Finding and Understanding Missed Optimizations in WebAssembly Optimizer (Experience Paper)

RUIYANG XU*, East China Normal University, China
ZETAO FAN*, East China Normal University, China
SHAN HUANG, East China Normal University, China
TING SU†, East China Normal University, China

As WebAssembly (Wasm) expands from web applications to high-performance domains, the standard optimizer, `wasm-opt`, is critical but frequently suffers from missed optimizations (MOs). This paper presents an experience report on detecting MOs in `wasm-opt` and understanding their root cause through the lens of Wasm's tree-structured intermediate representation (tree IR). To this end, we adapt an established marker-based technique from C compilers, overcoming the constraints of Wasm's structured control flow via a novel structure-aware instrumentation strategy. Complementing this, we design a cross-optimization differential testing strategy leveraging the monotonicity of optimization levels as an oracle. Together, these strategies enable the systematic identification of fine-grained MOs that are overlooked by existing cross-architecture methods. Our evaluation uncovered 24 distinct MOs (20 fixed, 100% confirmation rate, 0% false positive rate), demonstrating the high actionability and effectiveness of our approach. The performance impact, especially in code size, yields an average 1.30% improvement and no regressions on the Emscripten benchmark suite, further proving their practical value. Beyond detection, our analysis distills three practical lessons for designing MO testing techniques and understanding the optimization trade-offs imposed by `wasm-opt`'s tree IR.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Software Testing, WebAssembly, Compiler Optimization

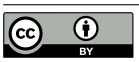
ACM Reference Format:
Ruiyang Xu, Zetao Fan, Shan Huang, and Ting Su. 2026. Finding and Understanding Missed Optimizations in WebAssembly Optimizer (Experience Paper). *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA085 (October 2026), 23 pages. https://doi.org/10.1145/3832176

1 Introduction

WebAssembly (Wasm) is a safe, portable, low-level bytecode format designed for efficient execution and compact representation [15]. Although initially designed to accelerate client-side web applications [15, 36, 50], Wasm has expanded beyond the browser to diverse non-web domains that demand high performance such as edge computing and IoT [2, 10–12, 28, 31, 57, 63]. While Wasm is expected to achieve near-native performance [15], in practice, current implementations frequently

*Ruiyang Xu and Zetao Fan contributed equally to this work.
†Ting Su is the corresponding author.

Authors' Contact Information: Ruiyang Xu, Software Engineering Institute, East China Normal University, Shanghai, China, xury2002@stu.ecnu.edu.cn; Zetao Fan, Software Engineering Institute, East China Normal University, Shanghai, China, 51285902042@stu.ecnu.edu.cn; Shan Huang, Software Engineering Institute, East China Normal University, Shanghai, China, shan.huang@stu.ecnu.edu.cn; Ting Su, Software Engineering Institute, East China Normal University, Shanghai, China, tsu@sei.ecnu.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA085
https://doi.org/10.1145/3832176

fail to meet these expectations [18, 31, 38, 40]. For example, Jangda et al. [20] reported an average slowdown of 1.45×–1.55× in browsers, widening to 1.59×–9.57× in standalone runtimes [51].

`wasm-opt` and Missed Optimizations. To bridge this performance gap, `wasm-opt` [3] serves as the standard Wasm optimizer and is widely integrated into industrial compiler toolchains [8, 14, 42–44, 48]. Given that Wasm acts as a universal compilation target for high-level languages (*e.g.,* C/C++, TypeScript, Rust), `wasm-opt` plays a critical role in enhancing the entire pipeline by improving speed and code size independent of the source language. It implements a comprehensive suite of optimizations, ranging from general-purpose passes (*e.g.,* inlining) to Wasm-specific transformations. However, like other optimizers [1, 4, 33, 45], `wasm-opt` suffers from missed optimization (MOs), where code fails to reach its most efficient form despite the optimizer's ability to do so. For instance, we observed that `wasm-opt` can fail to perform even elementary algebraic simplifications, such as folding an unsigned comparison against zero ($u(x) \geq 0$) when the expression involves side effects, thereby leaving redundant logic in the final output.

Prior Work and Limitations. Liu *et al.* [26] perform the systematic detection of MOs in `wasm-opt` via differential testing against native x86 executable. However, their approach has two key limitations. First, cross-architecture comparison between x86 and Wasm introduces inherent false positive, as inconsistencies may be due to architectural discrepancies (*e.g.,* computation models) rather than MOs. Second, even when correctly identified, many root causes lie in how upstream compilers lower source code to Wasm (*e.g.,* placing variables in linear memory instead of the global section), making them "technically infeasible" to address within `wasm-opt` itself. This limits the actionability of their findings. Additionally, their reliance on coarse-grained runtime traces cannot *systematically detect* fine-grained MOs hidden within unexecuted (dead) code blocks, and it offers little insight into `wasm-opt`'s structural issues that cause such misses.

Applied Approach. To address these limitations, we shift detection perspective from *cross-architecture* to *cross-optimization* (*e.g.,* between `-O2` and `-O3` in `wasm-opt`). The insight is that leveraging intra-optimizer comparisons ensures that identified gaps stem from the transformation logic itself rather than architectural or lowering-level differences. To this end, we apply an existing MO testing technique, originally used for modern C compiler [45], to detect fine-grained MOs within `wasm-opt`. This approach starts by injecting markers into the code, and then optimizing the instrumented code under different optimization levels. It finally collects the surviving markers in the output and uses them to identify MOs. The oracle is based on the *monotonicity of optimization levels*: a higher level should be at least as effective as a lower one in eliminating dead code. Consequently, any marker pruned at higher level should logically also be removed at level. Formally, let B denote the initial Wasm binary instrumentation. We define B_L and B_H as the optimized binaries generated by applying `wasm-opt` at a lower level L (*e.g.,* `-O2`) and a higher level H (*e.g.,* `-O3`), respectively. Let $\mathcal{M}(B_L)$ and $\mathcal{M}(B_H)$ denote the sets of surviving markers at each level. Under the principle of optimization monotonicity, we expect the inclusion property: $\mathcal{M}(B_H) \subseteq \mathcal{M}(B_L)$. A violation, where a marker is present in B_H but missing in B_L ($m \in \mathcal{M}(B_H) \wedge m \notin \mathcal{M}(B_L)$), flags an MO.

Our Adaptation. However, the existing technique cannot be directly applied to `wasm-opt` because of a fundamental implementation obstacle: *the strict nesting constraints of its tree-structured IR* (tree IR). Unlike in flat IRs where markers can be appended linearly, instrumentation must preserve the precise nesting and control-flow semantics of Wasm's blocks, loops, and conditionals. We therefore developed a *structure-aware instrumentation strategy* that inserts markers as valid nodes within the nested expression tree, for example, placing a marker in the fall-through path of a `br_if` instruction. This ensures semantic validity while making hidden MOs within these structures observable.

Evaluation and Lesson Learned. We evaluated this adapted approach by continuously testing the latest version of `wasm-opt` (`v122` [52] to `v124` [53]). This effort uncovered 24 distinct, previously unknown MOs. To date, all of these have been confirmed by the developers, with 20 already fixed.

Beyond merely detecting bugs, our systematic root cause analysis reveals several limitations in `wasm-opt`. Most notably, we demonstrate how `wasm-opt`'s tree-structured IR inherently entangles computation with side effects, forcing overly conservative optimization strategies. Based on these findings, we distill practical lessons for the broader compiler engineering community. We highlight the effectiveness of cross-optimization testing for filtering architectural noise in single-optimizer ecosystems, and demonstrate how dead code elimination can serve as a highly reliable optimization oracle. Finally, a systematic analysis of historical issues confirms that the MO categories we discovered fill a critical, long-standing gap in existing Wasm performance research.

Contributions. In summary, this paper makes the following contributions:

(1) We adapt and extend DCE-based testing into a Wasm-native, structure-aware detection method to expose fine-grained missed optimizations (Section 4).
(2) We identify 24 confirmed missed optimizations in `wasm-opt`, 20 of which have been fixed, spanning multiple optimizations levels and passes (Section 5).
(3) We reveal fundamental limitations of `wasm-opt` and deliver a series of practical lessons and potential future directions for `wasm-opt` and similar optimizers (Section 6).

2 Background and Motivating Example

2.1 WebAssembly Workflow

In this section, we provide a high-level overview of the WebAssembly (Wasm) workflow shown in Figure 1, comprising three key stages: source compilation, binary optimization, and final execution. In particular, we highlight the pivotal role of `wasm-opt` in the optimization stage.

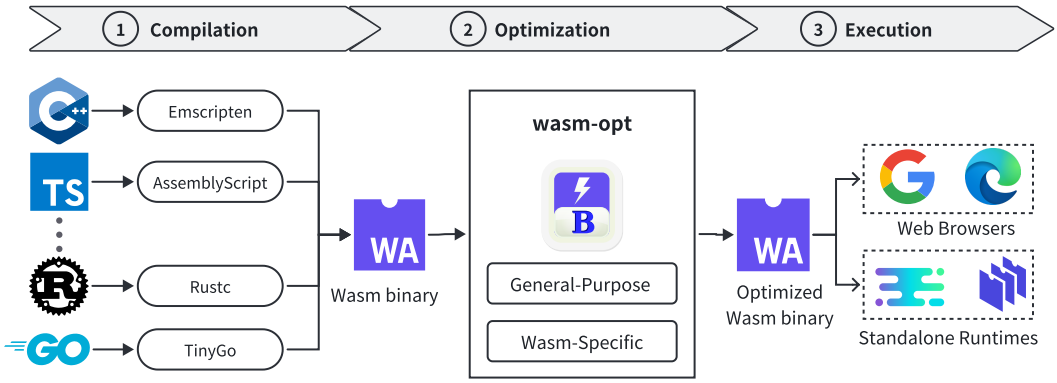


Fig. 1. Wasm workflow.

Compilation. In stage ①, source code written in high-level languages (*e.g.,* C++, TypeScript, Rust) is compiled into Wasm binaries via their respective compilers (*e.g.,* Emscripten, AssemblyScript, Rustc). Despite Wasm achieving extensive language support, these compilers prioritize optimizations tailored to their source languages, *often treating Wasm merely as a generic target.* Consequently, they tend to generate code patterns suited for native architectures (*e.g.,* x86, ARM) that prove inefficient for the Wasm execution model, resulting in bloated and slow binaries.

Optimization. Stage ② focuses on optimizing the Wasm binaries to enhance performance and minimize code size, regardless of the high-level language from which they were compiled in stage ①. This is achieved by `wasm-opt`, the sole official Wasm optimizer, which applies a comprehensive suite of optimization passes. These passes are broadly categorized into:

(1) *General-purpose*: These include classical, architecture-agnostic optimizations such as dead code elimination and function inlining. While the compilers in stage ① also perform these passes, `wasm-opt` re-applies them to capture optimization opportunities introduced during the translation process and interact with Wasm-specific passes for further optimization.
(2) *Wasm-specific*: These optimizations are tailored to the unique architecture and binary format of Wasm, addressing the specific constraints and requirements of the Wasm virtual machine. For instance, it optimizes *variable management* because Wasm operates without hardware registers, relying instead on indexed local variables to store intermediate values. It also refines *control flow structuring* to flatten the rigid nesting layers often produced by high-level compilers (a structural characteristic we detail in Section 2.2). Additionally, to optimize interactions with Wasm's *linear memory* (a contiguous, byte-addressable array), `wasm-opt` provides passes to leverage efficient bulk memory operations, such as `memory.copy`, to replace slower loop-based data manipulation. By bridging the gap between generic compilation output and the specific demands of the virtual machine, `wasm-opt` plays a pivotal role in ensuring that Wasm binaries achieve the compactness and performance required for production deployment.

Execution. As shown in Figure 1, Wasm binaries can execute in either Web browsers or standalone runtimes. In Web environments, browser engines (*e.g.,* V8 for the Chrome browser) compile Wasm binaries into native machine code and provide JavaScript glue code as an interface for accessing Web APIs. This achieves near-native computational performance while maintaining a robust security sandbox. Conversely, standalone runtimes like Wasmtime and Wasmer implement the WebAssembly System Interface (WASI) as a standardized abstraction layer for system resource access. This architecture enables Wasm to function as a high-performance, secure sandbox, extending its utility to cloud-native, edge, and IoT ecosystems.

2.2 Tree-Structured Intermediate Representation

Wasm programs are naturally hierarchical, organizing instructions within blocks and functions. `wasm-opt` leverages this hierarchy to lift the standard stack-based binary into a tree-structured intermediate representation (IR) for optimization. Figure 2 illustrates this mapping, distinguishing between the linear and human-readable text format and the hierarchical tree visualization.

The Wasm module (Left) defines a memory space, an export, and a function `example`. The IR (Right) maps this function into a hierarchical tree where ellipses denote structure, rectangles denote basic operations, and diamonds denote control flow. The mapping between the linear code and the tree is defined by the execution logic, which follows a *post-order traversal* (visiting children before parents). As indicated by the numbered tags (1–10), the execution proceeds as follows:

(1) It begins by evaluating the child nodes (Edges 1–5). The operands for the comparison are computed first: the constant `100` is pushed to the stack, the memory value is loaded, and the inner `block` is executed (performing a store and returning `0`). These values flow upward from the leaves to the parent node.
(2) Next, the parent node `i32.ge_u` consumes these results (Edge 6) to produce a boolean comparison. This outcome is immediately captured by the control flow instruction `br_if` (Edge 7), which determines whether to break execution and jump to label `$B0`.
(3) Finally, if the branch is not taken, the execution falls through to the remaining siblings in the outer block (Edges 8–10). It executes the `drop` instruction to discard the unused `100` and returns the final constant `200` to complete the function.

This traversal demonstrates how the IR transforms the linear, stack-based instruction stream into explicit parent-child connections. Because this tree visualization explicitly exposes structural

```
(module
  (memory $mem 1)
  (func $example (result i32)
    (block $B0 (result i32)
      i32.const 100
      i32.load (i32.const 0)
      (block (result i32)
        i32.const 0
        i32.const 99
        i32.store
        i32.const 0
      end)
      i32.ge_u
      br_if $B0
      drop
      i32.const 200
    end)
  )
  (export "example" (func $example))
)
```

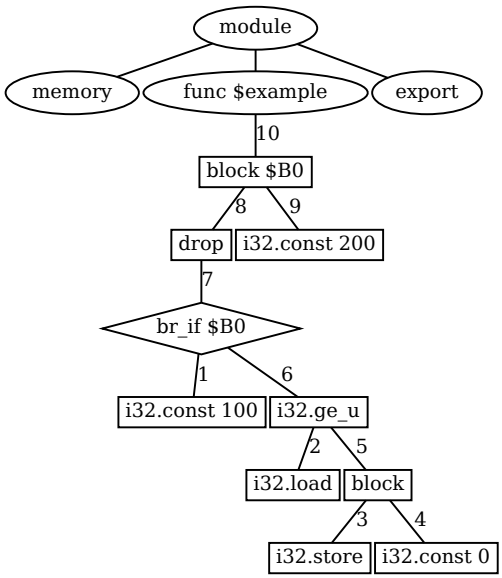


Fig. 2. A Wasm module example (left) and its corresponding simplified tree-structured IR (right).

interactions (hard to discern in linear format), we adopt it throughout the reminder of paper to illustrate the Wasm code snippets that exhibit missed optimization opportunities.

3 Motivating Example

In this section, we use a real-world missed optimization (MO) issue to illustrate our motivation and demonstrate how our methodology detects it in `wasm-opt`.

Missed Peephole Optimization. Peephole optimization is a fundamental technique that improves efficiency by replacing instruction sequences with faster equivalents using pattern matching [27, 34]. Although `wasm-opt` maintains an extensive library of such rules [61], we surprisingly observe that it often fails to handle even the most basic algebraic simplifications.

We illustrate this issue with a real-world example shown in Figure 3. The instruction `i32.ge_u` performs an unsigned comparison between the return value of `call $get_i32` and the constant `0`. Since unsigned integers are non-negative by definition ($u(x) \geq 0$), this comparison can theoretically be simplified to the constant `1`. Consequently, the enclosing `br_if $B0` functions as an unconditional branch, causing the control flow to exit `block $B0` immediately. Ideally, the optimizer should fold this condition and eliminate the unreachable code following the branch. However, `wasm-opt` fails to apply this simplification, leaving the redundant logic and dead code persisting in generated code.

Root Causes. This example illustrates the *entanglement of computation and side effects*, a fundamental limitation inherent in the Wasm tree IR. Here, the operand `call $get_i32` may introduce side effects (*e.g.*, memory writes). To safely optimize such case, the optimizer must preserve these side effects (highlighted in gray).

In a flat, linear IR (*e.g.*, LLVM's SSA), computation is separated from value definition. A function call and its subsequent comparison are represented as distinct, linear, peer-level instructions. The comparison instruction merely consumes the value produced by the call (*i.e.*, an SSA variable). Therefore, an optimizer can trivially replace the comparison with a constant while preserving the call instruction and its associated side effects.

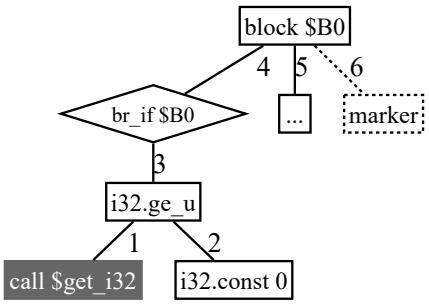


Fig. 3. A real-world missed peephole optimization in `wasm-opt`. Gray node: side-effecting operation; dashed box: a marker injected via instrumentation.

In contrast, the tree IR is designed to mirror Wasm's structured control flow and enforces a strict parent-child hierarchy. Here, a side-effecting `call` is not an independent instruction but a nested child node within the `i32.ge_u` parent. This structural coupling means that folding the parent node into a constant would implicitly prune the entire subtree, thereby erroneously erasing the necessary side effects. For safety, the optimizer must perform a complex structural mutation to restructure the subtree and preserve the side effects (a fix detailed in section 5). Faced with this complexity, `wasm-opt` conservatively retains the original tree structure, thereby causing MOs.

Limitation of Existing Approach. Existing detection methods (*e.g.,* Ditwo [26]) struggle to uncover such issues due to their reliance on *coarse-grained* runtime traces (*e.g.,* global variable writes and function calls). In this case, since the redundant code following the `br_if` is never executed, the runtime traces of the unoptimized and optimal versions are identical. Consequently, such approaches cannot systematically detect MOs within unexecuted (dead) code blocks or those that do not alter observable execution behavior.

Motivation of Our Approach. To systematically transform these subtle inefficiencies into observable behaviors, we base our detection strategy on three key observations.

DCE as an Optimization Oracle. We observe that while individual peephole MOs are difficult to detect in isolation, they frequently impede the cascading process of DCE. Since DCE is a highly effective and easily observable optimization, its failure to eliminate a code block provides a clear and reliable signal for detection when triggered by a preceding MO.

Structure-aware Instrumentation. While using DCE as an optimization oracle is an established concept, adapting it to `wasm-opt` requires overcoming the strict nesting constraints of its tree-structured IR. To this end, we develop a structure-aware instrumentation technique tailored for `wasm-opt`'s structured control flow (detailed in Section 4). Our approach precisely injects "liveness markers" (shown as the dashed box) as valid nodes within the strictly nested expression tree. By ensuring that markers adhere to the IR's structural requirements, we can determine whether an enclosing block has been successfully eliminated simply by verifying the marker's presence.

Optimization Level Monotonicity. We observe a counter-intuitive behavior in this example, where `-O2` successfully optimizes the code while `-O3` fails. Generally, higher optimization levels are expected to be at least as effective as lower ones. Based on this assumption of optimization level monotonicity, we can construct a reliable and automated oracle via differential testing across the different optimization levels within `wasm-opt`. Given that `wasm-opt` serves as the standalone standard optimizer in the Wasm ecosystem, this *cross-optimization* strategy is practical and more precise than prior cross-architecture approaches. By strictly validating monotonicity within the

tool itself, we avoid the false positives and non-actionable issues caused by cross-architecture comparisons, thereby effectively exposing actionable MOs in `wasm-opt`.

In this example, the marker's elimination at `-O2` contrasts with its persistence at `-O3`, creating a clear regression signal that pinpoints the MO. Motivated by these three observations, we implement differential testing with structure-aware instrumentation to conduct a comprehensive evaluation of `wasm-opt`. This approach allows us to systematically expose widespread subtle MOs and, critically, to reveal the fundamental limitations inherent in the tree IR.

4 Methodology

Figure 4 shows the overall workflow for detecting missed optimizations (MOs). The core insight is to leverage DCE as a lens to reveal the efficacy of preceding optimizations. Since DCE's ability to prune code depends on the success of prior passes, it serves as a sensitive indicator for MOs. Our workflow consists of three steps: ① *Structure-aware Instrumentation*: We first insert "optimization marker" into basic blocks by respecting the Wasm tree IR structure to act as "liveness indicators" ② *Cross-optimization Differential Testing*: We optimize the instrumented binary at different optimization levels (*e.g.,* comparing `-O2` vs `-O3`), and compare marker liveness. A marker eliminated in a lower level but retained in a higher one indicates an MO, leveraging the monotonicity assumption to avoid cross-architecture noise. ③ *Isolation*: To isolate the root cause within the nested structure, we eliminate "non-primary markers" whose liveness depends on parent blocks, to facilitate manual inspection and bug reporting.

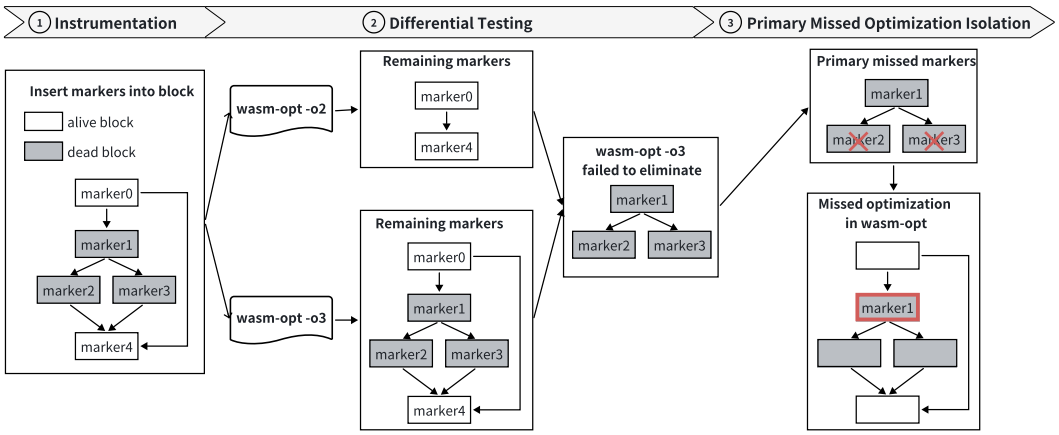


Fig. 4. Workflow of detecting missed optimizations

4.1 Structure-Aware Instrumentation

In step ①, we insert *optimization markers* into every basic block to serve as sensitive indicators. Given Wasm's structured control flow, we design a *structure-aware instrumentation strategy* for precise marker placement. To maintain the clarity of the general workflow, Figure 4 illustrates this step at a high level without depicting the specialized instrumentation logic, which we detail below.

Optimization Markers. To measure DCE effectiveness, we design an optimization marker in the form of external function calls whose definitions are unavailable to the optimizer. This prevents `wasm-opt` from analyzing or inlining them. Consequently, we can easily detect if DCE was successfully applied by finding these markers in optimized Wasm binaries. Specifically, the elimination of a marker implies that its surrounding basic block is identified as dead code and eliminated;

otherwise, its presence indicates that the block is either truly live or dead code but `wasm-opt` failed to detect. Optimization markers are not restricted to function calls; alternative implementations include compiler builtins, inline assembly, or global variable modifications.

Instrumentation for Wasm. Wasm adopts a structured control flow backed by a *tree IR*, a feature that differs fundamentally from the flat, linear instruction sequences in traditional binaries (*e.g.,* x86). This tree structure imposes strict nesting constraints: (1) control constructs (*e.g.,* `block`, `loop`, `if`) must be well-nested; and (2) branch instructions target labels defined by enclosing blocks rather than arbitrary memory addresses. This inherent hierarchy makes traditional linear instrumentation ineffective. To address this, we tailored our instrumentation approach to be *structure-aware*: guided by the Wasm specification [56], we precisely inject markers into the nested control constructs of the Tree IR, ensuring the semantic validity of the binary:

- *Unconditional Control Flow*: For unconditional instructions such as `br`, `unreachable`, and `return`, the subsequent code is dead. We therefore insert an optimization marker immediately following them. A successful DCE pass must always eliminate these markers, providing a baseline test for the optimizer's capability.
- *Conditional Branching*: For conditional instructions like `if/else` and `br_if`, markers are inserted in each potential execution path (*e.g.,* within both the "true" and "false" branches of an `if/else` statement). An MO is revealed if the optimizer fails to prove that a condition is constant, consequently leaving a marker in what should have been a dead branch.
- *Multi-Way Branching*: The `br_table` instruction, which functions as a multi-way branch similar to a `switch` statement. We instrument each of its case branches with a marker, allowing us to verify if the optimizer successfully eliminates any of the dead branches.
- *Excluded Instructions*: Instructions that do not create control flow branching, such as `nop`, `call`, and `call_indirect`, are not instrumented as they do not, by themselves, create opportunities for DCE in the manner we aim to detect.

Figure 5 illustrates the instrumentation for a `br_if` instruction. In the original control flow (left), the fall-through path—highlighted in gray and comprising the instruction *instr**—is executed only if the `br_if` condition (*cond*) evaluates to false. Our instrumentation (right) appends an optimization marker to this path, immediately following *instr**. By inspecting this marker, we can confirm if DCE is correctly applied: if the optimizer determines *cond* is always true, the fall-through path becomes dead code, and a successful DCE pass must eliminate the marker. This instrumentation principle is applied across all relevant control flow instructions, with marker placement adapted to their specific semantics.

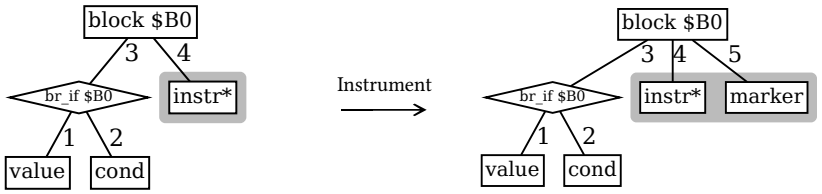


Fig. 5. Instrumentation of a `br_if` conditional branch. A marker is placed in the fall-through path.

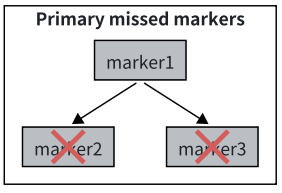
4.2 Cross-Optimization Differential Testing

In step ②, we optimize the instrumented binary B_{inst} at a lower optimization level L and a higher level H to generate the respective optimized binaries $B_L = Opt_L(B_{inst})$ and $B_H = Opt_H(B_{inst})$. We define a marker's status as *alive* if the basic block it marked is remained after optimization, and *dead* if it is removed. Let $\mathcal{M}(B)$ denote the set of markers alive in binary B. Our detection logic is

based on the *monotonicity of optimization levels* —the assumption that higher optimization levels should be at least as effective as lower ones in eliminating dead code. Thus, if a marker is dead at a lower optimization level (L) but alive at a higher one (H), this indicates the higher level missed an optimization opportunity. By establishing this *internal ground truth*, we avoid the noise and false positives inherent in cross-architecture comparisons (*e.g.,* memory model discrepancies between Wasm and x86). Under this assumption, any marker removed by Opt_L should logically also be eliminated by Opt_H, leading to the expected inclusion property $\mathcal{M}(B_H) \subseteq \mathcal{M}(B_L)$. A violation of this property flags an MO issue in Opt_H. To quantify such instances, we define the missed set as $\mathcal{M}_{missed} = \mathcal{M}(B_H) \setminus \mathcal{M}(B_L)$. We conclude that Opt_H fails to capture available optimization opportunities if and only if $\mathcal{M}_{missed} \neq \emptyset$.

4.3 Primary Missed Optimizations Isolation

Not every marker in $\mathcal{M}_{missed}$ represents a unique MO. Due to the hierarchical nature of the tree IR, the elimination of a parent block inherently removes all its nested child blocks. Consequently, if *Opt_L* removes a parent block while *Opt_H* fails to do so, all markers within that entire subtree will appear in $\mathcal{M}_{missed}$. In such cases, the child markers are merely "shadows" of the parent block's survival and do not flag independent optimization failures. As illustrated in Figure 6, suppose $\mathcal{M}_{missed}$ contains `marker1`, `marker2`, and `marker3` after step ②. As shown in the left panel, we identify only `marker1` as the primary MO because the liveness of the others depends on it. In the corresponding example (right panel), the inner `if` statement (containing `marker2` and `marker3`) remains alive solely because its parent `if` statement (containing `marker1`) was not eliminated. Consequently, our process identifies the outer block as the root cause, classifying `marker1` as the primary failure while disregarding the others as secondary effects. Formally, a marker $M \in \mathcal{M}_{missed}$ is considered primary if and only if none of its immediate parent control-flow blocks contain any other markers belonging to $\mathcal{M}_{missed}$. If a parent block contains a missed marker, it implies that the survival of M is likely a dependency of the parent's non-elimination. By applying this filtering, we ensure that each remaining marker in $\mathcal{M}_{missed}$ corresponds to an independent root cause, thereby streamlining the subsequent manual inspection and bug reporting process.




```
1 if False:
2     call marker1  # primary missed marker
3     if cond1:
4         call marker2
5     else if cond2:
6         call marker3
```

Fig. 6. An example of primary missed optimization isolation.

5 Evaluation

We evaluate our approach by answering the following questions:
RQ1 How effective is our approach in terms of finding MOs in `wasm-opt`?
RQ2 What is the performance impact of the fixed MOs on real-world Wasm applications?
RQ3 What are the characteristics and root causes of the detected MOs?
RQ4 Does our approach outperform state-of-the-art MO detection approaches?

5.1 Experimental Setup

The whole testing process comprises four parts: test generation, instrumentation, differential testing, and reduction. **First**, for test generation, we used `Csmith` (`v2.3.0`) [65] to generate C programs as

input for testing. These programs are self-contained, requiring no external inputs, and typically contain large amounts of dead code, making them suitable for evaluating the effectiveness of our approach. Furthermore, we used `emcc` (`v4.0.5`) to compile C programs into unoptimized Wasm binaries (using `-O0`) as test input. **Second**, for instrumentation, we applied our structure-aware instrumentation strategy by inserting markers, implemented as external function calls (*e.g.,* `call $marker_ID`), into nested basic blocks while strictly adhering to the Wasm control-flow structure. **Third**, for differential testing, we applied `wasm-opt` (commit `0997f9b`) to optimize the binaries across optimization levels ranging from `-O1` to `-O4`. **Fourth**, for test case reduction, we used `wasm-reduce` (`v122`) to minimize their code size and `WAPT` (`v1.0.37`) to convert the binaries into textual format. The tool is implemented in approximately 1,500 lines of Python code.

Our experiment was conducted on a machine running Ubuntu 24.04.3 LTS, equipped with an Intel(R) Core(TM) Ultra 5 225F(10 cores) and 7.6GB RAM. We ran the experiment for three months, during which we continuously generated test cases, tested, and reported issues to the developers.

5.2 Triage Effort

Process. We systematically triaged the raw discrepancies from our differential testing to produce unique and developer-ready bug reports. *First*, we isolated (detailed in Section 4.3) the primary markers responsible for the discrepancy, and afterwards minimized test cases using `wasm-reduce`. *Second*, we manually verified that the target marker incorrectly survived in the higher-level optimized program while being eliminated at the lower level. These verified cases were then reported to the `wasm-opt` developers, with potential fix patches when possible. *Third*, we discussed the issues with `wasm-opt` developers, merging reports confirmed to be the same issue or resolved by the same fix commit into single issues to prevent duplicate counting. *Finally*, by analyzing developer feedback and issue fixes, we categorized the confirmed issues using an open-card sorting approach [37].
Results. Following this triage process, an initial set of 28 raw marker-level discrepancies was successfully deduplicated into 24 unique issues, each supported by developer confirmation or a fix (as listed in Table 1). The four removed cases were duplicate issues, and no false positives were found during the triage process. The final classification yielded five root-cause classes, which are analyzed in RQ3. in the `wasm-opt` implementation. Notably, some issues (*e.g.,* issues 1–7) share the same root cause class but are treated as unique. This is because they currently lack a universal fix, and instead require separate, localized patches in the `wasm-opt` implementation.

5.3 RQ1: Effectiveness in Detecting Missed Optimizations

Bug Count. As shown in Table 1, we have found 24 MOs in `wasm-opt`, and all of them were previously unknown and confirmed by the developers. Up to the date of this paper's submission, 20 MOs have been fixed, while the remaining 4 were left unfixed due to the trade-offs between fix complexity and performance gains. This 100% confirmation rate indicates that our approach produced no false positives caused by architectural differences or violations of optimization monotonicity, and the 20 fixes further demonstrate the actionability of the detected MOs. Aside from MO, we also exposed one miscompilation bug (issue #15). This bug revealed incorrect `0/0` division handling in `getMaxBits`, a common utility function for computing the maximum amount of bits used in an integer expression, which further causes the MO issue. Developers acknowledged this as a *"good finding"*. These results are encouraging, and to our knowledge, no prior work on `wasm-opt` has detected as many actionable issues, nor achieved a comparable number of confirmed fixes.

Table 1. Overview of confirmed missed optimizations (unique). Columns: **ID** = issue identifier; **Class** = bug category; **Affected Pass** = optimization pass where the issue occurs; **Opt. Level** = optimization level that triggers it; **Fixed?** = whether the issue has been fixed; **Description** = underlying reason. Symbols: ^{W} marks Wasm-specific passes (others are general-purpose). † indicates issues from interactions among multiple passes. ✓ = fixed; ✓* = silently fixed.

ID	Class	Affected Pass	Opti.	Fixed?	Description
1	I	OptimizeInstructions	-O3	✓	overly conservative treatment of $u(x) < 0 \Rightarrow$ `i32(0)`
2	I	OptimizeInstructions	-O3	✓	overly conservative treatment of $u(x) \geq 0 \Rightarrow$ `i32(1)`
3	I	OptimizeInstructions	-O3	✓	overly conservative treatment of $u(x) \leq -1 \Rightarrow$ `i32(1)`
4	I	OptimizeInstructions	-O3	✓	overly conservative treatment of $u(x) > -1 \Rightarrow$ `i32(0)`
5	I	OptimizeInstructions	-O3	✓	overly conservative treatment of $x \mid -1 \Rightarrow -1$
6	I	OptimizeInstructions	-O3	✓	overly conservative treatment of $bool(x) \mid -1 \Rightarrow 1$
7	I	OptimizeInstructions	-O3	✓	overly conservative treatment of $bool\ \&\ -1 \Rightarrow 0$
8	II	MergeBlocks^{W}	-O3	✓	failure to hoist constant value from binary operation
9	II	MergeBlocks^{W}	-O3	✓*	failure to hoist constant value from unary operation
10	II	MergeBlocks^{W}	-O3	✓	failure to hoist constant value from binary operation
11	II	MergeBlocks^{W}	-O3	✓	`getMaxBits` fails to handle `block` type
12	III	LocalCSE^{W}	-O3	✓	aggressive elimination of local registers via `local.tee`
13	III	CoalesceLocals^{W}	-O2		aggressive reuse local registers via `local.tee`
14	III	LocalCSE^{W}	-O3		inadequate tracking of the bits of local variables
15	III	LocalCSE^{W}	-O3	✓	incorrect handling of `getMaxBits` for 0 / 0
16	IV	RemoveUnusedBrs^{W}	-O3	✓	failure to remove unused `br_if`'s value in `block` tails
17	IV	RemoveUnusedBrs^{W}	-O3	✓	failure to optimize `break` with constant fallthrough values
18	IV	RemoveUnusedBrs^{W}	-O3	✓	`restructureIf` causing counterproductive effect
19	V	OptimizeInstructions	-O3	✓	missing rule for binary expressions emitting zero bits
20	V	Precompute	-O3	✓	inadequate math computation in precompute
21	V	Precompute-propagate	-O3	✓*	limited local tracking in precompute-propagate
22	V	Flatten^{W}	-O4		Flatten pass polluting constant propagation pass
23	V	Multi-pass†	-O3	✓*	pass ordering issue (`Vacuum` and `MergeBlocks`)
24	V	Multi-pass†	-O3		pass ordering issue (`Vacuum` and `ConstantFieldPropagation`)

Distribution. These detected MOs are across optimization levels: most cases (22) were triggered under `-O3`, with one case found in `-O2` and one in `-O4` [1]. This distribution reveals that *higher optimization levels are more prone to regression*. This occurs because `-O3` applies more sophisticated and aggressive tree-IR optimizations. As demonstrated by our later root cause analysis, these complex optimizations can inadvertently perturb the code structure, thereby obstructing basic optimizations (such as DCE) that would otherwise succeed at lower levels. Consequently, this high concentration of MOs in `-O3` validates our cross-optimization testing strategy: using lower levels as a baseline effectively exposes the fragility of complex optimizations.

Affected Passes. The detected MOs span 8 distinct passes, covering both general-purpose and Wasm-specific optimizations. Specifically, 12 cases involve general-purpose passes, which are mostly `OptimizeInstructions` (8), along with `Precompute`-related passes (2) and multi-pass interactions (2). The other 12 cases relate to Wasm-specific passes, including `MergeBlocks` (4), `LocalCSE` (3), `RemoveUnusedBrs` (3), `CoalesceLocals` (1), and `Flatten` (1). This demonstrates that although our approach only observes DCE outcomes, it can exposes MOs of passes corresponding to DCE.

5.4 RQ2: Performance Impact of Fixed Missed Optimizations

Setup. To measure the real-world impact of the fixed MOs, we evaluated them on the Emscripten benchmark suite [9]. We chose this suite because it is used for `wasm-opt`'s official benchmarking

[1] `wasm-opt` provides `-O4` as a more aggressive optimization level. However, it does not support newer Wasm features, and developers suggested that it may eventually be deprecated. Due to these limitations, we stopped testing `-O4`.

Table 2. Summary of the performance impact of the fixed MOs on 21 benchmarks from the Emscripten benchmark suite. Positive values indicate speedups and code size reductions, respectively, single "–" symbol indicates that no regression was observed.

Metric	Execution Time	Code Size
Improved Benchmarks	15	15
Regressed Benchmarks	6	0
Unchanged Benchmarks	0	6
Maximum Improvement	+2.96%	+3.90%
Maximum Regression	-8.62%	–
Average Change	-0.52%	+1.30%

and covers a diverse set of real-world applications, including physics engines, compression libraries, and virtual machines. In our setup, we compared `wasm-opt` (`v127` [54]) with a baseline created by reverting the fix commits corresponding to our reported MOs. Specifically, we reverted the 17 explicit fix commits and excluded the three silently fixed cases because their fixes could not be isolated as separate commits. To ensure that the observed performance gains were solely due to `wasm-opt` and not influenced by frontend optimizations, we compiled each benchmark with `emcc -O0` and then optimized the resulting Wasm binaries with `wasm-opt -O3`. For execution time measurement, we ran each benchmark under Node.js five times and used the average.

Results. Table 2 shows the overall results: across the 21 evaluated benchmarks, 15 run faster and 15 produce smaller binaries. For code size, the fixes reduced binary size by an average of **1.30%**, with a maximum reduction of **3.90%**. Notably, none of the benchmarks exhibited any code size regression. The runtime impact was more varied: while 15 benchmarks showed speedups (peaking at 2.96%), the remaining six benchmarks incurred slowdowns (up to 8.62%), thus leading to an average execution time slowdown of 0.52%. Overall, these results highlight a size and speed trade-off: while the MO fixes provide a reliable reduction in code size, they can occasionally introduce regressions in execution time. These results were acknowledged by the developers, who commented that *"the code size reduction seems pretty good"*.

5.5 RQ3: Characteristics and Root Causes of Missed Optimizations

To understand these MOs and `wasm-opt`'s limitations, we classified the 24 confirmed issues into five major classes: "*Overly Conservative Peephole Optimization*", "*Overly Conservative Block Merging*", "*Local Variable Over-Reuse*", "*Insufficient Unused Branches Elimination*", and "*Others*". Moreover, we analyzed their root causes by examining the `wasm-opt`'s source code and discussing with developers. The remainder of this subsection details the characteristics and root causes of these five issue classes, illustrating each with representative examples and their corresponding fixes.

Overly Conservative Peephole Optimization (Class I). This class stems from `wasm-opt`'s conservative handling of side effects within the tree IR, resulting in missed peephole optimizations.

A representative example (issue #2) is shown in Figure 7 (left panel), which corresponds to the case analyzed in Section 3. Here, `wasm-opt` conservatively refuses to apply the peephole optimization ($u(x) \geq 0 \implies 1$) and retains the original structure to ensure safety. As discussed previously, fixing this issue requires the optimizer to perform a complex structural mutation rather than simple folding. The corresponding fix (right panel) replaces the comparison with the constant `1` but explicitly retains the side-effecting `call` instruction, using a `drop` to discard the unused return value. This localized patch effectively simplifies the computation while strictly preserving the necessary side effects.

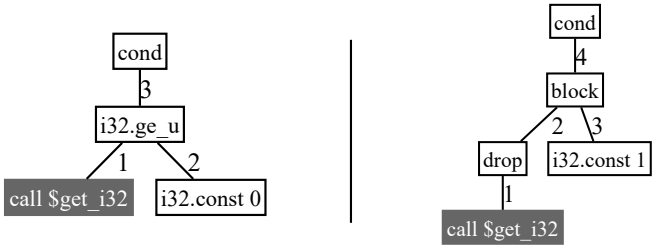


Fig. 7. An example of overly conservative rule applications (Class I, left) and the corresponding fix (right).

Class I constitutes a significant portion (8 of 24) of the detected issues, involving eight distinct peephole optimization strategies. These findings further demonstrate the root cause analysis in Section 3: while the tree-structured IR is designed to closely mirror the Wasm binary format for fast and simple conversion, it inherently entangles computation with side effects within its nested structure. This structural entanglement forces the optimizer to adopt the overly conservative behaviors. Although localized patches have been applied to address these specific peephole rules, the underlying architectural limitation lacks a comprehensive solution. We revisit this challenge and provide a potential remedy in Section 6.

Overly Conservative Block Merging (Class II). The `MergeBlocks` pass [59] in `wasm-opt` is designed to simplify program structure and reduce nesting depth by eliminating redundant blocks and flattening nested block structures. Yet, similar to the conservative strategy of Class I, our findings (Class II in Table 1) reveal that the optimizer fails to simplify blocks containing potential side effects, leading to missing `MergeBlocks` optimizations.

Figure 8 illustrates a concrete instance of Class II (issue #8). In the left panel, the `i32.ge_u` operation compares a loaded value (`i32.load`) against the result of a `block`, which performs a memory write (`i32.store`) before returning the constant `0`. Ideally, `MergeBlocks` should hoist the `store` operation (highlighted in gray) and flatten the block, thereby exposing the constant `0` to enable further simplification. However, due to the *strict nesting constraints of the tree IR*, the `i32.ge_u` operator strictly requires two child operands. To flatten the block, the internal `i32.store` would have to be hoisted before the entire comparison. Crucially, this hoisting reorders the `store` ahead of the first operand's `i32.load`, risking side effects such as overwriting or reading incorrect data. Consequently, `wasm-opt` must conservatively skip this transformation, resulting in a missed `MergeBlocks` optimization. More importantly, this failure further prevents the subsequent peephole optimization ($u(x) \geq 0 \Rightarrow 1$) discussed in Class I. The corresponding fix (right panel) allows `wasm-opt` to focus on the block's return value independently of its internal side effects. By separating the side effects from the return value, the optimizer can safely replace the comparison with the constant `1` without altering the execution order of the `load` and `store`.

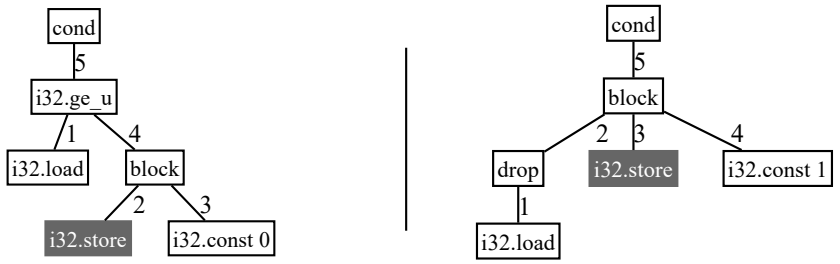


Fig. 8. An example of limited merging rules (Class II, left) and the corresponding fix (right).

In total, we identified four issues related to missed `MergeBlocks` optimizations. Similar to Class I, the root cause stems from the tree IR entanglement. As identified in our design analysis, the hierarchical IR entangles computation with side effects. The optimizer is forced into *overly conservative decisions* because simplifying the structure requires complex disentanglement of these side effects from the expression tree. Furthermore, these conservative strategies create negative interactions between various optimization passes, where the failure of one pass (*e.g.,* `MergeBlocks`) directly hinders subsequent passes. This demonstrates how an architectural constraint lead to numerous MOs, ultimately undermining the efficiency of the entire optimization pipeline.

Local Variable Over-Reuse (Class III). This class relates to the `LocalCSE` (Local Common Subexpression Elimination) pass [58], a fundamental technique utilized by `wasm-opt` to eliminate redundant computations. Specifically, this pass identifies identical expressions within a basic block; instead of re-evaluating these common expressions, it computes the value once and stores the result for subsequent reuse, thereby improving execution performance and reducing code size. Unexpectedly, the detected issues (Class III in Table 1) indicate that this optimization is not always beneficial. Rather than facilitating optimization, `wasm-opt` may apply overly aggressive variable reuse, inadvertently obstructing more effective optimizations.

For Class III, we detail a typical scenario (issue #12) as shown in Figure 9. In the original program (left), an `i32.eq` operation compares two identical expressions (*instr**). The `LocalCSE` pass detects this redundancy and caches the first result via a `local.tee` instruction, replacing the second instance with a `local.get`. However, this transformation inadvertently prevents the peephole optimizer from identifying the underlying equivalence. Specifically, the peephole optimization (see Class I) relies on syntactic pattern matching (*e.g., i32.eq*$(x, x) \Rightarrow 1$); however, it fails to equate the `local.tee` node with the `local.get` node. This MO is caught through our cross-optimization differential testing: at `-O2`, where `LocalCSE` is not applied, the `wasm-opt` successfully folds the `cond` to the constant `1` and eliminates the marker inserted in the branch (omitted in the figure); at `-O3`, however, the marker persists, violating monotonicity.

This illustrates *the fragility of peephole rules in the tree IR*: the `LocalCSE` pass creates a structural divergence, effectively breaking the canonical tree shape expected by the peephole optimization. Consequently, the simplification to the constant `1` is missed. The corresponding fix (right) addresses this by constraining `LocalCSE`: it now evaluates the cost of *instr** and only applies recomputation for low-cost operations. By preserving the original structure, the fix enables the peephole optimizer to successfully replace the comparison with `1` and insert a `drop` instruction to safely maintain the possible side effects.

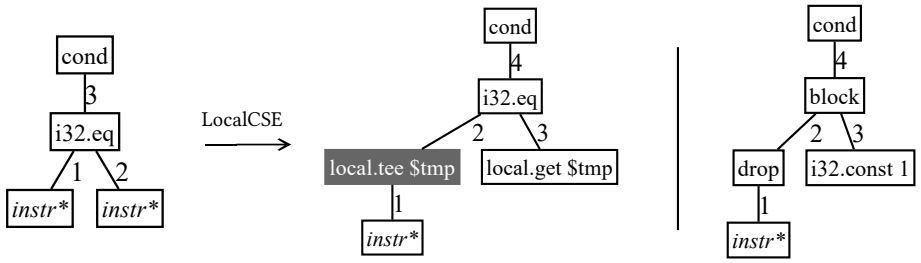


Fig. 9. An example of local variable over-reuse (Class III, left) and the corresponding fixed (right).

We further uncovered four missed `LocalCSE` optimizations. These issues are rooted in the lack of global dataflow information, forcing passes like `LocalCSE` to rely on localized heuristics. By prioritizing short-sighted simplifications, these passes often create suboptimal code structures that

inadvertently block more effective optimizations later in the pipeline. Additionally, issues #13 and #14 involve complex scenarios and remain unfixed due to practical trade-offs; developers lowered their priority because the analysis is too expensive for the limited performance benefit.

Insufficient Unused Branches Elimination (Class IV). This class involves the `RemoveUnusedBrs` pass [60], which is designed to streamline control flow by eliminating unreachable or redundant branches. However, we observed that this pass fails to recognize the semantic equivalence of two structurally different branches, leading to missed `RemoveUnusedBrs` optimizations.

A representative instance of Class IV (issue #16) is shown in Figure 10. In the original code (left panel), the program first pushes a value, v_1, onto the stack. It then encounters a conditional branch: if true, it exits the block immediately, leaving v_1 on the stack; if false, it drops v_1 and pushes a new value, v_2. Ideally, this code is optimizable when v_1 equals v_2, as both paths produce an identical final stack state. However, `wasm-opt` only sees the different instructions on each path, failing to recognize this equivalence and missing the opportunity to eliminate the redundant branch. The corresponding fix (right panel) simplifies this by removing the branch entirely; it executes the condition within a `drop` instruction to preserve side effects and directly returns the value v_2.

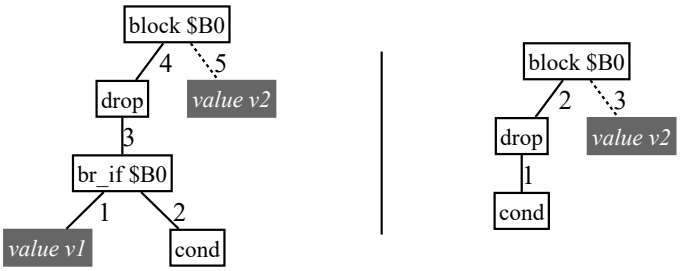


Fig. 10. An example of inadequate control flow optimizations (Class IV, left) and the corresponding fix (right).

We identified three missed `RemoveUnusedBrs` optimizations in total. These findings expose the *fundamental design limitation of insufficient semantic capture*. To determine whether distinct branches are equivalent, the optimizer relies on simple syntactic inspections, merely identifying the structural discrepancies between instruction sequences. Consequently, it fails to capture deeper semantic information, specifically the stack state changes and the final stack outcome.

Others (Class V). This category accounts for the remaining 6 MOs. Unlike the previous classes, which stem from Wasm design limitations, these issues are attributed to optimizer design oversights including rule coverage (3), feature maturity (1), and pass scheduling (2).

First, incomplete rule coverage in the `-O3` optimization level leads to missed opportunities in the `Peephole` and `Precompute` passes. The `Peephole` pass (detailed in Class I) focuses on pattern matching, and the `Precompute` pass evaluates constant expressions at compile time, directly substituting them with constants. However, we identified the absence of optimization rules in both: `Peephole` failed to simplify logical operations like $i32(x) \wedge i32(0) \Rightarrow i32(0)$ (issue #19), while `Precompute` lacked rules for operators such as $i32.eqz(0) \Rightarrow i32(1)$ (issue #20) and a complex comparison related to `i32.ne` (issue #21). Although straightforward rules were added to fix #19 and #20, developers rejected adding narrow rules for the complex comparison in #21, which was ultimately silently fixed.

Second, aggressive experimental features in the `-O4` level can degrade code quality. Issue #22 revealed that the `Flatten` pass, which aggressively alters control flow structures, inadvertently disrupted the program structure required for constant propagation. Upon receiving our report, the developer acknowledged that `-O4` is rarely used and lacks support for mature features, leading us to exclude it from further testing.

Third, inappropriate pass scheduling may result in suboptimal optimizations. We observed two MOs arising from the ordering between the `Vacuum` pass (dead code elimination), the `MergeBlocks` pass (block simplification) and the `ConstantFieldPropagation` pass (constant propagation). In issue #23, the `wasm-opt -O3` pipeline inferred a if-condition is always false, but was scheduled after the `Vacuum` pass, resulting in the dead code remaining uneliminated. This can be resolved by inserting an additional `Vacuum` pass. Similarly, in issue #24, dead code nested within blocks failed to be eliminated. While reordering the `Vacuum` and `MergeBlocks` passes offers a solution, developers considered the resulting complexity and overhead unacceptable. Instead, they advocated for a future pass manager capable of identifying optimization needs and picking passes adaptively.

5.6 RQ4: Comparison with State-of-the-Art Detection Approaches

DiTwo [26] is the state-of-the-art tool for detecting MOs in Wasm optimizers. Unlike our approach, it relies on cross-architecture differential testing, identifying MOs by comparing runtime traces (*e.g.*, global writes and function calls) between Wasm and native x86 execution.

To demonstrate the fine-grained detection capabilities of our approach, we evaluated DiTwo against the 24 confirmed MOs we identified. Notably, DiTwo *failed to detect any of these issues.* This empirical result validates the analysis in Section 3: DiTwo's reliance on coarse-grained runtime traces renders it blind to MOs hidden within unexecuted code. Moreover, coarse-grained traces offer little insight into the structural root causes of these MOs, which primarily stem from the inherent constraints of the Tree IR.

Furthermore, we observe that most MOs reported by DiTwo were deemed unfixable. The inherent architectural discrepancies in cross-architecture comparisons (x86 vs. Wasm) frequently expose gaps that are technically infeasible to address within `wasm-opt`, limiting the actionability of their findings. As illustrated in Figure 11, the x86 backend successfully eliminates the dead store to `b`, whereas `wasm-opt` retains it. This stems from differing memory models: while LLVM treats globals as isolated symbols, Wasm maps them to linear memory offsets, forcing `wasm-opt` to conservatively assume potential aliasing. Consequently, this represents an unactionable optimization gap imposed by architectural constraints, rather than a fixable defect in the optimizer [6].

By shifting to a cross-optimization strategy exclusively within `wasm-opt`, we ensure that detected issues are fixable optimization regressions rather than architectural constraints. Furthermore, our structure-aware instrumentation enables the detection of fine-grained MOs nested within complex tree structures, including those in unreachable code that remain invisible to existing methods. Consequently, our findings are highly actionable, achieving high confirmation and fix rates.

6 Lessons Learned

Lesson 1: DCE is not a universal MO testing oracle, but an effective, practical, and extensible one. Indeed, there is no silver bullet for MO detection, and our approach is no exception. However, DCE remains an effective, easy-to-implement, and low-risk oracle in practice. **First**, from a conceptual standpoint, DCE acts as an optimization "sink" [45], because its success inherently relies on the prior execution of upstream passes (e.g., constant propagation), observing DCE evaluates the broader pipeline. This point is strongly supported by our empirical results: relying solely on DCE observations, our approach detected 24 MOs in `wasm-opt` across 8 distinct passes. **Second**, the observation mechanism is easy to implement, as it only requires *statically* checking whether inserted markers survive, rather than relying on complex semantic modeling [7], marker alignment [25], or runtime tracing [26]. **Third**, DCE is a low-risk oracle because it minimizes false positives in bug reporting. Unlike optimizations such as loop unrolling, where developers might justify a "missed" opportunity as a deliberate trade-off to avoid code bloat, eliminating dead code is

```c
#include <stdint.h>
int32_t a;
static int32_t b;
int main() {
    a = 1;
    b = 1;
    return a;
}
```

(a) C source code

```
main:
    movl $1, a@GOTOFF(%eax) ;; "a = 1"
    ;; "b = 1" is eliminated
    movl $1, %eax ;; "return a"
    retl
```

(b) x86 assembly code (clang -O3)

```
;; linear memory: a contiguous byte-addressable array
(memory 258 258)

(func $main
    ;; store 1 at offset 65536 in linear memory
    i32.const 65536
    i32.const 1
    i32.store

    ;; store 1 at offset 65540 in linear memory
    i32.const 65540
    i32.const 1
    i32.store

    ;; load from offset 65536 and return
    i32.const 65536
    i32.load)
```

(c) Wasm code (wasm-opt -O3)

Fig. 11. An example illustrating a false positive in cross-architecture comparison.

universally beneficial [29, 30]. Therefore, unremoved dead code is an undeniable optimization flaw, ensuring that reported bugs are highly likely to be accepted by developers.

Furthermore, several extensions can partially mitigate the limitations of DCE. First, test programs can be seeded with optimization-dependent dead code, such as conditional markers whose guards become statically false only after earlier optimizations. Second, DCE can be combined with lightweight complementary oracles, including instruction-count oracles for simplification-related MOs and variable-usage or function-call-chain oracles for dynamic compilers.

Lesson 2: Cross-optimization provides a stable oracle for ecosystems with a single optimizer implementation. Existing MO detection approaches primarily rely on *cross-compiler*, *cross-version*, or *cross-optimization* comparisons to construct ground truths [5]. Among these, cross-compiler testing is the main choice and has proven effective in C/C++ compiler (LLVM vs. GCC) where comparable peer compilers exist [45]. Similarly, Liu et al. use a cross-architectural comparison (*i.e.,* Wasm vs. Clang) to detect MOs in `wasm-opt` [26]. While it can expose performance gaps between Wasm and C/C++ compilers, in practice, it induces false positives due to architectural disparities.

For ecosystems with a single dominant optimizer like `wasm-opt`, *cross-optimization* provides a much more stable ground truth. Instead of comparing against a different compiler or architecture, it compares optimization levels within the same tool. This approach is based on *optimization monotonicity*: the expectation that higher optimization levels should perform no worse than lower ones. One concern is that violations of optimization monotonicity may introduce false positives. Nevertheless, such violations are expected to be infrequent, as higher optimization levels are designed to enable more aggressive and effective optimizations. Moreover, a small number of monotonicity violations is acceptable for MO detection. Our experimental results support this assumption, with a 0% false positive (FP) rate caused by monotonicity violations. Indeed, this 0% FP rate is largely attributed to the combination of the DCE oracle and *cross-optimization*. Since DCE is a fundamental and widely applicable optimization, it rarely violates optimization monotonicity. This is also one reason why, as discussed in Lesson 1, DCE serves as an effective and low-risk oracle for MO detection.

Lesson 3: While beneficial for fast decoding and translating, the tree IR's strict hierarchy inherently constrains aggressive optimization. Our findings show that many MOs in `wasm-opt` stem directly from its architectural design. The tree IR is designed to mirror its binary format, which benefits fast decoding and translation. However, as shown in Class I and Class II findings, this design inherently entangles computation with side effects. In a linear IR (*e.g.,* LLVM's SSA), computation is separated from value definition, allowing an optimizer to replace a calculation without affecting nearby memory operations. In contrast, the tree IR enforces a strict parent-child hierarchy. Simplifying a parent node often requires complex structural mutations to preserve the side effects nested within its children. For safety, `wasm-opt` frequently makes conservative choices, retaining the original structure and missing optimization opportunities. Furthermore, this design forces passes to rely on local heuristics rather than deeper global data-flow semantics. While previous studies have explored the nature of tree-based IRs in compiler frontends and hybrid pipelines [16, 32], the implications of relying solely on a tree IR for optimization (especially in Wasm's optimizer) have remained unexplored. Future `wasm-opt` must carefully consider this trade-off, potentially improving its IR architectures that separate computation from control flow.

7 Discussion

Analysis of Historical MOs. We analyzed 11 years (2015.11-2026.5) of historical Github issues for `wasm-opt`. Two authors spent ten days on this review. We collected 210 optimization-related issues (miscompilations, crash/hangs or missed optimizations). After manual filtering, 36 issues remained as confirmed MOs. Our results showed that 15 of the 36 confirmed MOs (41.7%) match our findings: overly conservative peephole optimization (2), overly conservative block merging (1), insufficient branch elimination (2), and other like incomplete rule coverage (10).

The remaining 21 unmatched MOs (58.3%) mostly involve advanced Wasm features like Garbage Collection (12) or inlining heuristics (2). These bugs fall outside our categories for two reasons: (a) some MOs are just not related to DCE, (b) our input generator (*i.e.,* Csmith) only tests Minimum Viable Product (MVP) Wasm and cannot reach advanced feature interactions. The final 7 unmatched bugs are either unnecessary or beyond `wasm-opt`'s intended capability.

In summary, matched historical bugs confirm that our findings align with practical Wasm issues. Beyond that, our work reveals MO categories (*e.g.,* Class II and III) that are rarely or never reported in historical issues. This shows our work fills a clear gap in existing research and bug trackers.

Threats to Validity. One primary internal threat comes from our DCE-based testing approach. *First*, it relies on the presence of dead code in test programs; otherwise, marker survival cannot expose MOs. *Second*, it cannot directly detect MOs caused by DCE-unrelated optimizations, such as register allocation. *Third*, it falls short of effectively testing dynamic compilers because dead code is never executed, and the inserted markers thus remains invisible to the optimization pipeline. Another internal threat lies in the implementation: our choice of marker type (external function calls) might affect the bug-finding capability. Although other markers exist (e.g., compiler builtins [46] or variable writing traces [26]), we leave the exploration of different marker types to future work.

One external threat arises from the selected input generator, which restricts the coverage of Wasm language features. As our historical bugs analysis indicates, certain MOs are triggered by specific Wasm extensions (*e.g.,* garbage collection). The Csmith-based input generation cannot test GC or interactions among advanced features. Further work will incorporate more diverse input generator like `wasm-smith` [55] to expand this testing scope.

A Potential Future Direction for `wasm-opt` and Other Optimizers. Our findings indicate that `wasm-opt` lacks sufficient support for complex data-flow and memory-dependency analyses, which are standard techniques in mature compilers like LLVM. Reimplementing decades of optimization research for `wasm-opt` is impractical. Thus, we propose a high-level, hybrid strategy: augmenting

`wasm-opt` by selectively offloading optimization tasks to LLVM. Specifically, this involves translating a compatible subset of the IR into LLVM IR, leveraging LLVM's mature pipeline, and converting the optimized result back. The potential impact could be substantial. This strategy enables `wasm-opt` to directly inherit LLVM's extensive suite of optimizations, thereby effectively closing the performance gap. Furthermore, it would extend these benefits to languages compiling to Wasm without an LLVM-based toolchain (*e.g.,* Go, AssemblyScript). However, challenges remain as LLVM does not support all Wasm features (*e.g.,* Garbage Collection). A practical solution requires a mechanism to isolate LLVM-compatible regions, translate them for optimization, and merge them back. In conclusion, leveraging established compiler infrastructures presents a far more efficient engineering approach than reimplementing optimizations for `wasm-opt` and other optimizer.

8 Related Work

Detecting Wasm Optimizer Missed Optimization. The most closely related work by Liu *et al.* [26], which detects MOs in `wasm-opt` by comparing coarse-grained runtime traces between Wasm and native x86. In contrast, we shift perspective from *cross-architecture* to *cross-optimization*. By injecting markers into Wasm code and leveraging the internal monotonicity of optimization levels, we can identify fine-grained MOs within basic blocks while avoiding architectural noise.

Finding Missed Compiler Optimization. Unlike crash or miscompilation issues, missed optimizations are silent performance defects. Early approaches relied on differential testing of static features: Barany *et al.* [1] compared instruction counts and stack usage across compilers, while Hashimoto *et al.* [17] focus on arithmetic optimization via equivalent AST generation. Liu *et al.* [26] employed differential testing to Wasm optimizer by comparing execution trace against native binaries. Instead of comparing different compilers, recent works focus on internal inefficiency indicators. Tan *et al.* [41] utilized dynamic analysis to pinpoint redundant memory operations. A foundational technique by Theodoridis *et al.* [45] uses Dead Code Elimination (DCE) as an oracle; if a compiler fails to remove injected "dead" markers, a missed optimization is flagged. Our work adopts this oracle but *adapts the instrumentation strategy* to accommodate Wasm's unique structured control flow and binary format, enabling *fine-grained detection* within `wasm-opt`. Building also on DCE oracle, Zhang *et al.* [66] proposed Synchronized Behavior Checking (SBC), which cross-validates correlated optimizations (*e.g.,* loop unswitching and DCE) within a single compiler to detect inconsistencies. Recent studies have also explored how input manipulation reveals missed optimization opportunities. Gao *et al.* [13] found that "de-optimizing" source code can surprisingly lead to better machine code, while Theodoridis [46] and Weber *et al.* [62] demonstrated that providing more information (*e.g.,* value ranges) or more precise alias analysis can paradoxically degrade optimization results. Finally, Large Language Models (LLMs) have been adopted for test generation: Italiano *et al.* [19] leveraged LLMs to mutate code for finding code-size regressions, and Xu *et al.* [64] combined LLMs with formal verification to discover missing peephole optimizations.

Characterizing WebAssembly Application Performance. Prior studies have proposed various methodologies ranging from standard benchmarking to differential testing to analyze execution behavior and detect performance defects. Jangda *et al.* [20] conducted the first comprehensive performance analysis by porting the SPEC benchmark suite to Wasm (BROWSIX-SPEC), revealing significant slowdowns compared to native code due to increased load/store operations and safety checks. Building on this, Jiang *et al.* [21] introduced *WarpDiff*, a differential testing framework that compares execution time ratios across multiple Wasm runtimes to detect performance issues. Beyond general execution speed, specialized analyses have targeted domain-specific overhead: Hasselt *et al.* [49] and Pockstaller *et al.* [35] investigated energy efficiency profiles in mobile web environments, while Kjorveziroski *et al.* [22] examined cold-start latency variances in serverless runtimes. While these efforts focus on characterizing the performance of *existing Wasm applications*

with *fixed* binary inputs, our work *shifts the focus upstream to the optimization phase* (*i.e.,* `wasm-opt`). By identifying and eliminating inefficient code generation, we address the root cause of performance overhead, thereby benefiting all downstream execution environments.

Leveraging Dead Code for Compiler Validation. Dead code (or unreachable code) has served as a powerful lever for validating complex software systems. A landmark approach in this domain is *Equivalence Modulo Inputs (EMI)* introduced by Le *et al.* [23], which validates compilers by mutating unexecuted (dead) statements of a test program. Tools like Orion [23], Athena [24], and Hermes [39] instantiate EMI by stochastically pruning or inserting code in dead regions; any behavioral deviation in the compiled executable signals a miscompilation. Conversely, Tu *et al.* [47] (Xdead) investigated the inverse problem: whether Dead Code Elimination (DCE) passes erroneously remove *live* code, highlighting the risks of aggressive optimization. Moving from correctness to performance, Theodoridis *et al.* [45] utilized DCE as a "lens" to detect missed optimizations, as already discussed above, if a compiler fails to remove injected dead markers, it implies a failure in the preceding optimization passes that were supposed to render those markers dead.

9 Conclusion

In this paper, we presented a systematic approach leveraging structure-aware instrumentation and optimization monotonicity to identify MOs in `wasm-opt`. Our method uncovered 24 distinct MOs (20 fixed). Beyond detection, our root cause analysis revealed fundamental design limitations. We provide practical lessons and future directions for `wasm-opt` and similar optimizers. All artifacts are publicly available.

Acknowledgment

We thank the anonymous ISSTA reviewers and the shepherd for their valuable feedback. This work was partially supported by the Shanghai Special Program for Promoting High-Quality Industrial Development (Project No. 250401) and NSFC Project No. 62572192.

Data Availability

We make the source code and experiment data online available in an anonymous GitHub repository at https://github.com/anonymityanonymity993/WasmMOsDetector.

References

[1] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. *Proceedings of the 27th International Conference on Compiler Construction* (2018). doi:10.1145/3178372.3179521

[2] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. 2021. A survey on blockchain interoperability: Past, present, and future trends. *Acm Computing Surveys (CSUR)* 54, 8 (2021), 1–41. doi:10.1145/3471140

[3] Binaryen Contributors. 2026. Binaryen Optimizations. https://github.com/WebAssembly/binaryen#binaryen-optimizations.

[4] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2022. FiGO: Fine-Grained Query Optimization in Video Analytics. *Proceedings of the 2022 International Conference on Management of Data* (2022). doi:10.1145/3514221.3517857

[5] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *Acm Computing Surveys (Csur)* 53, 1 (2020), 1–36.

[6] Binaryen Contributors. 2022. wasm-opt: global varaible is not optimized. GitHub Issue #4947. https://github.com/WebAssembly/binaryen/issues/4947.

[7] Manjeet Dahiya and Sorav Bansal. 2017. Modeling Undefined Behaviour Semantics for Checking Equivalence Across Compiler Optimizations. In *Haifa Verification Conference*. doi:10.1007/978-3-319-70389-3_2

[8] Emscripten Contributors. 2026. Emscripten. https://github.com/emscripten-core/emscripten.

[9] Emscripten Contributors. 2026. Emscripten Test Suite. https://emscripten.org/docs/getting_started/test-suite.html. Accessed: 2026-05-05.

[10] EOSIO. 2022. EOS VM: A Low-Latency, High Performance and Extensible WebAssembly Engine. https://github.com/EOSIO/eos. Accessed: 2026-01-25.

[11] Phani Kishore Gadepalli, Sean P. McBride, Gregor Peach, L. Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge. *Proceedings of the 21st International Middleware Conference* (2020). doi:10.1145/3423211.3425680

[12] Phani Kishore Gadepalli, Gregor Peach, L. Cherkasova, Rob Aitken, and Gabriel Parmer. 2019. Challenges and Opportunities for Efficient Serverless Computing at the Edge. *2019 38th Symposium on Reliable Distributed Systems (SRDS)* (2019), 261–2615. doi:10.1109/SRDS47363.2019.00036

[13] Fengjuan Gao, Hongyu Chen, Yuewei Zhou, and Ke Wang. 2024. Shoot Yourself in the Foot—Efficient Code Causes Inefficiency in Compiler Optimizations. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1846–1857. doi:10.1145/3691620.3695548

[14] grain. 2025. https://github.com/grain-lang/grain[Accessed: 2025-05-29].

[15] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 185–200. doi:10.1145/3062341.3062363

[16] David R Hanson. 1999. Early Experience with ASDL in lcc. *Software: Practice and Experience* 29, 5 (1999), 417–435. doi:10.48550/arXiv.cs/9810013

[17] Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs. *IPSJ Transactions on System and LSI Design Methodology* 9 (2016), 21–29. doi:10.2197/ipsjtsldm.9.21

[18] Devon Hockley and Carey L. Williamson. 2022. Benchmarking Runtime Scripting Performance in Wasmer. *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (2022). doi:10.1145/3491204.3527477

[19] Davide Italiano and Chris Cummins. 2024. Finding missed code size optimizations in compilers using llms. *arXiv preprint arXiv:2501.00655* (2024). doi:10.1145/3708493.3712686

[20] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: Analyzing the performance of {WebAssembly} vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 107–120.

[21] Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R Lyu. 2023. Revealing performance issues in server-side webassembly runtimes via differential testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 661–672. doi:10.1109/ASE56229.2023.00088

[22] Vojdan Kjorveziroski and Sonja Filiposka. 2023. Webassembly as an enabler for next generation serverless computing. *Journal of Grid Computing* 21, 3 (2023), 34. doi:10.1007/s10723-023-09669-8

[23] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. doi:10.1145/2666356.2594334

[24] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *Acm Sigplan Notices* 50, 10 (2015), 386–399. doi:10.1145/2858965.2814319

[25] Jay P. Lim and Vinod Ganapathy anFd Santosh Nagarakatte. 2017. Compiler Optimizations with Retrofitting Transformations: Is there a Semantic Mismatch? *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security* (2017). https://api.semanticscholar.org/CorpusID:21163089

[26] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring missed optimizations in webassembly optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 436–448. doi:10.1145/3597926.3598068

[27] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015). doi:10.1145/2813885.2737965

[28] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. 2021. WebAssembly Modules as Lightweight Containers for Liquid IoT Applications. In *International Conference on Web Engineering*. https://api.semanticscholar.org/CorpusID:234365967

[29] Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, Simone Romano, Salvatore Lombardi, Giuseppe Scanniello, and Patricia Lago. 2023. JavaScript Dead Code Identification, Elimination, and Empirical Assessment. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3692–3714. doi:10.1109/TSE.2023.3267848

[30] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. 2021. Learning to Make Compiler Optimizations More Effective. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. doi:10.1145/3460945.3464952

[31] Pankaj Mendki. 2020. Evaluating Webassembly Enabled Serverless Approach for Edge Computing. *2020 IEEE Cloud Summit* (2020), 161–166. doi:10.1109/IEEECloudSummit48914.2020.00031

[32] Jason Merrill. 2003. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Summit*. 171–180.

[33] Tipp Moseley, Dirk Grunwald, and Ramesh V. Peri. 2009. OptiScope: Performance Accountability for Optimizing Compilers. *2009 International Symposium on Code Generation and Optimization* (2009), 254–264. https://api.semanticscholar.org/CorpusID:2506168
[34] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016). https://api.semanticscholar.org/CorpusID:1410773
[35] Dennis Pockstaller, Stefan Huber, and Lukas Demetz. 2023. Comparing the Energy Consumption of WebAssembly and JavaScript in Mobile Browsers.. In *WEBIST*. 121–127.
[36] Alan Romano and Weihang Wang. 2020. WASim: Understanding WebAssembly Applications through Classification. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020), 1321–1325. https://api.semanticscholar.org/CorpusID:229702617
[37] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[38] Benedikt Spies and Markus U. Mock. 2021. An Evaluation of WebAssembly in Non-Web Environments. *2021 XLVII Latin American Computing Conference (CLEI)* (2021), 1–10. https://api.semanticscholar.org/CorpusID:245421441
[39] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 849–863.
[40] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. 2022. Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking. *2022 IEEE International Symposium on Workload Characterization (IISWC)* (2022), 256–268. https://api.semanticscholar.org/CorpusID:254639663
[41] Jialiang Tan, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2020. What every scientific programmer should know about compiler optimizations?. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.
[42] The AssemblyScript Authors. 2026. AssemblyScript. https://github.com/AssemblyScript/assemblyscript.
[43] The rustwasm Contributors. 2025. wasm-bindgen. https://github.com/rustwasm/wasm-bindgen.
[44] The TinyGo Authors. 2026. TinyGo. https://github.com/tinygo-org/tinygo.
[45] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.
[46] Theodoros Theodoridis and Zhendong Su. 2024. Refined input, degraded output: The counterintuitive world of compiler behavior. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 671–691.
[47] Haoxin Tu, Lingxiao Jiang, Debin Gao, and He Jiang. 2024. Beyond a joke: Dead code elimination can delete live code. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 32–36.
[48] Tweag I/O. 2025. Asterius: A Haskell to WebAssembly compiler. https://github.com/tweag/asterius.
[49] Max Van Hasselt, Kevin Huijzendveld, Nienke Noort, Sasja De Ruijter, Tanjina Islam, and Ivano Malavolta. 2022. Comparing the energy efficiency of webassembly and javascript in web applications on android mobile devices. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. 140–149.
[50] Weihang Wang. 2021. Empowering Web Applications with WebAssembly: Are We There Yet? *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), 1301–1305. https://api.semanticscholar.org/CorpusID:246081531
[51] Wenwen Wang. 2022. How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. *2022 IEEE International Symposium on Workload Characterization (IISWC)* (2022), 228–241. https://api.semanticscholar.org/CorpusID:253258973
[52] wasm-opt. 2025. release version 122. https://github.com/WebAssembly/binaryen/releases/tag/version_122.
[53] wasm-opt. 2025. release version 124. https://github.com/WebAssembly/binaryen/releases/tag/version_124.
[54] wasm-opt. 2025. release version 127. https://github.com/WebAssembly/binaryen/releases/tag/version_127.
[55] wasm-smith. 2025. wasm-smith. https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith.
[56] wasm-spec. 2025. https://webassembly.github.io/spec/core/syntax/instructions.html#control-instructions[Accessed: 2025-05-29].
[57] WebAssembly Community Group. 2026. Non-Web Embeddings. https://webassembly.org/docs/non-web/. Accessed: 2026-01-25.
[58] WebAssembly Group. 2026. Binaryen LocalCSE pass. https://github.com/WebAssembly/binaryen/blob/main/src/passes/LocalCSE.cpp. Accessed: 2026-07-16.
[59] WebAssembly Group. 2026. Binaryen MergeBlocks pass. https://github.com/WebAssembly/binaryen/blob/main/src/passes/MergeBlocks.cpp. Accessed: 2026-07-16.
[60] WebAssembly Group. 2026. Binaryen RemoveUnusedBrs pass. https://github.com/WebAssembly/binaryen/blob/main/src/passes/RemoveUnusedBrs.cpp. Accessed: 2026-07-16.

[61] WebAssembly Group. 2026. wasm-opt peephole optimizations. https://github.com/WebAssembly/binaryen/blob/main/src/passes/OptimizeInstructions.cpp. Accessed: 2026-07-16.
[62] Michel Weber, Theodoros Theodoridis, and Zhendong Su. 2025. Relaxing Alias Analysis: Exploring the Unexplored Space. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 201–224.
[63] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)* (2020), 1–4. https://api.semanticscholar.org/CorpusID:221106770
[64] Zhenyang Xu, Hongxu Xu, Yongqiang Tian, Xintong Zhou, and Chengnian Sun. 2025. Leveraging Large Language Models to Detect Missed Peephole Optimizations. *arXiv preprint arXiv:2508.16125* (2025).
[65] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*. https://api.semanticscholar.org/CorpusID:868674
[66] Yi Zhang, Yu Wang, Linzhang Wang, and Ke Wang. 2025. Synchronized Behavior Checking: A Method for Finding Missed Compiler Optimizations. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 1541–1568.

Received 2026-01-30; accepted 2026-06-25