026.
doi:10.14778/3827998.3828120

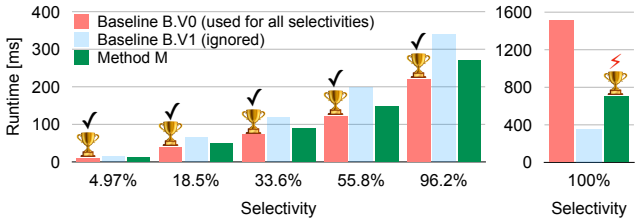
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.rlp.net/mvee.

1 INTRODUCTION AND MOTIVATION

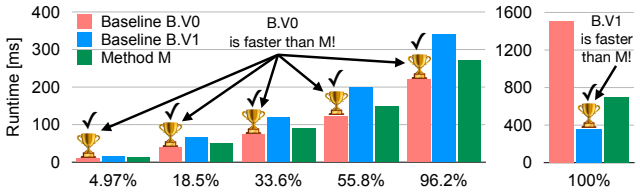
Traditionally, database research heavily relies on experiments. A typical workflow for an evaluation of a new method looks as follows: We first integrate the method along with a baseline into a single codebase. Then, we run both on the same workload and compare the individual runtimes. If our method is not fast enough, we incrementally try to optimize while re-running the benchmark until we reach our goal. While at first glance, this looks like a reasonable workflow, it unfortunately relies on the assumption that the incremental builds of the codebase we compare with each other are actually *comparable*. We assume that all modifications we apply to our method remain local in the actual build and do not affect any other parts of the codebase, such as the generated code of the baseline method.



(a) Problem 1: Merging results from different builds → false interpretation.



(b) Problem 2: Showing only the results of one build and "forget" about another potentially better build → false interpretation.



(c) Solution MVEE: Actively including the results of *all* seen versions to get the most complete picture → meaningful interpretation.

Figure 1: Evaluation workflow with and without MVEE.

Unfortunately, we learned in previous work [8] that this assumption cannot be safely made when using a general-purpose compiler such as `gcc`: After experiencing unexplainable runtime variances across builds in one of our research projects [7] that evaluated scan-accelerating index structures across different selectivities, a deep investigation revealed irregularities across these builds on the assembly level: The generated code of a baseline method `B` was heavily affected by certain changes in completely unrelated code parts within the same compilation unit, presumably triggered by one of the many optimization heuristics [3–6] used within the compiler. This effectively resulted in the unpredictable occurrence of two very different versions `B.V0` and `B.V1` of the same baseline.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828120

If these versions go unnoticed one could easily miss comparing to the "right" version. Depending on the workflow, we have to differentiate between the following two cases: In the first case, results from *different* builds are gathered. Figure 1a showcases this with real data from [8], where the baseline result from one build is used in the plot for a selectivity of 100%, whereas the results from another build are used for the remaining selectivities. As the example shows, if we are unaware of the existence of the two baseline versions `B.V0` and `B.V1`, we will conclude that `M` generally outperforms the baseline. In reality, however, we just compared against the versions that perform worse in the respective situations. In the second case, all results originate from a *single* build, but from an arbitrary one, as showcased in Figure 1b. While this at least ensures consistency, we might still draw wrong conclusions as we can miss better versions: In our example, this is the case for a selectivity of 100%, where we compare `M` against the worse version `B.V0`, while `B.V1` would perform significantly better.

1.1 Multi-Version Experimental Evaluation

Unfortunately, as long as we deal with complex black-box compilers, we have to assume that such anomalies occur in practice, even if we try to counter-steer the generation of different versions via a stabilization tool [1]. However, instead of simply ignoring the generated versions, we argue to actively incorporate them into the experimental evaluation workflow: If the generated assembly code of a baseline method suddenly changes from one build to the other due to a compilation anomaly without a corresponding change in the source code, we do not ignore this, but treat the anomaly as a new compiled version of the baseline method. At the same time, we do not forget about the old compiled version, but also keep it, along with the experimental results produced therefrom. As a consequence, as shown in Figure 1c, we can include the results from *all* seen compiled versions to get a complete picture of whether our method *actually* beats the baseline or not.

In the following, we materialize this concept as what we call *Multi-Version Experimental Evaluation (MVEE)*. We integrated MVEE as a comfortable extension for the VSCode IDE, which we will showcase in this demonstration. During the development workflow, MVEE automatically and transparently analyzes the relevant compiled code of generated builds on the `x86` assembly level for occurring anomalies. As soon as an anomaly is detected, MVEE does not only report this to the developer for closer inspection by showing the corresponding assembly and source code portions, but automatically registers the change as a newly seen compiled version of the corresponding source code section, effectively building up a version graph that is shown to the user. Additionally, it stores the experimental results of the corresponding run and maps them to this particular version, which allows to include these results in the produced plots.

Before we jump into the description of the workflow, let us clearly define what we consider as equivalent builds and what as anomalies. We consider two sequences of assembly code lines S_1 and S_2 as equivalent if the following three conditions are all met: (1) Every instruction in S_1 also exists in S_2 and vice versa. (2) Every instruction in S_1 operates on the same data as the corresponding instruction in S_2. (3) The control flow of S_1 is the same as of S_2. Consequently, any pair of assembly code line sequences that violates our equivalency definition is considered as an anomaly. Note that within these checks, we currently make the following relaxations for efficiency: First, we ignore indirect jumps based on register content. This allows us to check for these conditions statically by analyzing the code without requiring any run-time data. Second, we ignore the register assignment, as anomalies would be detected on the instruction and control flow level already.

2 MVEE WORKFLOW

To demonstrate how MVEE operates, we consider an exemplary development workflow, where we want to compare the methods `M`, `B0`, and `B1`. Method `M` represents our own method that we optimize incrementally, whereas `B0` and `B1` represent baseline methods.

To bootstrap the system, the user first communicates to the MVEE extension which code sections are actually relevant for the experimental evaluation and should be monitored. Further, the user has to introduce an identifier for each code section, such that MVEE can later map both active modifications as well as occurred anomalies in the code to the corresponding sections. This is done by surrounding the code of interest by calls to our mark pre-processor definitions `gen_begin_mark()` and `gen_end_mark()`.

Now, let us go through the development workflow for a couple of steps and see how MVEE handles the occurring effects. Based on the modification and anomaly detection, MVEE builds up a *version graph* as shown in Figure 2. This version graph keeps track of all occurred compiled versions for the individual methods. More importantly it allows to identify at any point in time which versions must currently be considered for a complete result interpretation.

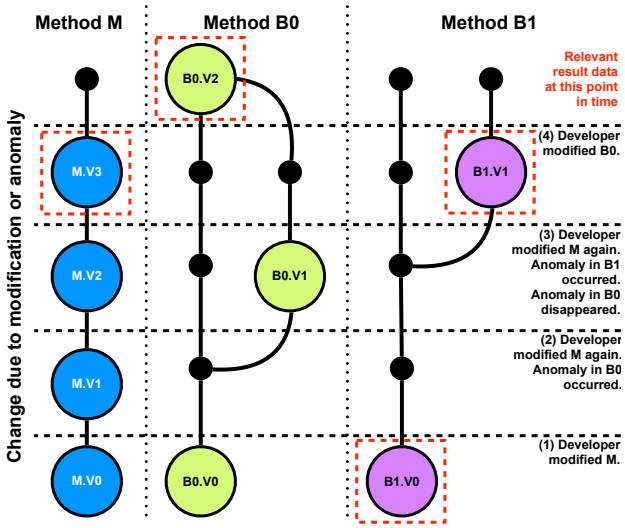


Figure 2: The version graph built up by MVEE. Each colored circle represents a newly seen compiled version of the method. A source code modification continues respectively merges paths and an anomaly forks the path.

We start with an initial build of the codebase, which creates the initial versions `M.V0`, `B0.V0`, and `B1.V0`. In step (1), the user optimizes method `M` and rebuild the codebase, which creates the version `M.V1`. Since the other methods were not modified and also did not undergo an anomaly, their initial versions and results remain valid, which is indicated by the small black circles in the graph. In step (2),

the users optimizes `M` again and recompiles, resulting in the compilation of a new version `M.V2`. However, the generated code of `B0` changed over the previous build, which MVEE detects and classifies as an anomaly, since the corresponding source code has not been modified. In the version graph, this anomaly is reflected by a fork to version `B0.V1`. Due to the fork, both versions `B0.V0` and `B0.V1` would be considered for result interpretation. In step (3), let us now assume that `M` is modified once more, which now causes two side-effects: First, it triggers an anomaly in `B1`, resulting in a fork between `B1.V0` and `B1.V1`. Second, the previously occurred anomaly in `B0` disappears and `B0` compiles again to the previously seen version `B0.V0`. Both versions of `B0` still need to be considered for result interpretation. This changes in step (4), when the users actively modifies the baseline `B0`, e.g., to fix a bug. This modification obviously creates a new version `B0.V2`, which merges the fork, indicating that the previous versions `B0.V0` and `B0.V1` are outdated and should no longer be considered. In summary, at any point in time, MVEE locates for each method all relevant versions by traversing from top to bottom along all paths until the first version on each path is visited. These versions in conjunction are considered during result interpretation. In Figure 2, we mark all versions that are relevant after the last step in red boxes.

3 ARCHITECTURE AND IMPLEMENTATION

Figure 3 provides a high-level view on the components of our MVEE implementation and how they interact with each other. MVEE itself is split into two components: The VSCode extension and MVEE core, which handles the assembly file analysis.

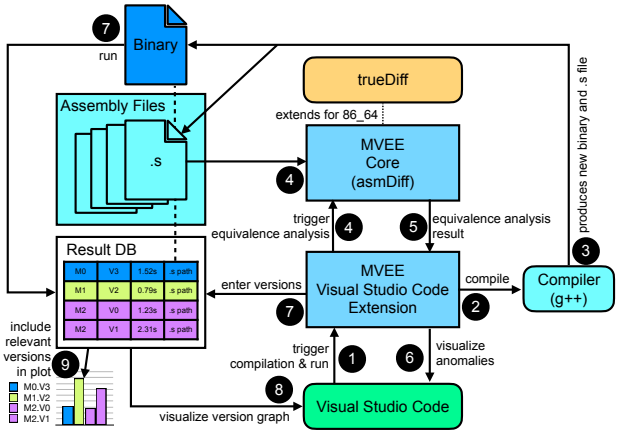


Figure 3: High-level architecture and interaction.

MVEE operates as follows: In ❶, from the IDE, the user triggers the MVEE extension to compile the current codebase in order to perform a new experimental run. In ❷, our extension then issues the compilation, which in ❸ produces a new binary and the corresponding `.s` file. This assembly file will be stored with the previous builds in the assembly files directory, named by the timestamp of the run. Next, in ❹, our extension instructs MVEE core to perform an equivalence analysis for all methods that were not modified in the source code since the last build. This happens based on the corresponding `.s` files. The core component passes the result of this equivalence analysis for each compared method in ❺ to our extension — if anomalies have been detected, they can be inspected by the user in ❻ directly in the IDE. In ❼, the extension now triggers the actual run of the binary, which produces a new measurement for each method. If an anomaly has been detected before in a method, the new measurement of that method enters the result database as a new version. Additionally, in ❽, the version graph that is displayed in the IDE is updated accordingly. Finally, in ❾, all relevant versions of the result database are extracted from the result database to generate a corresponding results plot.

3.1 Assembly Code Analysis

The actual assembly code analysis operates as follows: Given a pair of assembly files to compare, MVEE core first parses each file and computes a corresponding tree-like intermediate representation. The parser currently support x86-64 assembly in AT&T syntax. However, it is easily extendable to other assembly styles. Then, for the comparison, MVEE core leverages *truediff* [2], a tree based structural diffing algorithm which is designed to return concise and type safe edit scripts for typed tree-shaped data. To be usable in our context, we extended *truediff* to handle `x86` assembly code. As a result of the diffing, *truediff* produces a so-called *edit script*, which captures all the observed differences between the input assembly files. MVEE core then analyzes the edit script and identifies differences that violate our definition of equivalence (see Section 1.1). In the following, we outline the steps in detail:

Assembly files → *truediff* input. MVEE core first parses all assembly instructions of a file and generates a corresponding intermediate representation. From the intermediate representation, MVEE core next extracts only the code portions that are relevant for the experimental evaluation, which relies on the marks placed in the C++ code. Precisely, MVEE core indirectly builds and traverses the control flow graph from each start mark until the corresponding end mark. To keep this control flow intact, we group instructions into fallthrough-groups. Each fallthrough-group is a sequence of instructions where one can be executed after another without an unconditional jump or call occurring in-between. Such fallthrough-groups can be reordered without changing the flow of control.

Fallthrough-groups → edit script. Next, MVEE core passes the fallthrough-groups of both assembly files as source tree and target tree to the *truediff* algorithm to compute the edit script. This edit script describes how to modify the source to obtain the target. On a high level, *truediff* encodes the structure and literals of all contained subtrees as hashes and identifies similarities by comparing these hashes. Subtrees of the source that match with subtrees of the target are reused, while non-matching subtrees are deleted or inserted as needed. In this way the *truediff* algorithm creates a concise edit script, while ensuring to use every node exactly once, within linear run-time complexity.

Edit script → anomalies. Then, MVEE core analyzes whether the edit script violates our definition of equivalence in three steps: (1) MVEE core checks whether the script removes or inserts any new instructions, operands or labels. The existence of such significant structural edits clearly violates equivalency. (2) MVEE core checks whether the updates do not change any data and are consistent. Precisely, it gathers updates to operands (memory references or immediate values). If updates change an immediate or a memory reference this is considered as a violation of the equivalency. If an update changes a label, MVEE core verifies that all other references to that label are changed to the same new label as well. Is that not

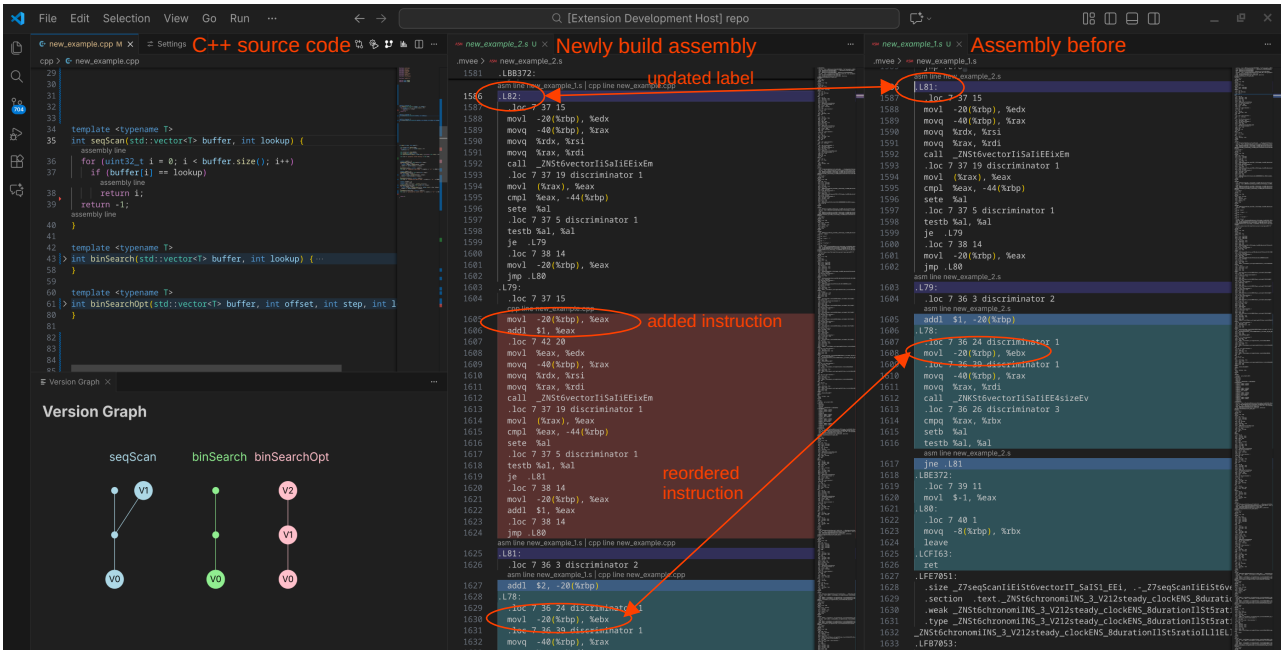


Figure 4: The inspection of a detected anomaly in our MVEE VSCode extension.

LoC	Parsed	Analyzed	Edits	Parse (ms)	Extr. (ms)	Eq. (ms)	Total (ms)
~100	33977	3325	10	498.0	28.8	87.6	614.4
~1000	57509	13055	8	806.0	48.8	165.4	1020.2
~10000	290377	110339	10	3346.0	369.4	834.2	4549.6

Table 1: Runtime breakdown of MVEE core while scaling synthetically generated code.

the case, then the code is non-equivalent. (3) MVEE core checks whether the reordering of code changes the control flow. Reordering a fallthrough-group does not change the control flow, as jumps remain consistent. But reordering single instructions (or operands) does change the control flow. In the latter case MVEE core consideres a reordering as a violation of the equivalency. Finally, it returns the equivalence result and all edits.

To analyze the overhead of MVEE core, we break down its runtime in Table 1. We perform a benchmark where we generate synthetic C++ code, copy it, change one line in the copy, and compare both using MVEE core. While the lines-of-code (LoC) are scaled with a factor of 10× the total runtime of MVEE core decreases only by a factor of 2–5×. Therefore, we argue that the overhead which MVEE core adds is acceptable for the benefit it provides.

4 CONFERENCE EXHIBIT

Figure 4 shows how the equivalence result is presented to the user in our VSCode extension. Using `CodeLens`, our extension links the detected anomaly to the corresponding C++ source code lines, such that the user sees from where the anomaly originates. By clicking on the link, the corresponding assembly file opens, in which all edits are highlighted in different colors depending on their category. Additionally, the MVEE extension continuously visualizes the version graph in a git-style manner.

At the venue, the audience will be able to experience the behavior of the extension and the benefits of MVEE at the example of multiple observed real-world anomalies which have been produced during compilation with the general-purpose compiler gcc. These real-world anomalies include the discussed case from our motivating example [8], but also further observed anomalies — identified by colleagues in their own benchmarking code bases but also found via automated extensive search using AI tools. To ensure that the audience actually experiences anomalies on site, we prepare for all code bases in advance a set of snapshots that capture the state between critical source code changes that lead to the generation of compilation and performance anomalies. The audience can then (a) observe how our MVEE extension automatically detects the anomalies in the respective code parts, and how they are correctly registered as new versions in the version graph. Additionally, (b) the audience is able to analyze the type of anomalies both on the C++ and assembly level in detail, as shown in Figure 4. To experience the impact on an experimental evaluation, the audience is further able to (c) produce runtime plots that correctly include all identified versions that are relevant for the experimental evaluation.

REFERENCES

[1] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: statistically sound performance evaluation. In *ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 219–228.

[2] Sebastian Erdweg, Tamás Szabó, and André Pacak. 2021. Concise, type-safe, and efficient structural diffing. In *PLDI '21: 42nd ACM SIGPLAN, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 406–419.

[3] GNU. 2025. https://gcc.gnu.org/onlinedocs/gccint/. (2025).

[4] Jan Hubicka. 2005. Profile driven optimisations in GCC. *Proceedings of the 2005 GCC Developers' Summit* (2005).

[5] Jan Hubicka. 2012. Advanced Interprocedural Optimization in GCC. *Proceedings of the 2012 GCC Developers' Summit* (2012).

[6] Raja Mehrotra et al. 2013. Improving GCC's loop optimizer. *Proceedings of the 2013 GCC Developers' Summit* (2013).

[7] Felix Schuhknecht and Justus Henneberg. 2023. Accelerating Main-Memory Table Scans with Partial Virtual Views. In *DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*, Norman May and Nesime Tatbul (Eds.). ACM, 89–93. https://doi.org/10.1145/3592980.3595315

[8] Felix Schuhknecht and Justus Henneberg. 2023. Why Your Experimental Results Might Be Wrong. In *DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*, Norman May and Nesime Tatbul (Eds.). ACM, 94–97.