WAX: Optimizing Data Center Applications With Stale Profile

Tawhid Bhuiyan
Columbia University
New York, NY, USA
mb5332@columbia.edu

Sumya Hoque
Columbia University
New York, NY, USA
sh4242@columbia.edu

Angélica Aparecida Moreira
Microsoft Research
Redmond, WA, USA
anmoreira@microsoft.com

Tanvir Ahmed Khan
Columbia University
New York, NY, USA
tk3070@columbia.edu

Abstract

Data center applications' large instruction footprints cause frequent front-end stalls by overwhelming on-chip microarchitectural structures such as instruction cache (I-cache), instruction translation look-aside buffer (iTLB), and branch target buffer (BTB). To reduce pressure on these structures, data center providers leverage profile-guided optimizations by reordering binary layout along a relatively small number of hot code paths. Such reordering provides the highest benefit if profile collection and optimization happen on the same version of the binary. In practice, companies have to optimize and deploy a fresh version of the binary with a profile from a previous version, making a large fraction of the profile stale. In this paper, we propose WAX[1], a novel technique to optimize data center applications with stale profiles. WAX's key insight is to leverage the debug and source code information while optimizing fresh binaries with stale profiles. We evaluate WAX for 5 data center applications to show that WAX provides significant (5.76%-26.46%) performance speedups. WAX achieves 1.20%-7.86% greater speedups than the state of the art, obtaining 65%-93% of fresh profiles' benefits.

CCS Concepts: • **Computer systems organization → Pipeline computing**; • **Software and its engineering → Compilers**; **Software performance**.

Keywords: Profile-guided optimizations; Profile staleness; Frontend stalls; Debug information; Data centers

[1]We open source our work at https://github.com/ice-rlab/wax

ACM Reference Format:
Tawhid Bhuiyan, Sumya Hoque, Angélica Aparecida Moreira, and Tanvir Ahmed Khan. 2026. WAX: Optimizing Data Center Applications With Stale Profile. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790248

1 Introduction

Data centers consume a substantial amount of energy, causing significant operating cost [20, 32]. Data center operators such as Google and Meta have to enable efficiency by reducing operating cost, while also maintaining the performance customers need [7, 8, 20, 32, 55, 58, 68]. One key obstacle to achieving energy and performance efficiency is the large code footprint of data center applications [8, 32, 55, 57, 58]. As these applications have complex logic [55] with frequent invocations of libraries [32], language runtime [5, 54, 55], and kernel modules [8], their large code footprints exhaust microarchitectural structures in the processor frontend, such as instruction cache (I-cache) [36, 38], instruction translation lookaside buffer (iTLB) [12, 71], and branch target buffer (BTB) [34, 66, 67]. As a consequence of these *frontend stalls*, data center processors fail to perform useful operations, causing significant performance and energy inefficiencies worth millions of dollars [8, 28, 32, 53, 68, 80].

Data center operators aim to address frontend stalls through Profile-Guided Optimizations (PGO) by changing code layout to place frequently executed (*i.e.*, "*hot*") code together. Specifically, companies like Google and Meta continuously profile their data center applications in production via efficient hardware support [2] and leverage these profile samples to guide optimizations using systems like AutoFDO [13], BOLT [57], and Propeller [65]. Unfortunately, application binaries corresponding to such profiles are always lagging behind the most recent sources [13], as we show in Fig. 1. Profiling the fresh binary before deployment is also not possible, as collecting representative profiles requires live traffic [32, 35, 55, 60].

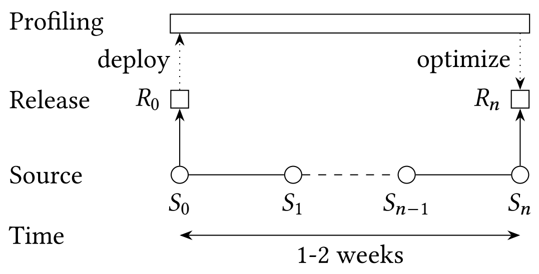


Figure 1. Google [13] and Meta [9] continuously profile data center applications' one version (R_0) and use these profiles to optimize a subsequent version (R_n). As practitioners deploy new versions every one or two weeks, source codes change rapidly between these versions, causing 70-92% of profile samples to become stale; adapted from [9].

Consequently, practitioners have to optimize the fresh binary with the profile of the last deployed binary [9, 13]. If sources and profiles for data center applications change slowly, *i.e.*, over several weeks, PGO techniques could still adapt to changing profiles [13] as companies like Google and Meta release new binaries weekly or bi-weekly [9]. Alas, recent work [9, 27, 28] reports that data center applications' profiles change rapidly and even within a week, 70% of profile samples do not match the fresh binary, rendering them *stale*, "invalid for optimizations [9]." As a result, PGO systems fail to avoid two thirds of all frontend stalls [9], losing significant performance potential [27].

Prior works [9, 48, 74] aim to address profile staleness by using binary code similarity [6, 26]. To map stale profile to the fresh binary, these techniques match the similarity of assembly instructions, basic-blocks, and functions between stale and fresh binaries. These techniques measure the similarity of binary functions based on their "mangled names" [31], while hashing assembly instructions and jump targets to match basic-blocks. Unfortunately, as we (§2.4) and others [9, 13] observe, matching the similarity of assembly instructions and basic-blocks is especially challenging for C++ binaries as function inlining vary widely across versions of the same application.

In this paper, we first quantify the performance implications of these challenges for C++ binaries, and then characterize how functions and basic-blocks change across various versions, while also identifying potential solutions. Optimizing 5 real-world data center applications with stale profile, state of the art [9] obtains 3.9%-18.60% speedups, which are only fractions of fresh profiles' speedups (7.64%-38.45%). For function mapping, we observe that even small changes in mangled names make it challenging for state-of-the-art techniques to address profile staleness. The mangled names of C++ functions depend on their various components, such as names, namespaces, and parameters. Furthermore, Link-Time Optimizations (LTO) such as function partitioning [11] and constant propagation [39], also affect functions' mangled names. Analyzing the changes in the source code of data center applications, we observe that many hot functions exhibit changes in their names, namespaces, parameters, or LTO partitions, modifying their mangled names. As a result, state-of-the-art techniques [9, 74] fail to map 2%-33.9% of function samples from stale profile to fresh binary.

Similarly, existing techniques [9, 48, 74] face significant challenges while mapping basic-blocks due to changes in function inlining and source code. Existing techniques map basic-blocks using hash values. These techniques compute the hash value of a basic-block based on its predecessor, successor, and instruction content, such as opcodes and operands of its assembly instructions. For a basic-block, changes in function inlining mainly affect its predecessors and successors, while changes in source code primarily modify its instruction content. For example, minor changes in source code leading to addition or removal of even a single instruction, or opcode change of a single instruction, alter a basic-block's hash value across versions. Inspecting the changes in the source code of data center applications, we observe that many hot basic-blocks experience such small changes. As a result, the state of the art fails to map 9.5%-39.3% of basic-block samples from stale profile.

The key takeaway of our characterization is that information available at the binary level is insufficient to address profile staleness. Our characterization also reveals that the source code of data center applications includes useful information to address this insufficiency. We could leverage this useful information of the source code using the debug information of the data center applications' binary.

Based on our characterization's insights, we propose Wax, a novel technique that addresses profile staleness by leveraging source code and debug information. In particular, Wax maps the source code from the stale version to the fresh version using *Source Mapping*. Wax combines these source mappings with debug information to perform *Function Mapping* and *Basic-block Mapping*.

For *Source Mapping*, Wax first identifies all source locations corresponding to an application's binary using the debug information. A source location consists of a source file and a line number. Consequently, Wax creates 1-to-1 mappings of stale and fresh source locations by comparing their file names, absolute paths, and source code content.

For *Function Mapping*, Wax first identifies all the binary functions corresponding to each source file using the debug information. Next, to map a function from a stale file, Wax compares it against all functions from the corresponding fresh file. While comparing functions, Wax leverages demangling [31] to compare their namespace, name, parameters, and LTO partitions separately. Comparing these demangled information separately and limiting the comparison only to the mapped pair of source files enables Wax to accurately map more functions, despite changes across versions.

For *Basic-block Mapping*, WAX first identifies all the basic-blocks corresponding to each function using the debug information. Next, to map a basic-block from a stale function, WAX compares it against basic-blocks from the corresponding fresh function using their assembly instructions, predecessors, and successors. While comparing basic-blocks and instructions, unlike prior work [9, 48, 74], WAX only considers basic-blocks and instructions corresponding to a mapped pair of stale and fresh source lines. Limiting comparison only to the mapped pair of source lines helps WAX avoid inaccuracies due to changes in function inlining and source code, while also reducing the overhead of comparing all basic-blocks and instructions with each other.

We evaluate WAX on 5 data center applications—gcc, clang, mysql, postgres, and mongodb—using a range of major and minor versions. Our evaluation results show that WAX guides optimizations with stale profiles across all applications, providing significant performance gains (5.76%-26.46%). In particular, WAX provides 1.20%-7.86% greater speedups compared to the state-of-the-art technique [9]. Even with stale profiles, WAX achieves 65%-93% of fresh profiles' benefits. We also evaluate WAX's efficiency in terms of compilation time and memory overhead. Furthermore, we demonstrate that WAX retains its performance gains across various workloads/inputs, application versions, and optimizations. In terms of accuracy, WAX achieves better F_1-*scores* than the state-of-the-art technique. Using stale profile, WAX outperforms the state-of-the-art online code layout optimizations system, OCOLOS [77, 78], even when OCOLOS finishes optimizations with fresh profile. Finally, we apply WAX's approach to optimize clang with Propeller [65] to show that Propeller speeds up clang by 6.4% with WAX's approach compared to 5.4% gains with the state of the art.

This paper makes the following contributions:

- A comprehensive characterization of profile staleness in data center applications, identifying key challenges of mapping stale functions and basic-blocks to fresh binaries.
- WAX: a novel technique to map stale functions and basic-blocks to fresh binary using source and debug information.
- An extensive evaluation of WAX for 5 data center applications to show that WAX achieves 5.76%-26.46% performance speedups, providing 1.20%-7.86% greater speedups than the state of the art.

2 Understanding the Challenges of Optimization with Stale Profile

In this section, we investigate the behavior of 5 data center applications to understand the challenge of using stale profiles for optimizations. In particular, we examine the gap between the speedup of binaries optimized with the fresh profile and the speedup of binaries optimized with the stale profile with the state-of-the-art approach. Providing a brief overview of the state-of-the-art approach, we show how data center applications' behavior stops the state of the art from addressing profile staleness.

Table 1. Data center applications, their fresh versions, stale versions, and workloads we study, similar to prior work [9].

Application	Fresh	Stale	Workloads
gcc	9	8	Compiling a large template-heavy
clang	15	14	LLVM source file [40]
mysql	8.1.0	8.0.28	Different sysbench queries [4]
postgresql	14.2	13.2	Different pgbench queries [3]
mongodb	7.0.16	6.0.19	Different YCSB queries [15]

2.1 Experimental Methodology

Hardware Setup. We conduct our experiments on a Linux server with Intel(R) Platinum 8380 (icelake) CPU with 256GB memory. The Intel processor supports Last Branch Record (LBR), avoiding profiling overhead. We conduct each experiment multiple times, at least five times, to ensure that the standard deviation stays within 5% of the average.

Applications and Workloads. Google and Meta report that their applications have multi-megabyte code footprints. Unfortunately, these applications and their corresponding workloads are proprietary. Consequently, similar to prior work [9, 34, 36–38, 41, 47–49, 57, 58, 65, 66, 70, 71, 79], we leverage 5 open-source applications and workloads with large code footprints. We describe these widely-used applications, their stale versions, fresh versions, and workloads in Tab. 1. In particular, we carefully follow the experimental methodology of the state of the art [9]. For example, we compile all applications with the highest level of optimization flags, O3+LTO+AutoFDO for clang and O3+LTO for others, to build baseline stale and fresh binaries. Similarly, we profile both binaries with LBR, collecting stale and fresh profiles, and use them to optimize the fresh binaries with BOLT [57]. For BOLT version, we use LLVM 20, as BOLT is a sub-project of LLVM [40]. Finally, we measure the performance of these binaries to evaluate how effectively prior work addresses profile staleness.

2.2 Importance of using stale profile to optimize data center applications

Large code footprints of data center applications cause frequent I-cache, iTLB, and BTB misses, leading to costly front-end stalls [7, 8, 20, 32, 55, 58, 68], reducing processor efficiency by more than 20% [8, 32]. To reduce front-end stalls, data center providers collect application profiles with Intel's Last Branch Records (LBR) [2] and use these profiles to distinguish frequently executed instructions (*i.e.*, "hot" code) from rarely executed instructions (*i.e.*, "cold" code). By putting hot and cold code separately from each other in the application binary, profile-guided optimizations [13, 57, 65] help data center providers reduce the size of the code footprint by an order of magnitude [58], significantly improving performance efficiency (up to 8% end-to-end [57]). Consequently,

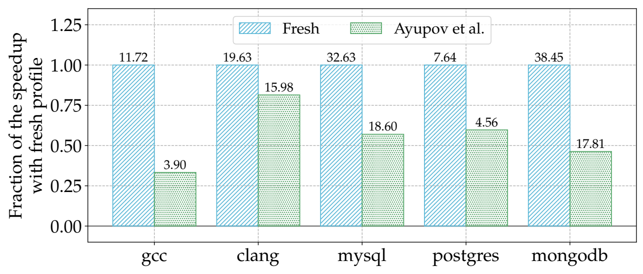


Figure 2. The relative and absolute performance speedups the state of the art [9] achieves with stale profile. We also show the speedups of optimizations with fresh profiles for comparison. The heights of the bars represent the relative speedup, denoting the fraction of the fresh profile's benefits. The values on top of the bars specify absolute speedups.

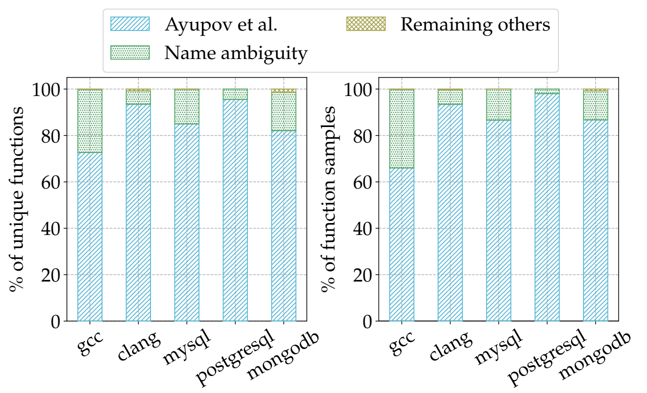


Figure 3. Breakdown of all stale functions and their samples including Ayupov et al.'s [9] mapping results: name ambiguity stops prior work from accurately mapping up to 21% of stale functions and 33.9% of samples.

PGO systems have seen widespread adoption in today's data centers, accelerating half of all CPU cycles [13, 57, 58, 65].

Achieving significant performance benefits with PGO systems requires high-quality profiles that are statistically representative of typical usage scenarios [9, 13, 60]. Otherwise, denoting incorrect code regions as "hot," PGO systems would increase the code footprint, leading to performance degradation [14, 69]. Data center providers avoid such performance regression by employing continuous profiling [9, 13, 60]. Google and Meta continuously profile data center applications over weeks and use these profiles to optimize the subsequent release binary before deployment [9, 13]. As profiles lag behind the most recent source [13], 70-92% of profile samples become stale within weeks [9].

We first study the performance implications of such *profile staleness* for 5 data center applications. We measure the performance speedups BOLT provides with fresh profiles, while also quantifying BOLT's speedups with the state-of-the-art approach [9] to use stale profile. As we show in Fig. 2, while prior work helps BOLT obtain 3.9%-18.60% speedups using stale profiles, they are still only fractions of fresh profiles' speedups (7.64%-38.45%). Next, we investigate why data center applications lose such a significant performance potential due to profile staleness, despite using the state of the art [9].

2.3 What stops the state of the art from using all stale functions?

PGO systems identify hot functions and basic-blocks using LBR profiles. Parsing LBR samples, these systems count the executions of functions and basic-blocks to optimize the code layout. Consequently, optimizing fresh binary's code layout with stale profile's execution counts requires that PGO systems first identify the functions and basic-blocks in the fresh binary that correspond to the functions and basic-blocks from the stale binary. We refer to this identification process as *mapping*. PGO systems also require that this mapping is 1-to-1, *i.e.*, one function in the binary corresponds to only one function in the profile [9]. Now, we briefly describe how the state of the art maps functions and then show its limitations.

State-of-the-art function mapping. The state of the art [9] maps functions by first identifying them using *mangled names* [31]. The mangled name of a C++ function encodes different components, such as the function's namespace, basename, and parameters. While partitioning a function, Link-Time Optimizations (LTO) also append the partition's suffix to the mangled name. For example, gcc-8 has the function, `wi::force_to_size(long*, long const*, unsigned int, unsigned int, unsigned int, signop)(.constprop.3843)`

Here, `wi` is the namespace, `force_to_size` is the basename, `(long*, long const*, unsigned int, unsigned int, unsigned int, signop)` are parameters, and `.constprop.3843` is the LTO suffix. Consequently, the function's mangled name is `_ZN2wi13force_to_sizeEPlPKljjj6signop.constprop.3843`

Assuming little or no changes in functions' mangled names, the state of the art [9] maps stale and fresh functions in a way that minimizes the edit distance of their names.

Key limitation: name ambiguity. Using edit distance of names to map functions stops prior work from correctly mapping functions when those names change in a way that creates name ambiguity. As components like namespace or parameters change, functions' mangled names also change, introducing ambiguous mappings. For example, as we manually inspect source code changes from `mysql-8.0.28` to `mysql-8.1.0`, we observe the renaming of a function from `MYSQLlex(YYSTYPE*, YYLTYPE*, THD*)` to `my_sql_parser_lex (MY_SQL_PARSER_STYPE*, MY_SQL_PARSER_LTYPE*, THD*)`. Instead of mapping them, the state of the art maps the stale function to a different fresh function, `Query_block::save_properties(THD*)` as its mangled name's edit distance is less than the edit distance of the correct function's mangled name.

Implications of name ambiguity. We analyze the implications of such name ambiguity by studying the breakdown of all stale functions and their corresponding samples for 5 data center applications. In particular, we classify them into three categories: (1) functions Ayupov et al. [9] maps correctly, (2) functions with name ambiguity, and (3) remaining others (*e.g.*, functions removed). As we show the results in Fig. 3, we observe that name ambiguity prevents state of the art from mapping a large fraction of functions (up to 21%), corresponding to an even greater fraction of samples (33.9%).

Addressing name ambiguity. To address name ambiguity, we manually inspect how function names change across versions for two data center applications, `mysql` and `clang`. We use the debug information of all stale functions with name ambiguity to identify their source code location, *i.e.*, file name and line number. We carefully examine both stale and fresh versions of the corresponding source code to identify how different components of functions' name change across versions. Tab. 2 shows the corresponding results.

Table 2. Manual inspection of how different components of functions' names change. While file names of these functions from stale profiles rarely change, 1 for `mysql` and 2 for `clang`, inaccuracy of debug information amplifies such changes.

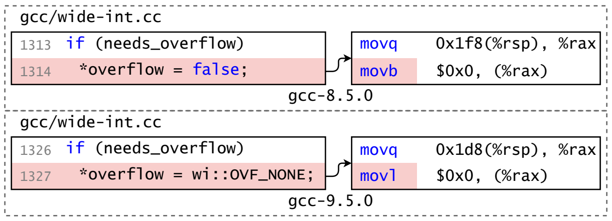
Application	Number of Changes				
	Filename	Namespace	Basename	Parameters	Suffix
`mysql`	11	7	44	97	23
`clang`	20	10	21	76	1

As we show, the frequency of changes differs across different components. Specifically, the namespace changes less than the basename, which changes less than the parameters. For file name, functions rarely move between files based on our manual inspection though inaccuracy of the debug information artificially boosts the frequency of such changes.

Based on the insights of our manual inspection, we develop heuristics to resolve name ambiguity for stale functions. We leverage our heuristics to study how functions change between versions for the remaining data center applications. As we show the results in Tab. 3, we observe a similar trend. Comparing the source file and the components of the functions' names, step by step, we could address name ambiguity.

Table 3. Automated characterization of how different components of functions' names change for different applications.

Application	Number of Changes				
	Filename	Namespace	Basename	Parameters	Suffix
`gcc`	39	84	146	261	845
`postgres`	4	0	19	0	35
`mongodb`	27	0	0	0	0



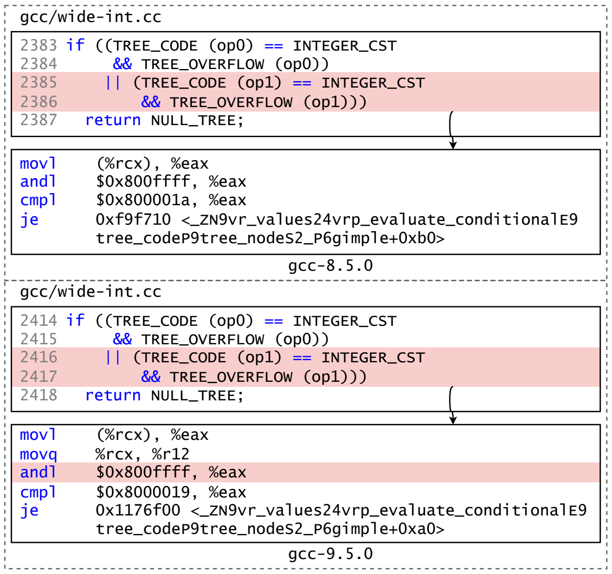
```
if (needs_overflow)
  *overflow = false;
```

```
movq    0x1f8(%rsp), %rax
movb    $0x0, (%rax)
```

```
if (needs_overflow)
  *overflow = wi::OVF_NONE;
```

```
movq    0x1d8(%rsp), %rax
movl    $0x0, (%rax)
```

(a) Key opcode change affecting basic-block's hash values



```
if ((TREE_CODE (op0) == INTEGER_CST
     && TREE_OVERFLOW (op0))
    || (TREE_CODE (op1) == INTEGER_CST
        && TREE_OVERFLOW (op1)))
  return NULL_TREE;
```

```
movl    (%rcx), %eax
andl    $0x800ffff, %eax
cmpl    $0x800001a, %eax
je      0xf9f710 <_ZN9vr_values24vrp_evaluate_conditionalE9
        tree_codeP9tree_nodeS2_P6gimple+0xb0>
```

```
if ((TREE_CODE (op0) == INTEGER_CST
     && TREE_OVERFLOW (op0))
    || (TREE_CODE (op1) == INTEGER_CST
        && TREE_OVERFLOW (op1)))
  return NULL_TREE;
```

```
movl    (%rcx), %eax
movq    %rcx, %r12
andl    $0x800ffff, %eax
cmpl    $0x8000019, %eax
je      0x1176f00 <_ZN9vr_values24vrp_evaluate_conditionalE9
        tree_codeP9tree_nodeS2_P6gimple+0xa0>
```

(b) Instruction addition updating basic-block's hash values

Figure 4. Examples of changes in basic-blocks that creates issues for state of the art [9] to solve profile staleness. Here, highlighted assembly instructions represent key modifications corresponding to the highlighted `C++` codes.

Observation: Due to changes in functions' namespaces, basenames, parameters, and LTO partitions, it is challenging to map functions using names.
Insight: *Comparing functions' contexts (e.g., source file, namespaces, basenames, parameters, and suffixes) step by step enables mapping more functions.*

2.4 What stops the state of the art from mapping all stale basic-blocks?

State-of-the-art basic-block mapping. The state of the art [9] maps basic-blocks using hash values. In particular, prior work computes multiple hash values of a basic-block using its content (*e.g.*, prioritizing opcodes over operands), predecessor, and successor. Maximizing the similarity of these hash values, prior work maps basic-blocks.

Key limitation: content change. Mapping basic-blocks using hash values limits prior work when stale and fresh basic-blocks' hash values do not match due to content changes. For example, as source code changes across versions, they

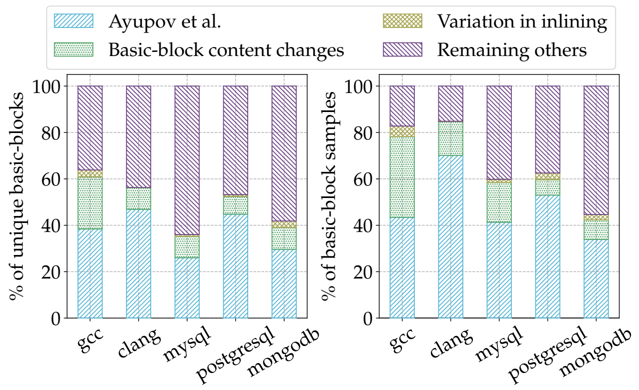


Figure 5. Breakdown of all stale basic-blocks and their samples including Ayupov et al.'s [9] mapping results: variations in basic-block content and function inlining decisions stop prior work from mapping up to 22.5% and 3% of basic-blocks, respectively, corresponding to 34.8% and 4.5% of samples.

result into content changes in corresponding basic-blocks. In Fig. 4, we show examples of such changes in gcc.

In Fig. 4a, as developers introduce a new C++ enumeration type `OVF_NONE` with value 0 to replace `false`, the opcode of a single assembly instruction changes from `movb` to `movl`, also modifying the corresponding basic-block's hash value. Similarly, in Fig. 4b, even with no change of source code, the corresponding basic-block contains an additional assembly instruction, potentially due to different function inlining decisions, altering the basic-block's hash value. Changes in function inlining similarly affect basic-blocks' predecessors and successors, while also moving entire basic-blocks from one function to another. As the state of the art [9] could not identify a basic-block when its hash value or enclosing function change, such content changes stop the techniques from mapping all basic-blocks.

Implications of content change. We analyze the implications of such content change by breaking down all stale basic-blocks and their samples in four categories: (1) basic-blocks Ayupov et al. [9] maps, (2) basic-blocks with content changes, (3) basic-blocks moving from one function to another due to variation in inlining, and (4) remaining others (*e.g.*, basic-blocks removed). As we show the results for 5 data center applications in Fig. 5, we observe that content change stops prior work from mapping up to 22.5% of basic-blocks, representing 34.8% of samples. Similarly, as basic-blocks move across functions due to variation in inlining, up to 3% of basic-blocks and their 4.5% samples remain stale.

Addressing content change. As we show in Fig. 4, the source code includes valuable information to address these content changes. While comparing stale and fresh basic-blocks, we could also compare their corresponding source codes using debug information that practitioners generate, strip before deployment, and store for future usage [13, 57, 58]. Combining source code and debug information, we could identify basic-blocks in inlined functions. Finally, we could separate multiple basic-blocks that originate from the same source code line (*e.g.*, short-circuit conditions [76], loop unrolling [75]) by comparing its preceding and succeeding lines.

Observation: Changes in source code and compilation alter the content, predecessor, and successor of basic-blocks, making it difficult to map them.
Insight: *Using source filenames, line numbers, content, and CFG information enables mapping more basic-blocks with better accuracy.*

3 Wax's Design

We show Wax's design in Fig. 6. Fig. 6a shows Wax's workflow to help PGO techniques optimize the fresh binary using the full potential of the stale profile. Toward this aim, Fig. 6b shows Wax's three modules: (1) *Function Mapping*, (2) *Source Mapping*, and (3) *Basic-block Mapping*. *Function Mapping* creates 1-to-1 mappings of stale and fresh functions using their demangled names and debug information from binaries. *Source Mapping* uses stale and fresh source code to create 1-to-1 mappings of stale and fresh source locations. *Basic-block Mapping* uses assembly instructions and debug information from binaries to generate 1-to-1 mappings of stale and fresh basic-blocks. For all three modules, Wax resolves ambiguity using stale profiles.

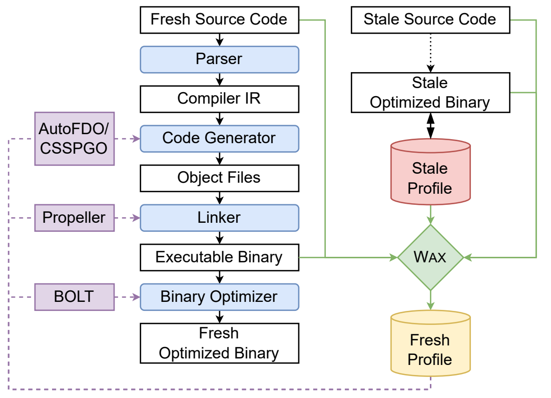
3.1 Function Mapping

As functions' namespace, basename, parameters, and LTO suffix change, Wax maps such functions by comparing their changes, while also using debug information, as shown in Fig. 7. Wax begins by identifying 1-to-1 mappings of source files based on their path names (§3.1.1). For each mapped pair of files, Wax then compares their functions, *i.e.*, functions with debug information pointing to these files. For stale functions, Wax considers only those with profiles, as their execution counts guide optimizations. Then, Wax repeatedly compares different components of function names, keeping newly-found 1-to-1 mappings (§3.1.2).

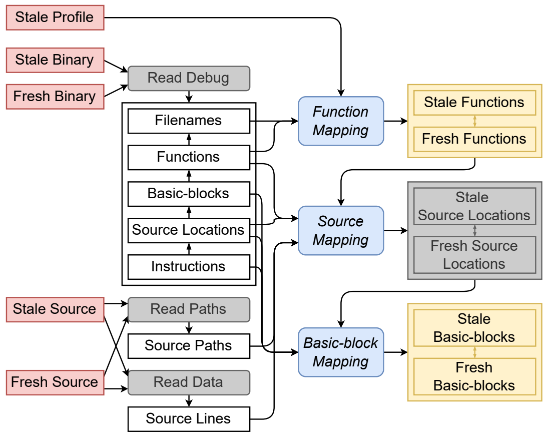
3.1.1 File Mapping by Name. Wax first maps source files based on their path names. As the debug information refers to source files, Wax extracts their names to map files across versions. If multiple files share the same name, Wax maps functions with the same parent directory name. Wax recursively traverses all predecessor directories until it finds a unique mapping. If Wax fails to find a unique mapping for a file, Wax utilizes Levenshtein Similarity [42] metric:

$$lev\_sim(s_1, s_2) = 1 - \frac{edit\_distance(s1, s2)}{\max(length(s1), length(s2))} \quad (1)$$

An example of file mapping is as follows:



(a) WAX's workflow in the context of data center applications's compilation and optimization pipeline



(b) WAX's internal architecture showing interactions among its modules (blue) and their inputs (white) and outputs (yellow).

Figure 6. An overview of WAX's design

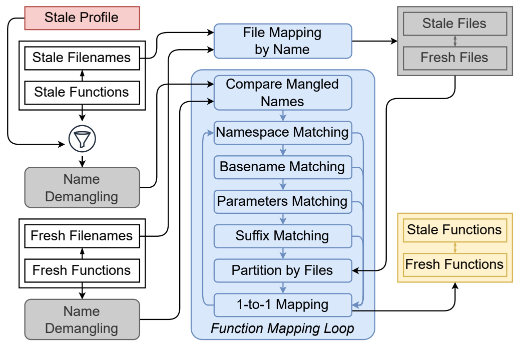


Figure 7. WAX's *Function Mapping* with various steps (blue boxes), their sequence (blue arrows) and intermediate results (black arrows) to produce the final output (yellow boxes).

stale files

```
S0: /home/abc/myapp-1.0/src/query.cpp
S1: /home/abc/myapp-1.0/test/query.cpp
```

fresh files

```
F0: /home/abc/myapp-2.0/src/query.cpp
F1: /home/abc/myapp-2.0/test/query.cpp
F1: /home/abc/myapp-2.0/src/test/query.cpp
```

Here, all files share the same name, `query.cpp`. As `S0` and `F0` share the same parent directory (`src`), WAX maps `S0` and `F0`. Similarly, `S1`, `F1`, and `F2` share the same parent, `test`. However, for all of them, `test`'s parent directories vary (`myapp-1.0`, `myapp-2.0`, and `src`). Therefore, WAX maps `S1` and `F1` as they have the highest Levenshtein Similarity.

3.1.2 Function Mapping Loop.

WAX first compares the mangled names of all stale and fresh functions to map the ones with identical mangled names. To map the rest of the functions, WAX uses a "matching" operation. Defining this matching operation, we describe how WAX performs the operation repeatedly using different components of functions' demangled names. We also discuss some efficiency considerations.

Compare Mangled Names. WAX first maps functions by comparing their mangled names. Specifically, WAX maps all the functions with identical mangled names across versions.

Matching. WAX matches a set of stale functions with another set of fresh functions by performing three steps: (1) comparing all pairs from both sets using a similarity metric, (2) finding the maximum similarity score among all comparisons, and (3) selecting only those pairs with the maximum similarity score. We refer to pairs WAX selects as "matched" pairs. WAX uses Levenshtein Similarity (Eq. 1) to match strings.

Matching functions using different components of their demangled names. As WAX finishes mapping functions with identical mangled names, WAX starts matching the remaining functions based on different components of their demangled names. WAX starts matching with a single component. If the matched pairs are not unique, WAX continues matching with a different component. While matching functions using one component after another, WAX follows the sequence: namespace, basename, parameters, and suffix. After finishing the sequence, if the matching is still ambiguous, WAX uses "Partition by Files." Here, for a mapped stale and fresh source file pair, WAX compares all the functions in the stale file to all the functions in the fresh file using all components of their demangled names. While performing matching and partition by files, if WAX finds unique matched pairs, WAX marks them as mapped using "1-to-1 Mapping." As WAX identifies a new set of 1-to-1 mappings, WAX restarts the matching sequence from the beginning ("Namespace Matching") to perform additional mappings.

Efficiency considerations. For efficiency purposes, WAX first maps stale and fresh functions with identical mangled names before partitioning them based on file names. Similarly, WAX removes all the fresh functions that have corresponding stale functions without profiles with the same namespace, basename, and parameters.

3.2 Source Mapping

Wax's *Source Mapping* module creates 1-to-1 mappings for (1) source lines and (2) source locations. We define a *source line* as a pair of source code file and line. We define a *source location* as a tuple of mapped function, source code file, and line. We include mapped function in source location's definition to account for function inlining, where assembly instructions corresponding to a single source line may appear in multiple functions. As we show in Fig. 8, Wax first uses *Source Line Mapping* (§3.2.1) to map source lines from stale to fresh version based on their content. Wax maps the remaining source lines of the mapped functions that *Source Line Mapping* misses using *Source Location Mapping* (§3.2.2).

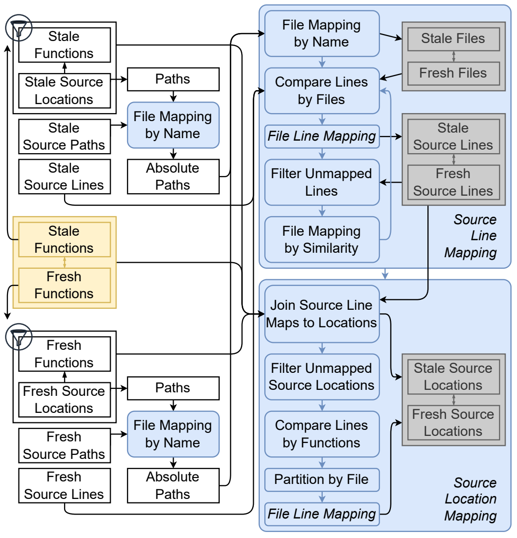


Figure 8. Wax's *Source Mapping* Module

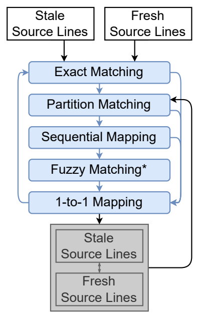


Figure 9. Zooming into *File Line Mapping* of *Source Mapping*

3.2.1 Source Line Mapping.

Wax maps source lines by first mapping source files and then comparing their lines. Specifically, Wax first collects all absolute paths in the source tree and applies *File Mapping by Name* (§3.1.1) to find source file mappings. Using file mappings, Wax reads their content to compare source lines.

Compare Lines by Files. For a mapped pair of stale and fresh files, Wax compares the stale file's source lines against the fresh file's source lines to find line mappings.

File Line Mapping. As illustrated in Fig. 9, Wax's *File Line Mapping* first maps the lines that are identical ("Exact Matching"). Using these mappings, Wax partitions the search space into segments to compare each stale line between a mapped stale line pair against only the fresh lines between those stale line pairs' corresponding fresh lines ("Partition Matching"). Within a partition, Wax maps identical lines sequentially ("Sequential Mapping"), mapping the first stale line to the first fresh line, and so on. Wax maps the remaining lines by fuzzy matching via Levenshtein Similarity (Eq. 1). As Wax creates 1-to-1 line mappings, Wax restarts from the beginning ("Exact Matching") to find new mappings.

Filter Unmapped Lines. Wax uses already mapped source lines to filter the source lines in both stale and fresh versions that are yet to be mapped.

File Mapping by Similarity. Wax maps the remaining unmapped lines, assuming that the developers have moved these lines from one file to another. Wax handles such lines by remapping files using *File Mapping by Similarity*. Here, Wax compares all pairs of stale and fresh source files to measure the number of identical unmapped lines between them. Maximizing the number of such lines, Wax creates 1-to-1 file mappings and restarts mapping lines.

Efficiency Considerations. Wax starts *File Line Mapping* with the full set of lines in the files to help partition the search space based on existing mapping. However, Wax restricts fuzzy matching to lines corresponding to debug information and reduces the number of similarity checks.

3.2.2 Source Location Mapping.

Wax combines source line and function mappings to find source location mappings.

Join Source Line Maps to Locations. While combining line and function mappings, Wax finds two unique scenarios:

- **Unmapped functions having mapped lines.** Assuming function inlining moved basic-blocks from one function to another across versions, Wax addresses such scenarios using *Basic-block Mapping* (§3.3).
- **Mapped functions with missing mapped lines.** Wax addresses such scenarios by rerunning *Source Line Mapping* (§3.2.1) at the granularity of individual function pairs.

Filter Unmapped Source Locations. Wax finds source locations in stale and fresh binaries without a mapping.

Compare Lines by Functions. For all unmapped locations, Wax compares source lines in stale unmapped locations of a stale function against the source lines in fresh unmapped locations of the corresponding mapped fresh function.

Partition by File. Wax only compares stale source lines to fresh source lines such that they belong to mapped stale and fresh file pairs, respectively.

File Line Mapping. Wax uses *File Line Mapping* (§3.2.1) to create 1-to-1 mappings of the unmapped source locations.

3.3 Basic-block Mapping

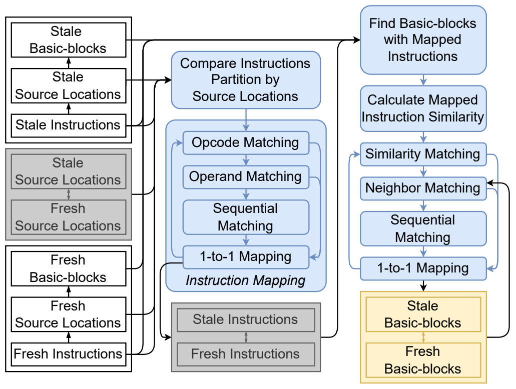


Figure 10. Wax's *Basic-block Mapping* Module

As we describe in §2, state-of-the-art techniques fail to map many basic-blocks due to changes in source code and function inlining. Wax addresses these challenges using *Basic-block Mapping*. As we show in Fig. 10, *Basic-block Mapping* produces 1-to-1 mappings of basic-blocks using the source location mappings and debug information. *Basic-block Mapping* does not rely on the function mappings, as basic-blocks can move from one function to another due to variations in function inlining decisions. A naive approach—mapping basic-blocks by maximizing the number of mapped source locations between them—does not perform well, as the same source location often spans multiple basic-blocks. Instead, Wax first partitions all assembly instructions using their corresponding source locations (§3.3.1), then compares instructions within a partition to identify instruction mappings (§3.3.2), and finally aggregates these instruction mappings to derive basic-block mappings (§3.3.3).

3.3.1 Partition instructions based on their source locations. Wax uses debug information to identify assembly instructions corresponding to a source location and then partitions all assembly instructions using *Source Location Mappings* (§3.2.2). Specifically, for each stale and fresh source location pair, Wax groups their corresponding stale and fresh instructions in the same partition. Within a partition, Wax then compares instructions to identify instruction mappings.

3.3.2 Instruction Mapping. Wax maps instructions by matching their content: opcode and operand. For both strings, Wax compares their similarity (Eq. 1), while also prioritizing opcodes over operands [9, 74]. After matching opcodes and operands, Wax may find multiple matched pairs. Wax breaks such ties using the sequential order of instructions. As Wax finds unique matched pairs, Wax marks them as mapped using "1-to-1 Mapping." As Wax identifies a new set of 1-to-1 mappings, Wax restarts the matching sequence from the beginning to perform additional mappings.

3.3.3 Mapping basic-blocks from instruction mappings. Wax uses 1-to-1 instruction mappings to compute similarity between stale and fresh basic-blocks. Specifically, Wax computes the similarity score for a basic-block pair by summing up the Levenshtein Similarity (Eq. 1) values of the mapped instructions between the basic-block pair. Next, Wax creates 1-to-1 mappings of basic-blocks by maximizing their similarity score. Wax breaks ties of similarity scores among the stale and fresh basic-block pairs by maximizing the number of stale basic-blocks neighbors mapped to fresh basic-blocks neighbors. Finally, Wax resolves any remaining ties by preserving the sequential order of basic-blocks within their containing functions.

Find Basic-blocks with Mapped Instructions. Wax uses instruction mappings to identify all pairs of stale and fresh basic-blocks that share at least one mapped instruction.

Calculate Mapped Instruction Similarity. Wax computes the similarity for all pairs of stale and fresh basic-blocks using their mapped instructions. Specifically, Wax computes the Levenshtein Similarity (Eq. 1) for the mapped instructions and sums them to obtain similarity scores for basic-blocks.

Similarity Matching. Wax aims to maximize the similarity score for stale and fresh basic-blocks. To this end, Wax groups each stale basic-block with fresh basic-block (s) having the maximum similarity score. Similarly, Wax groups each fresh basic-block with stale basic-block (s) having the maximum similarity score. Wax then combines both groups and removes duplicates. Then, Wax directly checks for 1-to-1 mappings. If any such mappings exist, Wax removes them from consideration and restarts running *Similarity Matching*.

Neighbor Matching. If multiple basic-block pairs share the maximum similarity score, Wax uses their control-flow information to break ties. Using control flow, Wax finds the neighbors of the stale and fresh basic-blocks. Among the neighbors, Wax counts the ones that are already mapped. Maximizing the number of such mapped neighbors, Wax finds 1-to-1 mappings. If such mappings exist, Wax removes them and restarts from *Similarity Matching*.

Sequential Matching. Wax resolves any remaining ties by preserving the sequential order of basic-blocks within their containing functions. For example, Wax maps the first unmapped basic-block in a stale function to the first unmapped basic-block in the corresponding fresh function.

Other similarity metrics. While comparing strings (*e.g.*, file names, source codes, and opcodes), Wax supports configurable similarity metrics. Among these similarity metrics, we empirically observe that Levenshtein Similarity [42] outperforms longest common subsequence [73] and prefix matching [21] for data center applications, as developers mainly add new features and bug fixes to these applications [9]. Instead, if applications observe code mutation (*e.g.*, two unrelated variables swap their names), they may benefit from a different similarity metric. In the future, we will investigate how to find a good similarity metric for such applications.

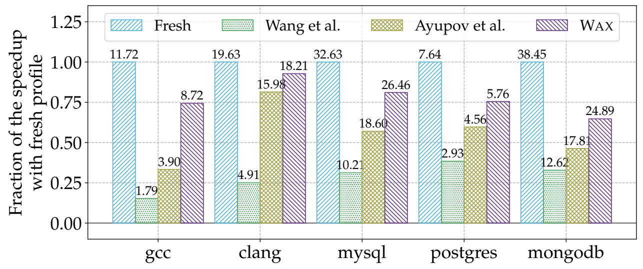


Figure 11. The relative and absolute performance speedups WAX achieves with stale profiles in comparison to prior work (Wang et al. [74] and Ayupov et al. [9]) for different data center applications. For comparison, we also include the speedups corresponding to optimizations with fresh profiles. The heights of the bars represent the relative speedup, denoting the fraction of the fresh profile's benefits. The values on top of the bars specify absolute speedups.

4 Implementation

We implement WAX in 2240 lines of `Python` code using libraries like `polars` [72] and `RapidFuzz` [10]. We read debug information and function symbols from application binaries using `llvm-objdump` and `llvm-readelf` utilities, respectively, while demangling function symbols with the LLVM Itanium demangler. We read basic-blocks and control-flow graphs (CFG) from application binaries by adding 126 lines of `C++` code to LLVM. We open-source our work at https://github.com/ice-rlab/wax.

5 Evaluation

5.1 Methodology

We evaluate WAX using 5 open-source data center applications (as described in §2.1). As presented in §2.1, we list the stale versions, the fresh versions, and workloads for these applications in Tab. 1. Similar to prior work [9], we compile `clang` with `O3+LTO+AutoFDO` and others with `O3+LTO`. We collect profiles for all applications using Intel LBR [2]. Using these profiles, we optimize the code layout of these applications with BOLT.

5.2 Performance analysis

Speedup. We show WAX's performance speedups for 5 data center applications in Fig. 11. For comparison, we also include the performance speedups prior techniques (Wang et al. [74] and Ayupov et al. [9]) provide for these applications. Finally, to understand the upper bound of performance speedups, we also present the performance gains profile-guided optimizations provide with the fresh profile. As we show, WAX achieves an average speedup of 14.32%, attaining 77.14% of the average speedup (18.56%) that profile-guided optimizations provide with fresh profiles. Moreover, WAX significantly outperforms both previous techniques [9, 74].

To understand how WAX achieves performance benefits similar to the fresh profile even with the stale profile, next, we quantify the cache misses WAX reduces across various micro-architectural structures such as L1/L2 I-cache, iTLB, and BTB.

Micro-architectural cache miss reduction. We quantified the speedup of WAX in terms of miss reduction in different micro-architectural caches such as L1 I-cache, L2 I-cache, iTLB, and BTB. We show the results in Fig. 12. In all of these cases, WAX reduces more misses than prior work [9, 74].

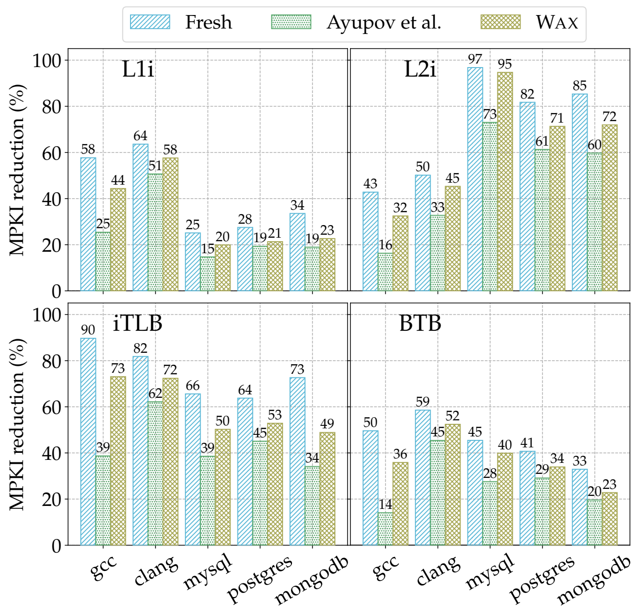


Figure 12. WAX's MPKI reduction for various micro-architectural structures compared to prior work [9, 74].

Performance generalization across different application workloads/inputs. We also investigate whether WAX could optimize data center applications with stale profiles across diverse application workloads/inputs. For our investigation, we leverage eight different `mysql` queries that are readily-available to evaluate the performance of `mysql`. For each query, we compare WAX against the state of the art [9] for oracle and average-case configurations. The oracle configuration leverages the same query to drive `mysql` while collecting the stale profile and evaluating the performance of the fresh binary after optimizing it using the stale profile. On the other hand, the average-case configuration aggregates profiles corresponding to all `mysql` queries. For both configurations, we also measure the performance benefits of using fresh profiles and show the results in Fig. 13. As Fig. 13 shows, WAX performs better than prior work [9] across all `mysql` queries for both oracle and average-case configurations, highlighting that WAX is more resilient to workload/input changes than previous techniques.

Performance generalization across major and minor versions. We evaluate WAX across major and minor version

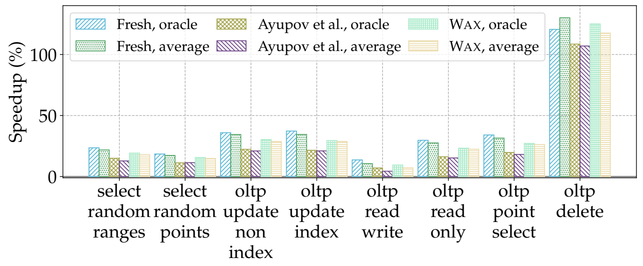


Figure 13. WAX's performance compared to the state of the art (Ayupov et al. [9]), across different application inputs (different `mysql` queries). The oracle configuration uses the same input for profile collection and performance evaluation. The average-case configuration aggregates profiles from all inputs. For both configurations, WAX significantly outperforms prior work across all inputs.

changes for `mysql` and `gcc` in Fig. 14. For `mysql` (Fig. 14a), we optimize `8.4.4` using stale profiles from minor (`8.4.2` and `8.4.0`) and major (`8.3.0`, `8.1.0`, `8.0.28`) versions. Similarly, for `gcc` (Fig. 14b), we optimize `9.5` and `11.5` using profiles from minor (`9.4` to `9.1`) and major versions (`10.5`, `9.5`, `8.5`), respectively. Unlike real-world scenarios, where 70% of profile samples become stale within a week [9], minor versions of `gcc` and `mysql` contain only 2-3% of stale profile samples. Their major versions, on the other hand, include more than 10% of stale samples. As profile staleness increases, WAX's gains over prior work [9, 74] also increase, validating WAX's resiliency. Across all major and minor versions, WAX outperforms previous techniques [9, 74].

Absolute number of samples mapped from stale profiles. We also measure the absolute number of samples WAX and the state of the art map from stale profiles. We list the number of samples for functions and basic-blocks in Tables 4 and 5, respectively. As these tables show, WAX maps significantly more samples than the state of the art. Specifically, WAX maps functions and basic-blocks from all categories quantified in Figures 3 and 5, except "Remaining others".

Table 4. Absolute number of function samples WAX and state of the art map from stale profiles

Application	# of function samples from stale profile		
	Total	Ayupov et al. maps	WAX maps
`gcc`	166943	110127	166669
`clang`	101616	94920	101224
`mysql`	5565071	4817643	5565001
`postgresql`	6541361	6411937	6541361
`mongodb`	1498661	1299119	1486768

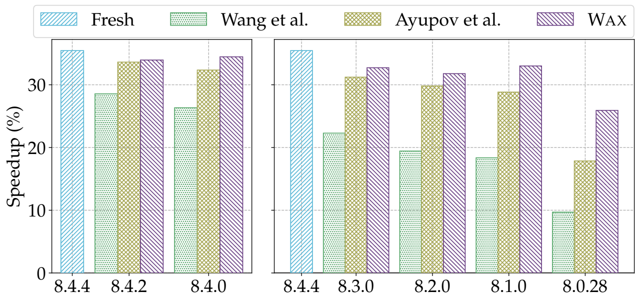
5.3 Sensitivity analysis

We study the sensitivity of WAX's different heuristics by using them separately, *i.e.*, by disabling some of them. To compare performance, we compute the speedup over the prior work [9], while normalizing it by the fresh profile's speedup $((speedup(x) - speedup(\text{Ayupov et al.}))/(speedup(\text{Fresh}) - speedup(\text{Ayupov et al.})) \times 100)$.

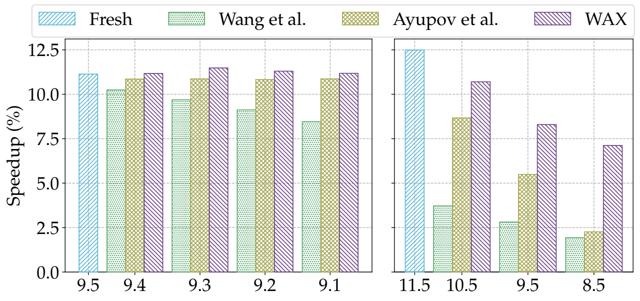
Effect of the two modules. We turn off *Basic-block Mapping* to measure the performance of *Function Mapping* only and compare it against full WAX, with both Function and basic-block mapping. In Fig. 15, we show the speedups that the applications get with WAX over Ayupov et al. [9], where we

Table 5. Absolute number of basic-block samples WAX and state of the art map from stale profiles

Application	# of basic-block samples from stale profile		
	Total	Ayupov et al. maps	WAX maps
`gcc`	3598300	1562178	2977511
`clang`	2033042	1425478	1724122
`mysql`	66883604	27618695	39981112
`postgresql`	125645896	66590782	78589857
`mongodb`	17992518	6096348	8032400



(a) Optimizing `mysql-8.4.4` with stale profiles from minor (left) and major (right) versions.



(b) Optimizing `gcc-9.5` (left) and `gcc-11.5` (right) with stale profiles from minor (left) and major (right) versions, respectively.

Figure 14. The performance speedup WAX provides for the fresh binary compared against prior work (Wang et al. [74] and Ayupov et al. [9]) with different stale profiles corresponding to different minor (left) and major (right) versions of `mysql` (a) and `gcc` (b). WAX significantly outperforms both prior work across all different configurations, gaining benefits comparable to the fresh profile. Furthermore, as the staleness in profile increases, WAX provides more benefits compared to prior work.

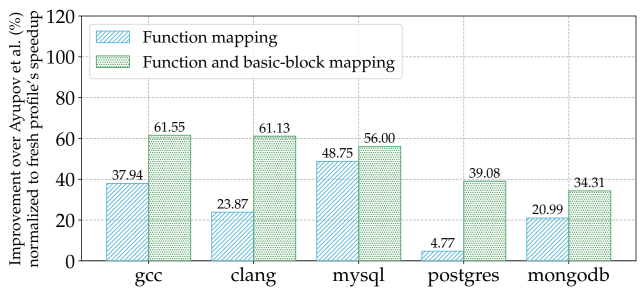


Figure 15. Performance speedup Wax's function and basic-block mapping achieve over Ayupov et al. [9].

find that function mapping provides higher benefit for most applications.

Baseline binaries with additional optimizations. As we describe in §2.1, we compile baseline stale and fresh binaries with the highest level of optimization flags (`O3`). Furthermore, we also evaluate Wax's effectiveness across baselines with additional optimizations, such as function inlining, Link Time Optimizations (LTO), and AutoFDO, as these optimizations exacerbate profile staleness [9]. As we show in Fig. 16, Wax outperforms prior work [9] for all baselines with these additional optimizations.

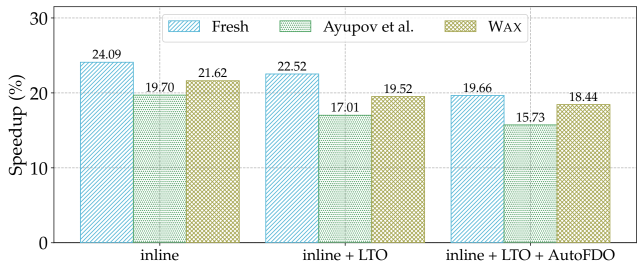


Figure 16. Performance speedup of Wax and Ayupov et al. [9] with different levels of optimizations.

Effect of using *Source Mapping* in the earlier and later stages of Wax. We also evaluate Wax's sensitivity of using *Source Mapping* in the earlier (Wax$_E$) and later (Wax$_L$) stages. Specifically, Wax$_E$ first maps the source code of the stale binary to the source code of the fresh binary, and then maps the corresponding binary instructions of the matching functions and basic blocks. On the other hand, Wax$_L$ leverages *Source Mapping* only in the later stages. For example, during *Function Mapping*, Wax$_L$ uses *Source Mapping* to compare and partition functions by files, as we show in Fig. 7. Similarly, as Fig. 10 shows, Wax$_L$ compares and partitions instructions by source locations during *Basic-block Mapping*. In terms of performance speedups, we do not observe any statistically significant difference between Wax$_E$ and Wax$_L$; p = 27.8% is much higher than the t-test's [19] typical significance level of 5% [16]. Consequently, we also measure how many functions and basic-blocks of the fresh profile both Wax$_E$ and Wax$_L$ successfully map using the stale profile. Tables 6 and 7 show the results, along with the corresponding number of samples present in the fresh profile. As we show, Wax$_E$ and Wax$_L$ map almost identical number of fresh functions and basic-blocks.

Table 6. Effect of using *Source Mapping* in the earlier and later stages of Wax, in terms of *Function Mapping*. Wax converges to same function mappings for both cases.

Application	# of Functions in Fresh profile			# of Function Samples in Fresh profile		
	Total	Wax$_E$	Wax$_L$	Total	Wax$_E$	Wax$_L$
`gcc`	3556	2833	2832	149798	145192	145188
`clang`	4814	3718	3718	132069	126558	126558
`mysql`	1231	977	978	5093786	4586574	4586574
`postgres`	975	886	886	4182079	4112923	4112923
`mongodb`	1934	1566	1565	818161	707266	707062

Table 7. Effect of using *Source Mapping* in the earlier and later stages of Wax, in terms of *Basic-block Mapping*. In both cases, Wax identifies almost the same basic-block mappings.

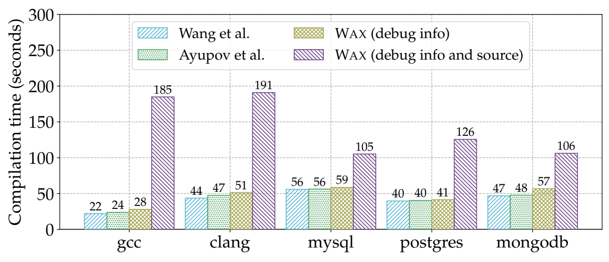
Application	# of Basic-Blocks in Fresh profile			# of Basic-Block Samples in Fresh profile		
	Total	Wax$_E$	Wax$_L$	Total	Wax$_E$	Wax$_L$
`gcc`	64904	50150	50127	1387051	1291319	1291245
`clang`	92461	74511	74511	1057549	1000103	1000103
`mysql`	12378	10190	10193	33800126	31045985	31046049
`postgres`	12433	10893	10895	42965872	40892895	40941260
`mongodb`	16319	13691	13684	5293115	4757992	4757001

5.4 Efficiency analysis

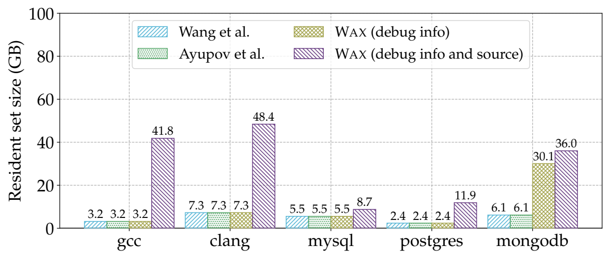
We analyze Wax's efficiency by comparing Wax's compilation time and memory consumption against prior work [9, 74] in Fig. 17. For Wax, we measure and show two configurations: (1) using only debug information (allowing Wax to map only functions) and (2) using both source and debug information (allowing Wax to map both functions and basic-blocks). As we show, the compilation time and memory overhead of Wax with only debug information are similar to prior work, while it still provides significant speedups over them (Fig. 15). Wax using both source and debug information consumes more time and memory to compare source code, assembly instructions, and basic blocks. Still, Wax completes all comparisons within 3.25 minutes with 48.4 GBs of memory.

5.5 Impact of incorrect mapping

We also evaluate the impact of Wax's inaccuracy by finding functions and basic-blocks that Wax maps incorrectly. We identify such incorrect mappings with the help of hot functions and basic-blocks. BOLT puts only hot functions and basic-blocks in the `.text` section after optimization, enabling us to identify them automatically. Similarly, we identify the rest of the functions and basic-blocks (*i.e.*, "*cold*")



(a) Compilation Time



(b) Memory Consumption

Figure 17. WAX's compilation time and memory consumption while using source and debug information in comparison to prior work [9, 74]. While using only debug information to guide function mapping, WAX consumes time and memory comparable to prior work.

that BOLT puts in different sections (*e.g.*, `.text.cold` and `.bolt.org.text`). For each function and basic-block, we compare their hot or cold status after optimizing with fresh and stale profiles. Aggregating these comparison results for all functions and basic-blocks, we identify incorrect mappings and compute the following quantities:

		Stale	
		Hot	*Cold*
Fresh	*Hot*	True Positive (TP)	False Negative (FN)
	Cold	False Positive (FP)	True Negative (TN)

Table 8. Numbers of functions WAX and prior work [9] map accurately (TP, TN) and inaccurately (FN, FP) using stale profile, compared to fresh profile.

Application	Ayupov et al.				WAX			
	TP	FN	FP	TN	TP	FN	FP	TN
gcc	2077	1478	559	44705	2837	718	768	44496
clang	2333	1000	1974	117095	2453	880	2158	116911
mysql	817	414	99	72858	979	252	106	72851
postgres	846	129	62	17261	888	87	63	17260
mongodb	1318	616	218	161515	1572	362	302	161431

We report these quantities (TP, FN, FP, and TN) for WAX's *Function Mapping* and *Basic-block Mapping* in Tables 8 and 9, respectively. We also use these quantities to compare WAX against Ayupov et al. [9]. For TP and TN, higher is better. For FN and FP, lower is better. As these tables show, WAX always outperforms Ayupov et al. in terms of TP and FN. For FP and

Table 9. Numbers of basic-blocks WAX and prior work [9] map accurately (TP, TN) and inaccurately (FN, FP) using stale profile, compared to fresh profile.

Application	Ayupov et al.				WAX			
	TP	FN	FP	TN	TP	FN	FP	TN
gcc	123443	134042	19912	30011	218387	39098	28280	21643
clang	149771	214996	153822	246288	158175	206592	166864	233246
mysql	25328	18063	2264	5327	36467	6924	2116	5475
postgres	40094	5488	1266	2163	42660	2922	1234	2195
mongodb	37754	19838	6082	34357	49960	7632	7331	33108

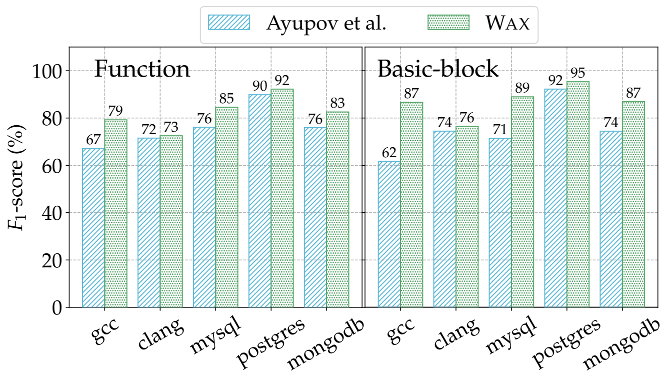


Figure 18. ***F_1-score*** of WAX's *Function Mapping* and *Basic-block Mapping* compared to the state of the art [9].

TN, Ayupov et al. is sometimes better than WAX. We compare the significance of TP and FN against the significance of FP and TN by aggregating them with the following metrics: $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$; $Precision = \frac{TP}{TP+FP}$; $Recall = \frac{TP}{TP+FN}$; and $F_1\text{-}score = \frac{2 \cdot Precision \cdot Recall}{Precision+Recall}$.

Next, we investigate how end-to-end performance correlates with these metrics by measuring their Pearson correlation coefficients (r).

Metric	*Accuracy*	*Precision*	*Recall*	*F_1-score*
Pearson r	0.91	0.59	0.95	0.95

As we show, *Recall* and *F_1-score* correlate highly with the end-to-end speedup. Consequently, we compare WAX against prior work using *F_1-score* in Fig. 18. Fig. 18 shows that WAX always outperforms prior work [9].

Incorrect mappings due to missing debug information. We further characterize WAX's inaccuracy due to missing debug information. While data center applications' optimization pipeline (Fig. 6a) aims to preserve debug information [1] as systems like AutoFDO require such information to map profile back to intermediate representation [13], different optimizations (*e.g.*, function inlining and LTO) exacerbate the quality of debug information [29]. Still, such exacerbation does not hamper WAX's recovery of a large percentage of function and basic-block mappings. As we show in Fig. 7, while missing debug information could stop WAX's *Function Mapping* from comparing or partitioning functions by source files, Wax still maps functions by matching namespace, basename, parameters, and suffix as they always exist in stale and fresh functions' demangled names. Similarly, as Fig. 10 shows, while missing debug information could prevent Wax's *Basic-block Mapping* from comparing and partitioning instructions by source locations, Wax still maps instructions and basic-blocks by matching instruction opcodes, instruction operands, and basic-block neighbors as they are part of stale and fresh binaries. Quantitatively, we also identify how many identical functions and basic-blocks Wax fails to map due to missing debug information. As we show in tables 10 and 11, Wax maps almost all identical functions and basic-blocks across 5 data center applications. Even in the worst cases, Wax fails to map only 0.04% of identical functions (gcc) and 1.5% of samples of identical functions (mongodb). Similarly, Wax fails to map only 1-5% of identical basic-blocks and 0.002-4.4% of samples of identical basic-blocks. Consequently, poor debug information does not stop Wax from obtaining 65%-93% of fresh profiles' benefits.

Table 10. Numbers of identical functions across stale and fresh versions Wax fails to map. At most, Wax could not use 0.04% of identical functions or 1.5% of samples.

Application	# of Identical Functions		# of Samples from Identical Functions	
	Total	Wax fails to map	Total	Wax fails to map
`gcc`	26227	10	100370	334
`clang`	57638	1	110059	2
`mysql`	51870	3	4845097	26085
`postgres`	15668	5	6411937	6834
`mongodb`	94149	11	1307523	19600

Table 11. Numbers of identical basic-blocks across stale and fresh versions Wax fails to map. Wax could not map only 1-5% of identical basic-blocks, missing 0.002-4.4% of samples.

Application	# of Identical Basic-Blocks		# of Samples from Identical Basic-Blocks	
	Total	Wax fails to map	Total	Wax fails to map
`gcc`	3429	172	97920	4316
`clang`	9705	277	138482	3
`mysql`	7748	92	11016218	15562
`postgres`	7397	135	29693497	5133
`mongodb`	12380	128	4125618	19861

5.6 Applying Wax to Propeller

We show Wax's broad applicability by using Wax's mapping to Propeller [24, 65] while optimizing `clang-15` with `clang-14`'s profile [25]. As we show in Table 12, Wax helps Propeller optimize with stale profiles, while also outperforming the state of the art [9].

Table 12. Propeller's [65] speedup for `clang` with fresh and stale profiles mapped using prior work [9] and Wax.

Profile	Fresh	Ayupov et al.	Wax
Speedup (%)	5.92%	5.38%	6.44%

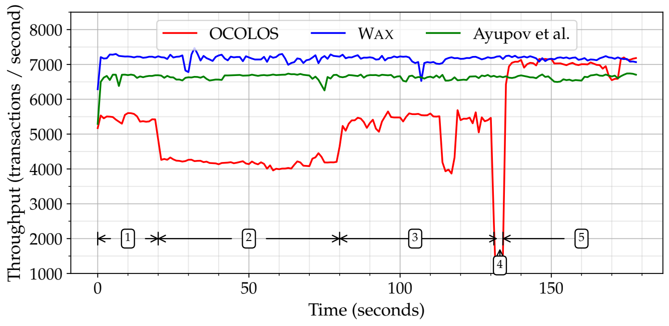


Figure 19. Throughput of `mysql read_only` before, during, and after OCOLOS's code replacement [77, 78] compared to prior work [9] and Wax.

5.7 Comparison against OCOLOS

OCOLOS [77, 78] is an online code layout optimization system to optimize running C/C++ applications in a profile-guided manner. As OCOLOS optimizes running applications online, it could leverage fresh profiles instead of stale profiles. Therefore, we compare Wax's effectiveness against OCOLOS for `mysql` in Fig. 19. As we show in Fig. 19, Wax significantly outperforms OCOLOS and Ayupov et al. during OCOLOS's *warm-up phase* $\boxed{1}$, *profile collection phase* $\boxed{2}$, *profile conversion phase* $\boxed{3}$, and *code replacement phase* $\boxed{4}$. Even after OCOLOS finishes optimizing `mysql` with the fresh profile (phase $\boxed{5}$), Wax still outperforms OCOLOS, although Wax uses the stale profile. With OCOLOS, function pointers in registers and memory always point to unoptimized functions [77, 78], even in phase $\boxed{5}$, causing OCOLOS to fall short of Wax. In contrast, prior work [9] falls short of OCOLOS in phase $\boxed{5}$.

6 Related Work

Existing techniques leverage profile to perform a wide range of optimizations, including function inlining and outlining [17, 43], function layout optimizations [30, 56, 59], function merging [62, 63], basic-block reordering [41, 51, 59], loop optimization [61], and register allocation [22, 45]. Various compilers and binary optimizers implement these techniques for different static and dynamic languages [54, 55], and at different stages of the compilation pipeline, such as code generation [13], linking [44, 64], and post-link optimization [46, 57, 65]. Intel's Last Branch Record (LBR) technology [13, 52] provides a low-overhead mechanism to collect representative profiles.

BMAT [74], Stale Profile Matching [9], Beetle [48], and Hydra [23] proposed different techniques to optimize a fresh version of the binary using a stale version's profiles. All these techniques use the binary information only to create a mapping from stale to fresh binary. In that way, these works relate to binary code similarity [6, 26]. Unfortunately, solving the profile staleness problem only using binary code similarity makes the problem unnecessarily harder, as debug and source code information are always present in the pipeline of profile-guided optimization systems, as we show in Fig. 6a. Consequently, Wax uses source and debug information to relax profile staleness, outperforming prior binary code similarity techniques.

Recent work [50, 77, 78] proposed online systems to avoid profile staleness by profiling and optimizing the fresh binary online. As we show in Fig. 19, even with a stale profile, Wax outperforms such an online system as it contains pointers to an unoptimized version [77, 78].

7 Conclusion

This paper characterized several key challenges of using stale profiles to optimize fresh binaries. We addressed these challenges with Wax, a novel technique to map stale profiles to fresh binaries using source code and debug information. Our evaluation showed that Wax enabled effective optimizations using stale profiles, achieving significant speedups (5.76%-26.46%) across 5 data center applications.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work was supported by generous gifts from Google. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We thank National Science Foundation's Chameleon Cloud [33] and CloudLab [18] for providing compute nodes where we run our experiments to obtain results presented in this paper.

References

[1] [n. d.]. BOLT Update Debug Sections. https://tinyurl.com/5bszcd5x.

[2] [n. d.]. An Introduction to Last Branch Records. https://lwn.net/Articles/680985/.

[3] [n. d.]. PostgreSQL: Documentation: 14: pgbench. https://www.postgresql.org/docs/current/pgbench.html [Online; accessed 19-Nov-2021].

[4] [n. d.]. Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[5] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 777–790.

[6] Saed Alrabaee, Mourad Debbabi, Paria Shirani, Lingyu Wang, Amr Youssef, Ashkan Rahimian, Lina Nouh, Djedjiga Mouheb, He Huang, and Aiman Hanna. 2020. Binary analysis overview. *Binary Code Fingerprinting for Cybersecurity: Application to Malicious Code Fingerprinting* (2020), 7–44.

[7] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 643–656.

[8] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th ISCA*.

[9] Amir Ayupov, Maksim Panchenko, and Sergey Pupyrev. 2024. Stale Profile Matching. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. 162–173.

[10] Max Bachmann. 2025. *rapidfuzz/RapidFuzz: Release 3.13.0*. doi:10.5281/zenodo.15133267

[11] Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, and Ollie Wild. 2007. WHOPR-Fast and Scalable Whole Program Optimizations in GCC. *Initial Draft* 12 (2007).

[12] Dimitrios Chasapis, Georgios Vavouliotis, Daniel A Jiménez, and Marc Casas. 2025. Instruction-Aware Cooperative TLB and Cache Replacement Policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 619–636.

[13] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *CGO*.

[14] William Y Chen, Pohua P. Chang, Thomas M Conte, and Wen-mei W. Hwu. 1993. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput.* 42, 9 (1993), 1045–1057.

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[16] Charlie Curtsinger and Emery D Berger. 2013. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 219–228.

[17] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. Inlining for Code Size Reduction. In *Proceedings of the 25th Brazilian Symposium on Programming Languages* (Joinville, Brazil) *(SBLP '21)*. Association for Computing Machinery, New York, NY, USA, 17–24. doi:10.1145/3475061.3475081

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[19] William Feller. 1991. *An introduction to probability theory and its applications, Volume 2*. Vol. 2. John Wiley & Sons.

[20] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices* 47, 4 (2012), 37–48.

[21] Edward Fredkin. 1960. Trie memory. *Commun. ACM* 3, 9 (1960), 490–499.

[22] Andreas Fried, Maximilian Stemmer-Grabow, and Julian Wachter. 2023. Register Allocation for Compressed ISAs in LLVM. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 122–132. doi:10.1145/3578360.3580261

[23] Elisa Fröhlich, Angélica Aparecida Moreira, and Fernando M. Quintão Pereira. 2026. Automatic Propagation of Profile Information through the Optimization Pipeline. *Proceedings of the ACM on Programming Languages* 1, 1, Article 1 (Jan. 2026), 26 pages.

[24] Google. 2020. Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker. https://github.com/google/llvm-propeller.

[25] Google. 2021. How to optimize clang with Propeller. https://github.com/google/autofdo/blob/master/docs/OptimizeClangO3WithPropeller.md. Retrieved June 18, 2025.

[26] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *Acm computing surveys (csur)* 54, 3 (2021), 1–38.

[27] Wenlei He. 2020. [RFC] Context-sensitive Sample PGO with Pseudo-Instrumentation. https://groups.google.com/g/llvm-dev/c/1p1rdYbL93s/m/AN_hgPmnAwAJ. Google Groups discussion on llvm-dev.

[28] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. 2022. Profile inference revisited. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–24.

[29] Wenlei He, Hongtao Yu, Lei Wang, and Taewook Oh. 2024. Revamping Sampling-Based PGO with Context-Sensitivity and Pseudo-instrumentation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 322–333.

[30] Ellis Hoag, Kyungwoo Lee, Julián Mestre, and Sergey Pupyrev. 2023. Optimizing Function Layout for Mobile Applications. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Orlando, FL, USA) *(LCTES 2023)*. Association for Computing Machinery, New York, NY, USA, 52–63. doi:10.1145/3589610.3596277

[31] Itanium C++ ABI Committee. [n. d.]. Itanium C++ ABI, Section 5: Mangling. https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling. Accessed: 2025-08-12.

[32] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd ISCA*.

[33] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[34] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 816–829.

[35] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 163–181.

[36] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159.

[37] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci. 2022. Whisper: Profile-guided branch misprediction elimination for data center applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 19–34.

[38] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *Proceedings (to appear) of the 48th International Symposium on Computer Architecture (ISCA) (ISCA 2021)*.

[39] Razya Ladelsky and Mircea Namolaru. 2005. Interprocedural constant propagation and method versioning in GCC. In *GCC and GNU Toolchain Developers' Summit*.

[40] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[41] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction*. 65–75.

[42] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[43] Roy Levin, Ilan Newman, and Gadi Haber. 2008. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 291–304.

[44] Gai Liu, Umar Farooq, Chengyan Zhao, Xia Liu, and Nian Sun. 2023. Linker Code Size Optimization for Native Mobile Applications. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 168–179. doi:10.1145/3578360.3580256

[45] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 17 (July 2019), 53 pages. doi:10.1145/3332373

[46] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: a post-link optimizer for the Intel/spl reg/Itanium/spl reg/architecture. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 15–26.

[47] Chaitanya Mamatha Ananda, Rajiv Gupta, Sriraman Tallam, Han Shen, and Xinliang David Li. 2025. PreFix: Optimizing the Performance of Heap-Intensive Applications. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 405–417.

[48] Angelica Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2023. BEETLE: A Feature-Based Approach to Reduce Staleness in Profile Data. (June 2023). https://www.microsoft.com/en-us/research/publication/beetle-a-feature-based-approach-to-reduce-staleness-in-profile-data/

[49] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2021. VESPA: static profiling for binary optimization. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–28.

[50] Wenlong Mu, Yue Tang, Bo Huang, and Jianmei Guo. 2025. AOBO: A Fast-Switching Online Binary Optimizer on AArch64. *ACM Transactions on Architecture and Code Optimization* (2025).

[51] Andy Newell and Sergey Pupyrev. 2020. Improved basic block reordering. *IEEE Trans. Comput.* 69, 12 (2020), 1784–1794.

[52] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a base of trust with performance counters for enterprise workloads. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 541–548.

[53] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. Udp: Utility-driven fetch directed instruction prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1188–1201.

[54] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–165.

[55] Guilherme Ottoni and Bin Liu. [n. d.]. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 340–350.

[56] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 233–244.

[57] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[58] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 119–130.

[59] Karl Pettis and Robert C Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 16–27.

[60] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro* 30, 4 (2010), 65–79.

[61] Rodrigo C. O. Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop Rolling for Code Size Reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 217–229. doi:10.1109/CGO53902.2022.9741256

[62] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Virtual, Canada) *(LCTES 2021)*. Association for Computing Machinery, New York, NY, USA, 110–121. doi:10.1145/3461648.3463852

[63] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective function merging in the SSA form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 854–868. doi:10.1145/3385412.3386030

[64] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, Vol. 114.

[65] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 617–631.

[66] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 742–756.

[67] Niranjan K Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. PDede: Partitioned, Deduplicated, Delta Branch Target Buffer. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 779–791.

[68] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[69] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 977–989.

[70] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. 2022. One profile fits all: Profile-guided linux kernel optimizations for data center applications. *ACM SIGOPS Operating Systems Review* 56, 1 (2022), 26–33.

[71] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. 2021. Morrigan: A Composite Instruction TLB Prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1138–1153.

[72] Ritchie Vink et al. 2023. *Polars: Fast multi-threaded DataFrame library in Rust and Python*. doi:10.5281/zenodo.8293363

[73] Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)* 21, 1 (1974), 168–173.

[74] Zheng Wang, Ken Pierce, and Scott McFarling. 2000. Bmat-a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism* 2 (2000), 1–20.

[75] Wikipedia contributors. 2025. Loop unrolling — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Loop_unrolling&oldid=1276558448. [Online; accessed 18-August-2025].

[76] Wikipedia contributors. 2025. Short-circuit evaluation — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Short-circuit_evaluation&oldid=1305642517. [Online; accessed 18-August-2025].

[77] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. 2022. Ocolos: Online code layout optimizations. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 530–545.

[78] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. 2023. Online code layout optimizations via OCOLOS. *IEEE Micro* 43, 4 (2023), 71–79.

[79] Yufeng Zhou, Xiaowan Dong, Alan L Cox, and Sandhya Dwarkadas. 2019. On the impact of instruction address translation overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 106–116.

[80] Kan Zhu, Yilong Zhao, Yufei Gao, Peter Braun, Tanvir Ahmed Khan, Heiner Litz, Baris Kasikci, and Shuwen Deng. 2025. From Optimal to Practical: Efficient Micro-op Cache Replacement Policies for Data Center Applications. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 716–731.