

## RESEARCH ARTICLE

# Tail Call Optimization Tailored for Native Stack Utilization in JavaScript Runtimes

HYUKWOO PARK<sup>ID</sup> AND SEONGHYUN KIM

Samsung Research, Samsung Seoul Research and Development Campus, Seocho-gu, Seoul 06765, Republic of Korea

Corresponding author: Hyukwoo Park (hyukwoo.park@samsung.com)

This work was supported by Samsung Research, Samsung Electronics Company Ltd.

**ABSTRACT** JavaScript has significantly evolved, broadening its capabilities. However, the uptake of tail call optimization (TCO) remains limited, largely due to concerns about debugging difficulties and the potential increase in overall complexity. This paper highlights the compelling importance of tail calls within web applications, advocating for TCO as a means to boost performance and enhance memory efficiency. We present an innovative TCO technique that leverages the native stack within JavaScript engines, capitalizing on the native stack's inherent benefits over heap memory. This technique is carefully crafted to comply with diverse TCO standards, prioritizing simplicity and providing debugging features. We tackle the inherent challenges of TCO and successfully deploy our method in a lightweight JavaScript engine. Our approach is rigorously evaluated, proving its effectiveness and practicality. Notably, our implementation facilitates considerable memory savings for web applications, comparable to the maximum Resident Set Size (RSS), and achieves an average performance improvement of approximately 19.8% for algorithms that are heavily recursive. This research not only demonstrates the versatility and efficiency of our TCO strategy but also makes a significant contribution to the wider adoption and comprehension of TCO, thereby improving JavaScript engines' performance and memory management efficiency.

**INDEX TERMS** Code optimization, compiler, interpreter, JavaScript, JavaScript engine, tail call optimization.

## I. INTRODUCTION

JavaScript, originally designed for client-side, event-driven interactions within web browsers, has significantly evolved. Through continuous improvement and the incorporation of diverse language features, JavaScript has expanded its reach into various domains beyond the web environment. These extensions include server constructions [1], IoT solutions [2], and even AI frameworks [3]. Consequently, JavaScript has become one of the most popular programming languages, a fact presented in the state of the octoverse 2023 report [4].

The JavaScript language specification, as detailed by the ECMAScript standard [5], is updated annually to include new features that browser vendors generally implement in a timely manner. However, one feature that has seen inconsistent support is *tail call optimization*, shortly TCO. Although TCO

was introduced in the 2015 revision of the ECMAScript standard [6], only a handful of JavaScript engines offer limited TCO support, as indicated by an online compatibility table [7] (which notes TCO support in only 7 out of more than 40 surveyed web engines). TCO is an optimization strategy that aims to efficiently manage recursive function calls by curbing the expansion of the call stack. In JavaScript, a tail call occurs when a function's last operation is to invoke another function or itself in the case of recursion, immediately returning the result of the callee, as illustrated in Figure 1. TCO streamlines this process by allowing the JavaScript engine to reuse the current call frame for the subsequent call, thereby decreasing overall stack memory usage and preventing stack overflow exceptions that may arise during prolonged recursive operations.

The primary reason for the incomplete adoption of TCO in modern web engines stems from uncertainty regarding the trade-offs between benefits and potential

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu<sup>ID</sup>.

```
function foo(n) {
  if (n <= 0) {
    return 0;
  }
  return bar(n - 1); // tail call
}
```

(a) Tail Call Case

```
function foo(n) {
  if (n <= 0) {
    return 0;
  }
  return foo(n - 1); // tail recursion
}
```

(b) Tail Recursion Case

**FIGURE 1.** Examples of tail calls, with tail recursion as a specific case of tail call where a function invokes itself recursively in the tail call position.

drawbacks, including challenges in debugging and implementation. Developers who rely on stack information, such as `Error.stack`, for debugging their code may encounter issues due to missed call frames caused by TCO. Additionally, implementing TCO can pose a burden on web engines, as it requires extra stack management, potentially increasing the overall complexity of the engines. In an effort to address these concerns surrounding TCO, an alternative approach called *syntactic tail call* (STC) [8] was proposed. STC aimed to make tail calls explicit, offering developers the option to choose TCO or not. Unfortunately, this proposal was ultimately rejected. As a result, the original TCO specification remains largely disregarded, with only a few web engines incorporating it. According to the compatibility table [7], WebKit [9], the browser engine developed by Apple primarily for its applications such as Safari [10], is the only prevalent browser implementing TCO. Other web engines that support TCO are typically designed for resource-constrained devices, where TCO can help reduce overall memory consumption—a critical factor in such devices.

Based on this historical context, we embarked on an exploration of the present state of web applications to gain valuable insights into the active utilization of TCO. Our focus centered exclusively on JavaScript aspects, driving our evaluation using the web-tooling-benchmark suite [11]. This suite is specifically designed to measure the JavaScript-related workloads commonly found in web developer tools. The data presented in Table 1 offers a comprehensive view of tail calls, including their prevalence, frequency and actual call counts across various scenarios. Specifically, the `Tail Call Occurrence` column provides the total number of return statements in tail call positions for each application. Each figure is complemented by respective proportions relative to the overall number of return statements, indicated within parentheses. The `Tail Call Count` column presents the

cumulative count of invoked tail calls, along with proportions relative to the total call count, also shown in parentheses. Lastly, the `Tail Recursion Count` column signifies the total count of tail recursion calls, where a tail call invokes itself recursively (the `Tail Call Count` column encompasses both tail recursion and general tail call cases). We paid particular attention to these special tail call cases, as tail recursion often involves extensive self-involutions, potentially impacting overall application performance significantly. Remarkably, the table highlights a significant occurrence of tail calls within return statements of functions, ranging from 13.89% to 33.53%. Furthermore, a substantial number of applications already leverage tail recursion, as seen in the `Tail Recursion Count` column of Table 1, illustrating the practical applicability of TCO. While the overall percentage of tail calls concerning the total call count may not be overwhelming, their potential impact on memory management through TCO is undeniable, given the substantial number of tail calls that are invoked. This analysis further reinforces the compelling need for TCO support within JavaScript engines, especially when considering resource-limited devices.

In this paper, we introduce an advanced TCO technique that harnesses the native stack of JavaScript engines. Specifically, we focus on a lightweight JavaScript runtime that utilizes the native stack of the engine for JavaScript call frames. In contrast to other major JavaScript engines that allocate JavaScript call frames in separate memory spaces, such as heap memory, it merges JavaScript call frames into the native stack of the engine. This approach offers several advantages in memory management, as call frames are automatically reclaimed after a JavaScript function call completes, similar to native function calls. In contrast, when call frames are allocated in a separate memory space, JavaScript engines require manual reclamation of each call frame, which can be resource-intensive. Moreover, this approach can alleviate the burden on memory systems, including garbage collection, potentially resulting in performance enhancements. Our primary goal is to efficiently expand the use of the native stack for TCO, enabling the reuse of the current JavaScript call frame allocated in the native stack for subsequent tail calls. To the best of our knowledge, the adoption of TCO while exploiting the native stack of the language engine has not been proposed before.

While the concept may appear straightforward, yet it introduces complex challenges, particularly in managing the native stack which is inherently more restrictive than manipulating heap memory. Adjusting or expanding the native stack while a JavaScript engine runs poses significant limitations. Our key objectives include ensuring comprehensive support for the diverse TCO scenarios specified in the language standard to guarantee a consistent TCO feature for web developers. Additionally, we prioritize implementing our proposal with minimal complexity to facilitate JavaScript engine maintenance. Lastly, we emphasize the importance of providing additional stack information related to TCO for

**TABLE 1.** The state of tail call in the web-tooling-benchmark.

Web Applications	Tail Call Occurrence	Tail Call Count	Tail Recursion Count
acorn	3190 (29.71%)	1304947 (4.56%)	8510 (0.02%)
buble	1025 (22.17%)	417315 (2.7%)	57603 (0.37%)
coffeescript	1604 (18.84%)	69260 (0.52%)	0 (0%)
espre	3441 (27.23%)	1164643 (4.19%)	6938 (0.02%)
esprima	1187 (24%)	465857 (1.74%)	0 (0%)
jshint	1259 (13.89%)	27250 (0.17%)	0 (0%)
lebab	1640 (22.9%)	719708 (4%)	456 (0%)
postcss	1797 (17.9%)	949893 (4.27%)	0 (0%)
prepack	2957 (16.44%)	976749 (5.99%)	889 (0%)
prettier	5424 (18.48%)	355108 (2.04%)	3042 (0.01%)
source-map	505 (20.35%)	130653 (0.64%)	0 (0%)
terser	2589 (24.53%)	199612 (1.51%)	1428 (0.01%)
typescript	10261 (33.53%)	604524 (3.35%)	416 (0%)
uglify-js	11008 (26.27%)	472140 (2.61%)	4816 (0.02%)

debugging purposes. We have addressed these challenges and successfully integrated our TCO strategy into a lightweight JavaScript engine. Our implementation was rigorously tested, evaluating both its feasibility and its impact on performance. The contributions of this paper can be summarized as follows:

- **Assessment of TCO in Contemporary Web Environments:** We conducted a thorough analysis of the current implementation and support for TCO across various web engines and real-world web applications. This analysis highlights the critical need for enhanced TCO support to improve performance and efficiency.
- **Development of an Advanced TCO Approach:** We introduced a novel TCO technique that leverages the native stack efficiently. This approach is carefully designed to align with existing TCO standards, ensuring broad compatibility and applicability.
- **Implementation in a Lightweight JavaScript Engine:** Our TCO technique was successfully integrated into a lightweight JavaScript engine. The focus was on ensuring the implementation remained straightforward, avoiding unnecessary complexity to facilitate ease of adoption and maintenance.
- **Enhancement of Debugging Capabilities:** To aid developers in debugging, we extended the JavaScript engine to provide additional stack information relevant to TCO. This feature is invaluable for troubleshooting and understanding the behavior of optimized code.
- **Comprehensive Evaluation and Validation:** The effectiveness and practicality of our TCO approach were validated through extensive evaluations. Notably, our TCO implementation demonstrated significant memory savings—comparable to the maximum RSS (Resident Set Size) memory size—for general web applications. Furthermore, it yielded an average performance enhancement of around 19.8% on workloads that are heavily dependent on recursion.

The rest of this paper is structured as follows: Section II gives background information on the TCO standard and the JavaScript engine that leverages the native stack. Section III provides an in-depth description of the proposed TCO

method, tackling the challenges introduced above. Section IV presents the results of our evaluation, along with specific analyses. Section V reviews related works within the field. Finally, in Section VI, we conclude this paper and suggest directions for future research.

## II. BACKGROUND

This section provides background information on the TCO standard of JavaScript and the JavaScript engine that utilizes the native stack. It lays the foundation for understanding the core concepts discussed in this paper.

### A. TAIL CALL OPTIMIZATION IN JavaScript

A tail call occurs when a function invokes another function as its final operation and directly returns the result without further modification. Figure 2(a) shows an example of a tail call within the function  $f$ . In this scenario, the calling function has no additional instructions to execute, making its call frame redundant. This redundancy allows for the elimination or reuse of the call frame, a technique known as tail call optimization (TCO). TCO is particularly advantageous in programs where recursive functions are heavily utilized. With TCO, such programs can run without encountering stack overflow issues. Additionally, TCO can improve the speed of recursive algorithms by reusing the call frame instead of managing additional call frames.

According to the ECMAScript standard [5], several criteria must be met for a function call to be considered a tail call:

- The callee of the tail call should be in `strict` mode: `"use strict"`; in Figure 2(a) is a directive in JavaScript that ensures the code executes in `strict` mode, which enforces stricter parsing and error handling, leading to more reliable and secure code.
- The caller of the tail call should not be in the form of an `async`, `generator`, or `async-generator` function: these are special types of functions designed to handle asynchronous operations and streams of data. TCO is exclusively applied to conventional synchronous functions.
- `ReturnStatement`: `return \llexpr\gg;`

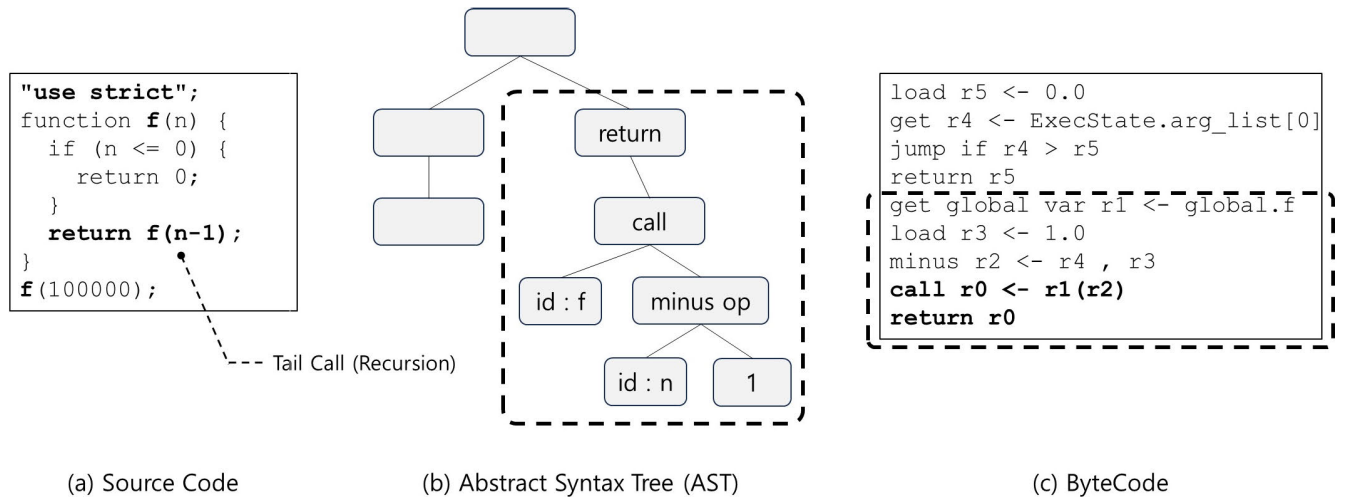


FIGURE 2. Transition of a JavaScript function with a tail call in the Escargot engine.

Only a return statement can encompass a tail call (arrow function may omit the return statement, but implicitly has one). While a return statement can encompass various expressions, including logical (&&, ||) or comma (,) expressions, the return statement must ultimately return the result of the tail call.

Recognizing a function call as a tail call enables the JavaScript runtime to optimize resource usage. It can either release the resources associated with the current call frame or reuse them for the next function invocation. While the ECMAScript standard defines the criteria for TCO application, the specifics of its implementation are left to the discretion of individual JavaScript engines. In alignment with the language standard, we propose a comprehensive application of TCO that optimizes various tail call scenarios, including tail recursion as depicted in Figure 2(a), by efficiently reusing the call frame for subsequent calls.

**B. JAVASCRIPT RUNTIME EXPLOITING NATIVE STACK**

Our proposed solution leverages the *Escargot* JavaScript engine [12], which is an open-source, lightweight engine mainly optimized for environments with limited resources, such as embedded systems. This is in contrast to more mainstream engines like V8 [13], JavaScriptCore [14], and SpiderMonkey [15], which may be unsuitable for such environments due to their larger memory requirements.

Escargot’s execution model is designed for efficiency, comprised of a parser and an interpreter responsible for executing JavaScript code. Initially, JavaScript source code is parsed into an Abstract Syntax Tree (AST), where each node represents a distinct language construct. The AST is then compiled into bytecode, serving as an intermediate, executable representation of the code.

Figure 2 (Figure (a), Figure (b), and Figure (c)) demonstrates the transformation of the function *f* from source code to AST and finally to bytecode. Within this figure,

the bold-dashed outlines connect the corresponding parts of the AST and bytecode that represent the tail call in the return statement `return f(n-1);`. During the parsing stage, the AST is traversed depth-first to generate the bytecode sequence, ensuring that the return statement’s result—the tail call—is processed appropriately. Escargot uses a register-based bytecode format, where operations are conducted using virtual registers (denoted as *r#*) that store temporary values, local variables, and parameters. As shown in Figure 2(c), the result of the tail call within *f* is stored in register *r0* which is in turn returned by the last `return` bytecode, corresponding to the concept of tail call.

After completing the code translation, the interpreter proceeds to execute the bytecode stream sequentially. To manage a function call, Escargot instantiates an interpreter instance and allocates a JavaScript call frame, which consists of *ExecState* and *RegisterFile*, directly on the native stack of the engine, rather than in heap space. This process is illustrated in Figure 3. A JavaScript call frame is a data structure used for storing the context needed to execute a function. The *RegisterFile* maintains virtual registers for the bytecode associated with each function call. Simultaneously, the *ExecState* contains essential execution pointers, such as the address of the current bytecode and references to the argument list, which facilitate execution within the interpreter. Specifically, the bytecode address points to the active bytecode, updating to the next in line upon the completion of the current one in the interpreter. Regarding call arguments, the virtual registers of the calling function, such as *r2* (as seen in Figure 2(c)), store the argument values, and the *ExecState*’s argument list points to these registers, which are located in the caller’s *RegisterFile*. During the target function’s execution, the interpreter accesses argument values via the *ExecState*, as demonstrated by the bytecode instruction `get r4 <- ExecState.arg_list[0]` in

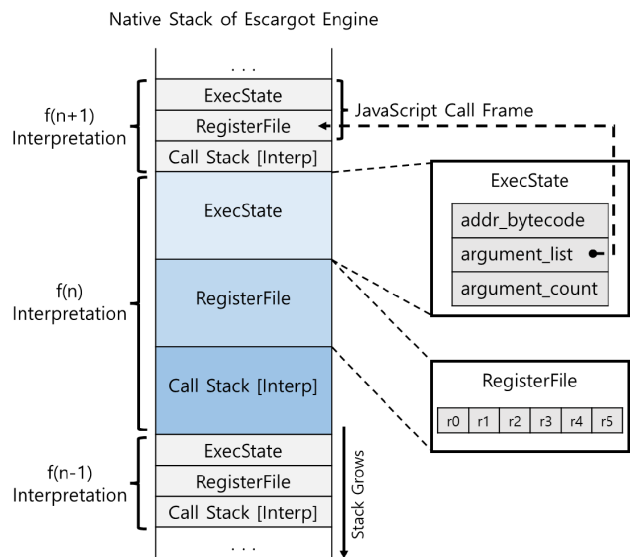


FIGURE 3. The native stack structure of the Escargot engine during the execution of a recursive function  $f$ .

Figure 2(c). These structures are instantiated just before invoking the interpreter.

Escargot distinguishes itself by efficiently utilizing the native stack, where the JavaScript function’s call frame and the interpreter’s call stack coexist. This approach greatly enhances data access speed for the interpreter because the JavaScript call frame resides in the same native stack space as the interpreter. Additionally, it leverages the native stack’s inherent memory management capabilities. Upon completion of a JavaScript function’s execution, both its call frame and the interpreter instance are automatically reclaimed by the system, substantially reducing the typical memory management overhead. In contrast, major JavaScript engines (like V8 [13], JavaScriptCore [14], and SpiderMonkey [15]) typically allocate the JavaScript call frame in a separate memory space and require manual management.

Figure 3 highlights a potential issue with the recursive tail calls in the function  $f$ . In their current form, these recursive calls still carry the risk of causing a stack overflow. To address this concern, our study enhances Escargot’s native stack utilization by introducing an advanced TCO technique. This innovation aims to maximize the utilization of the native stack, effectively mitigating the limitations that can lead to stack overflow and reducing overall stack usage. It’s important to note that asynchronous constructs, such as `async`, `generator`, and `async-generator` functions require heap allocation for their call frames. However, we have excluded them from our TCO scheme based on the tail call criteria outlined in Section II.A.

### III. ADVANCED TAIL CALL OPTIMIZATION

This section provides a detailed description of the proposed TCO method, specifically designed for leveraging the native

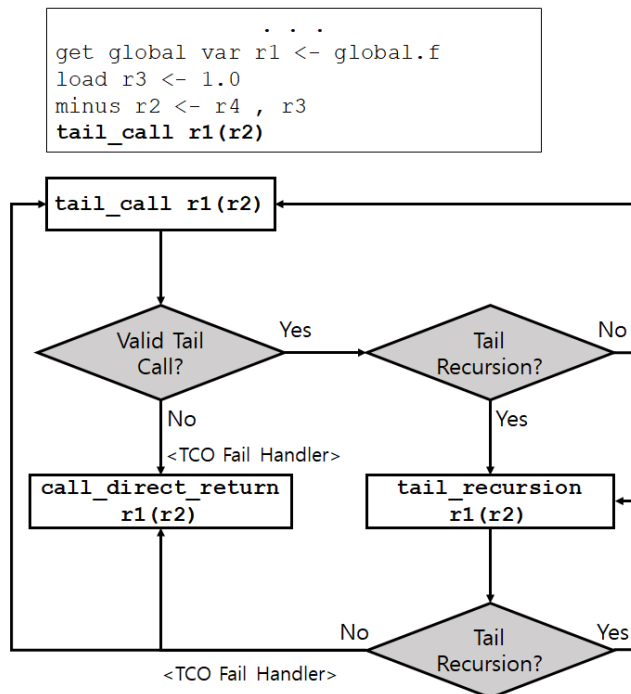


FIGURE 4. Bytecodes generated for TCO and their transitions, including fallback to direct return in case of TCO failure.

stack of the JavaScript engine. It also addresses the various challenges associated with this approach.

#### A. OVERVIEW OF TCO IMPLEMENTATION

Our method extends the capabilities of the native stack, optimizing all tail calls by reusing the native stack structures. We integrate this advanced TCO into the parsing and interpretation phases of the Escargot engine’s execution model.

##### 1) PARSING PHASE

During the parsing phase, the parser’s role includes generating specialized bytecode for tail calls. As illustrated in Figure 2(b), when encountering a return statement, the parser assigns a virtual register to store the expected return value. It then proceeds to generate bytecode for each child node iteratively, maintaining the association with the allocated register. While generating bytecode for the call expression, the parser verifies whether the call’s destination register matches the previously allocated register, indicating a potential tail call.

When the caller satisfies the tail call criteria specified in Section II.A, the parser generates a `tail_call` bytecode, specifically optimized for tail call execution, as depicted in Figure 4. This process focuses solely on the caller function, given that, unlike in statically-typed languages, the callee function can vary dynamically. If the caller does not satisfy the tail call criteria, the parser instead generates a `call_direct_return` bytecode. This alternative bytecode facilitates further optimization by directly handling the

return operation, capitalizing on situations where a tail call optimization is not applicable but performance gains can still be achieved.

Within the context of these optimized bytecodes, the conventional `return` bytecode becomes redundant and is consequently omitted. The `tail_call` bytecode, specifically designed for tail calls, defers the actual return operation to the subsequent call, as it directly returns the result value of the callee. In Figure 2(a), the `return 0;` statement serves as the exit point for recursive tail calls. In contrast, the `call_direct_return` bytecode encapsulates the return operation within its execution logic, with further details to be explored in an upcoming section. The parser’s primary role involves identifying tail call opportunities and generating the corresponding bytecodes. Concurrently, the interpreter, leveraging these custom instructions, is responsible for effectively executing TCO.

## 2) INTERPRETATION PHASE

In the interpretation phase, the interpreter aims to optimize the tail call by reusing the current function’s data structures, including `ExecState`, `RegisterFile`, and the interpreter’s own call stack. Due to the engine’s strategy of sharing the native stack with other modules, managing this stack requires a calculated and methodical approach. Furthermore, the interpreter needs to verify a set of conditions that must be satisfied before applying TCO.

During the processing of `tail_call`, the interpreter first verifies whether the target function of the tail call (callee) meets the tail call criteria specified in Section II.A. This verification is essential for every `tail_call` execution due to JavaScript’s dynamic nature, where functions are treated as first-class citizens [16] and can be reassigned. This flexibility means that the callee could potentially change during execution, unlike in statically typed languages. For example:

```
function other_func(m) { ... }
```

```
function f(n) {
  if (n <= 0) { return 0; }
  if (n == 100) {
    f = other_func; // set to another function
  }
  return f(n-1); // Is a valid tail recursion?
}
```

```
f(100000);
```

In this snippet, the function `f` could be altered even in the middle of recursive calls, potentially invalidating the TCO process.

After validating the callee, the interpreter assesses the overall size of the callee’s call frame to ensure it can fit within the current call frame of the calling function (caller). This constraint arises because the call frames of JavaScript functions and other engine internal methods/modules are

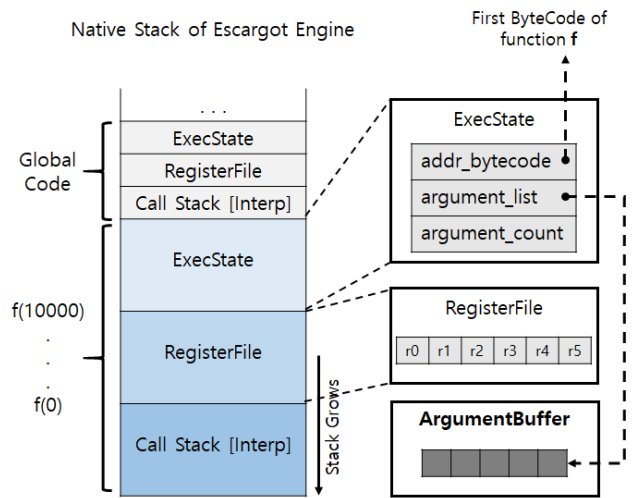


FIGURE 5. Reuse of native stack structures in the Escargot engine during the execution of a recursive tail call `f`.

allocated sequentially and densely on the same native stack. Therefore, resizing or enlarging the already assigned call frame of the caller is not a feasible option. Moreover, the call frame of a JavaScript function is designed for sequential alignment, with interpreters accessing stack data accordingly. If TCO were applied to a larger callee function, additional stack space would need to be allocated separately from the caller’s call frame. However, this would disrupt the sequential alignment of the call frames and invalidate bytecode execution through the interpreter. Consequently, TCO is exclusively applied to smaller or equal callee functions during tail calls to ensure compatibility with the existing stack allocation scheme and maintain the integrity of bytecode interpretation. In the case of tail recursion, where tail calls are invoked recursively, the TCO approach is fully applicable because both the caller and callee require identical call frame sizes. This limitation in call frame size represents the sole constraint imposed by native stack utilization.

Once the previous conditions are met, the interpreter proceeds to execute TCO. Figure 5 describes the overview of the native stack structures when TCO is applied to the source code depicted in Figure 2(a). The interpreter updates the current `ExecState` to reflect the context of the callee. The `RegisterFile` is entirely reused without any modification, as it matches the requirement of the target function. The interpreter then jumps to the first bytecode of the callee without invoking another interpreter instance, effectively reusing the entire stack structure for the tail call. This approach avoids incremental stack allocation, which could otherwise result in stack overflows, as depicted in Figure 3.

An additional consideration pertains to handling argument values. As shown in Figure 3, the original arguments are located in the `RegisterFile` of the caller function. If this `RegisterFile` is reused in the callee function, it may

inadvertently alter the argument values during execution, potentially leading to incorrect operations. To address this issue, we allocate a buffer called `ArgumentBuffer` in a separate memory space. This buffer serves as a temporary space to hold the argument values for subsequent tail calls. The interpreter copies the argument values to this predefined buffer, and `ExecState` points to this buffer as the argument list. Subsequently, the interpreter executes the bytecode of the callee while accessing the argument values via the `ArgumentBuffer`. Since these argument values are eventually allocated into virtual registers at the start of the function, as demonstrated in Figure 2(c) by the bytecode `get r4 <- ExecState.arg_list[0]`, the `ArgumentBuffer` needs to hold the arguments temporarily. Consequently, just one `ArgumentBuffer` can be shared among all tail calls, including nested tail calls or tail recursion cases, as illustrated in Figure 5.

If the previous checks fail, the interpreter promptly jumps to the fail handler to manage non-valid tail calls. In this scenario, the fail handler treats the target function invocation as a regular call. Moreover, it converts the `tail_call` instruction into `call_direct_return`, under the assumption that a function (callee) which failed in TCO once is likely to fail again. This strategy is based on a previous analysis [17] indicating that over 81% of calls in JavaScript are monomorphic, implying that the majority of call sites invoke the same function repeatedly. Therefore, instead of attempting TCO again in subsequent iterations, we opt to abandon it immediately to reduce the check overhead. For `call_direct_return`, the interpreter initiates the target function following the standard procedure of a regular call. Subsequently, the result of the call is directly returned instead of progressing to the next bytecode execution, as it represents the final operation of the current function.

In the context of recursive tail calls, we propose a specialized bytecode aimed at optimizing the execution of tail recursion, which can significantly impact overall performance. This new bytecode, `tail_recursion`, is designed to speed up the execution of recursive tail calls compared to the standard `tail_call` bytecode. Initially, the parser generates the `tail_call` bytecode, as it cannot predict in advance whether the tail call will be recursive or not. During the execution of the `tail_call`, the interpreter performs an additional check to determine if it is a tail recursion case by comparing the callee and caller objects. If this check confirms a tail recursion case, the `tail_call` bytecode is transformed into `tail_recursion`, allowing subsequent identical calls to be handled by the specialized bytecode. The interpreter then executes the `tail_recursion` more efficiently with a simplified check that compares the callee and caller objects to confirm the tail recursion case. This check is necessary because the callee of the call could be changed during runtime as seen in the previous example. After the check, the interpreter directly updates the JavaScript

---

**Algorithm 1** Interpretation Routine of `tail_call`


---

**Require:** *callee*, *caller*, *callee\_stack\_size*, *caller\_stack\_size*

- 1: **if** verifying *callee* fails OR *callee\_stack\_size* > *caller\_stack\_size* **then**
- 2:     **goto** TCO-Fail-Handler
- 3: **end if**
- 4: **if** *callee* == *caller* **then**
- 5:     transform `tail_call` into `tail_recursion`
- 6: **end if**
- 7: Copy argument values to *ArgumentBuffer*
- 8: Update *ExecState* to target function *callee*
- 9: Interpreter jumps to the first bytecode of target function *callee*

---



---

**Algorithm 2** Interpretation Routine of `tail_recursion`


---

**Require:** *callee*, *caller*

- 1: **if** *callee* ≠ *caller* **then**
- 2:     **goto** TCO-Fail-Handler
- 3: **end if**
- 4: Copy argument values to *ArgumentBuffer*
- 5: Update *ExecState* to target function *callee*
- 6: Interpreter jumps to the first bytecode of target function *callee*

---



---

**Algorithm 3** Interpretation Routine of `call_direct_return`


---

**Require:** *callee*

- 1: Invoke callee following the procedure of regular call
- 2: Return the result of callee

---

call frame and jumps to the callee as it would with a `tail_call`. If the check fails, the fail handler handles it similarly to the `tail_call` by invoking the callee as a normal call. However, instead of directly converting the `tail_recursion` into `call_direct_return`, the fail handler rechecks if the callee could still be handled by the general `tail_call`. If the callee satisfies the tail call criteria, the `tail_recursion` is transformed into `tail_call`, expecting that subsequent calls could be handled using TCO. Otherwise, the `tail_recursion` is transformed into `call_direct_return`. The overall process of bytecode transition is depicted in Figure 4. The processes of specialized bytecodes are described in Algorithm 1, 2 and 3.

Our proposal seamlessly integrates additional bytecodes into the existing execution framework, ensuring the simplicity of the TCO implementation while maintaining the integrity of the core system. This approach avoids complex modifications and reduces potential risks to the overall performance. A more detailed analysis of the complexity impact will be discussed in the evaluation section later.

**B. CASE STUDY OF TAIL CALL**

This section addresses the management of corner cases within our TCO implementation.

**1) TAIL CALL WITH CLOSURE**

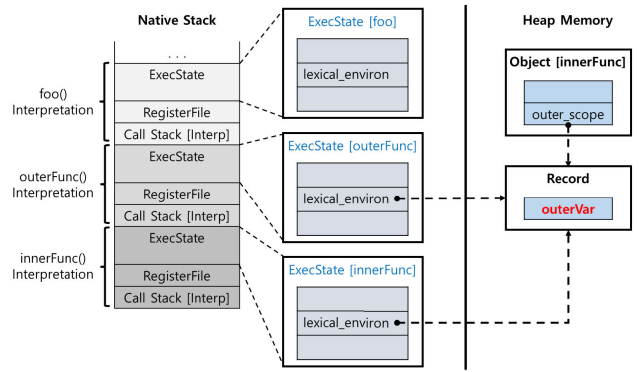
Closures in JavaScript represent a fundamental concept that is especially crucial in functional programming and the development of complex applications. A closure is created when a function (referred to as an inner function) is declared within another function (referred to as an outer function), allowing the inner function to capture and maintain access to the outer function’s scope even after the outer function has finished execution. This capability facilitates the creation of private variables, the encapsulation of data, and the employment of advanced function creation and management strategies.

```
"use strict";
function foo() {
  ...
  return outerFunc(); // tail call
}
function outerFunc() {
  var outerVar = 0;
  function innerFunc() {
    return ++outerVar;
  }
  innerFunc(); // return 1 (outerVar)
  return innerFunc;
}
```

```
var func = foo(); // return innerFunc
func(); // return 2 (outerVar)
```

In the above example, `innerFunc` exemplifies a closure as it is defined within `outerFunc` and maintains access to `outerVar`, a variable within the scope of `outerFunc`. Upon invocation of `outerFunc`, it returns `innerFunc`, which continues to have access to `outerVar` beyond the execution period of `outerFunc`, embodying the principle of a closure.

To accommodate closures, the Escargot engine employs a specialized data structure, named `Record`, stored in heap memory as depicted in Figure 6. Specifically, when `outerFunc` is invoked, a `Record` is allocated on the heap to preserve the value of `outerVar`, rather than storing it directly within the `RegisterFile`. The `ExecState` of `outerFunc` internally holds a reference to this newly created `Record`. Throughout the execution of `outerFunc`, any access to or modification of `outerVar` is conducted via this `Record`. Upon definition of `innerFunc` inside `outerFunc`, an object representing `innerFunc` is instantiated in the heap, inheriting the current scope’s `Record` reference. When `innerFunc` is eventually executed, its `ExecState` adopts the `Record` reference from the `innerFunc` object, thereby granting `innerFunc` access to `outerVar` contained within the `Record`. Since the `Record` resides independently within the heap, it persists even after the completion of `outerFunc`.



**FIGURE 6.** Illustration of closure management within the Escargot engine.

Incorporating closures into our TCO methodology is achieved by adhering to the established strategy. As demonstrated, upon tail call invocation of `outerFunc`, the existing call frame is reused for executing `outerFunc`. Simultaneously, a new `Record` for `outerVar` is allocated in the heap, with its reference updated in the current `ExecState`. Consequently, `outerFunc` and `innerFunc` operates as it typically would, unaffected by TCO. Thus, by leveraging the conventional strategy, our TCO implementation seamlessly supports the closure feature, avoiding any additional complexity.

**2) TAIL CALL IN TRY-CATCH-FINALLY**

The try statement in JavaScript consists of a try block, which is followed by either a catch block, a finally block, or both, as demonstrated in the following code snippet. The code within the try block is executed first, and if it encounters an exception, the code within the catch block is executed to handle the exception. Regardless of whether an exception occurs or not, the code within the finally block is always executed before control flow exits the entire construct.

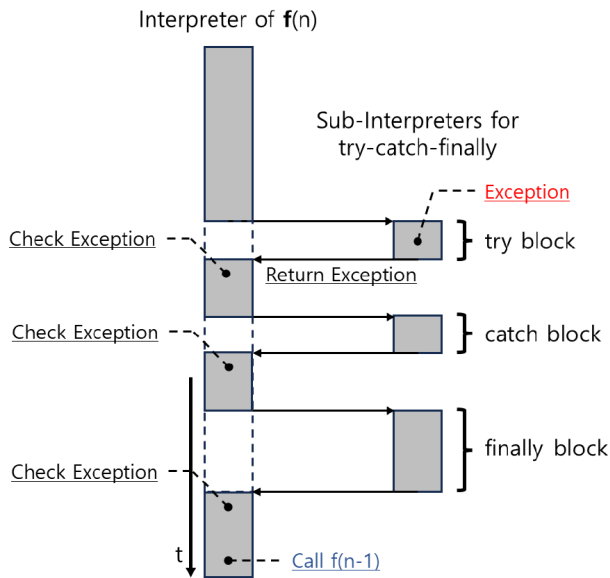
```
"use strict";
function f(n) {
  if (n <= 0) {
    return 0;
  }

  try {
    throw new Error(); // throw an exception
  } catch (err) {
    // handle exception
    ...
  } finally {
    // tail recursion
    return f(n-1);
  }
}

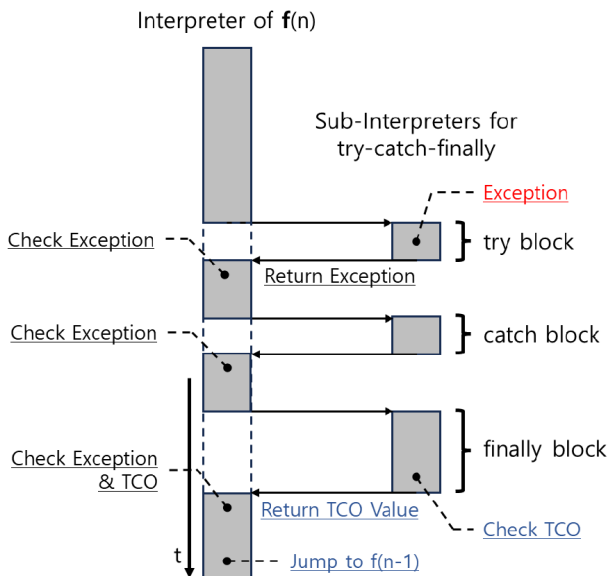
f(100000);
```

Escargot handles each block within the try statement by invoking a dedicated interpreter (sub-interpreter) to detect any exceptions that may occur in each block. If an exception arises in any of these blocks, it is returned to the main





(a) Execution flow of try-catch-finally



(b) Execution flow of try-catch-finally with TCO

**FIGURE 7. Illustration of a tail call within a finally block and its execution flow through the Escargot engine’s interpreter.**

interpreter, which then decides the subsequent course of action based on the returned result. In the above case where an exception is being triggered within the try block, this exception is passed back to the main interpreter of function  $f$ , as illustrated in Figure 7(a). Subsequently, the main interpreter checks the exception result and proceeds to invoke another sub-interpreter to handle the exception within the catch block. Finally, another sub-interpreter is invoked to execute the code within the finally block. This approach ensures proper exception handling and execution flow control within the try statement.

We implement our TCO for the try statement with simple modifications, ensuring that it does not disrupt the overall process. First, we consider that a tail call can occur in the catch block without the finally block or in the finally block itself, adhering to the principle that a tail call should be the last action of the function. During the parsing process, we identify such tail calls and generate a specialized bytecode called `try_tail_call` designed for the try statement. When the interpreter executes this bytecode, it applies the same TCO check procedure used for `tail_call`. If the call is indeed a valid tail call, a special value representing TCO is returned from the sub-interpreter of the catch or finally block. Instead of directly jumping to the target function within the block, this value serves as an indicator that TCO is applicable. In the final step, the main interpreter checks the returned value and, if it indicates TCO, directly jumps to the target function. This process allows us to reuse the native stack of the main interpreter, as represented in Figure 7(b).

It’s important to note that our implementation does not consider nested try statements (e.g., a try statement inside another try statement). When try statements are nested, it implies that the sub-interpreters for try, catch, and finally blocks are called in a nested manner. In such cases, delivering the special value of TCO from the deepest sub-interpreter to the uppermost main interpreter to trigger TCO can make the overall structure overly complicated. For the sake of maintaining the Escargot engine and ensuring simplicity, we have decided to support TCO only for non-nested try statements. Fortunately, cases of tail calls within nested try statements are rare, and there are no such cases in the web-tooling-benchmark, making this limitation acceptable for practical use.

### 3) TAIL CALL WITH eval

The `eval` function is a predefined function within the JavaScript global object. It is used to evaluate its argument as JavaScript code. When `eval` is called directly using the `eval(arg)` syntax, it executes the provided code within the scope in which it is called. However, handling direct calls to `eval` in Escargot involves a complex procedure as below:

- 1) The parser examines every function call with the name `eval` and generates a specialized bytecode for it.
- 2) The interpreter handles this bytecode differently from a regular function call. It first checks if the target function is the global `eval` function.
- 3) If the target function is not the global `eval` function, the interpreter treats it like a normal function and calls it.
- 4) If the target function is indeed the global `eval` function, the parser is invoked again to generate bytecode for the argument code.
- 5) The execution environment, including scope information, is organized, and the argument code is executed within a new interpreter instance.

The below example illustrates tail recursion involving the `eval` function. In this scenario, the `eval` property is overwritten with the function  $f$ , just before calling  $f$ . As a

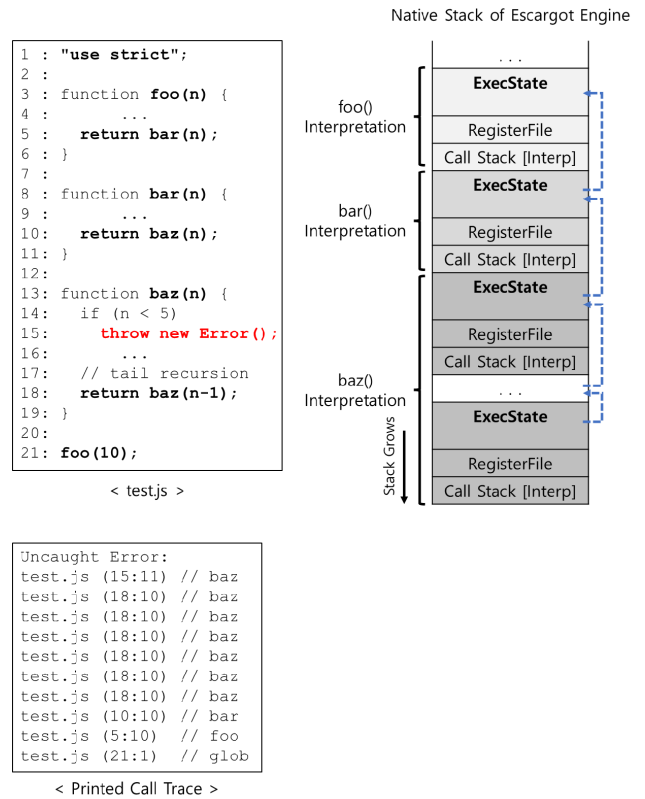
result, during the execution of `f`, it recursively calls itself due to the invocation of the modified `eval`. In order to support TCO for this specific case, changes must be made to the direct `eval` call mechanism. This includes incorporating additional checks and implementing the TCO process into the complex sequence of direct `eval` calls.

```
function f(n) {
  "use strict";
  if (n <= 0) {
    return 0;
  }
  return eval(n - 1); // tail recursion
}
eval = f; // overwrite eval
f(100000);
```

However, implementing this modification would require significant changes to the system, and as a result, combining TCO with `eval` could pose a burden on the system. In fact, using the `eval` function is strongly discouraged due to potential security risks and performance degradation it causes [18]. Previous works have addressed these issues by forbidding, filtering, or even removing the `eval` calls [19], [20], [21]. Therefore, we have chosen not to support TCO for the `eval` function. According to the analysis of the web-tooling-benchmark, tail calls with `eval` never occur, making our decision practical.

**C. DEBUGGING WITHIN TCO**

In JavaScript development, the call stack plays a pivotal role in debugging, as it reveals insights into function calls, their execution order, and the origins of errors. Developers often rely on language properties like `Function.caller`, `Function.arguments`, and `Error.stack` to inspect the stack. Additionally, Escargot inherently supports displaying call history during error occurrences. In Figure 8, the source code of `test.js` contains nested tail calls, where the function `foo` invokes `bar` as a tail call, `bar` then invokes `baz` as a tail call, and finally `baz` invokes itself recursively in a tail recursion manner. During recursive execution of `baz`, if an error occurs at line 15 in `test.js`, the call history from the error’s point of origin is displayed, as shown in Figure 8. To facilitate this debugging feature, the Escargot engine leverages the `ExecState` structure, where each call’s information is stored. In fact, each `ExecState` is linked to the next, organizing the entire call chain as depicted in Figure 8. When an error arises, Escargot traces from the current `ExecState` to linked ones, presenting each call information based on the `ExecState`. Furthermore, it can identify the last execution position in the source code, as the `ExecState` holds the address of the last executed bytecode. Each code position is represented with line and column numbers within parentheses in Figure 8. However, with TCO, these debugging properties become unavailable since call frames including `ExecStates` might be reused and consequently omitted during tail calls.



**FIGURE 8.** Illustration of call stack tracing using linked `ExecState` structures for debugging.

To maintain stack traceability with TCO, one could employ a *Shadow Stack* [22], [23], [24], which creates a separate set of call frames for debugging purposes. Despite its effectiveness in restoring the visibility of call frames reused by TCO, this approach incurs additional overhead and complexity, not ideal for lightweight engines like Escargot.

Our solution maintains debugging capabilities with minimal modifications. In non-strict mode, where TCO is inapplicable, properties enabled only in such mode like `Function.caller` and `Function.arguments` remain unaffected, thus requiring no additional support. However, for `Error.stack` and Escargot’s inherent stack tracing, basic diagnostic stack traces that outline the sequence and source of function invocations are needed, even in strict mode. The `Error.stack` property provides a trace of called functions, their order, and file locations, related to the `Error` instance, similar to Escargot’s default debugging feature. For tools that rely on these features, we leverage the existing `ExecState` structure, inherently linked along the call chain. Specifically, we allocate an `ExecState` for each tail call like normal function calls. For tail recursion, we allocate only one `ExecState` and reuse it, as countless `ExecStates` could be generated by recursive tail calls, potentially resulting in a stack overflow. Additionally, we include a tail recursion counter in `ExecState` to track the number of recursive tail calls. These enhancements

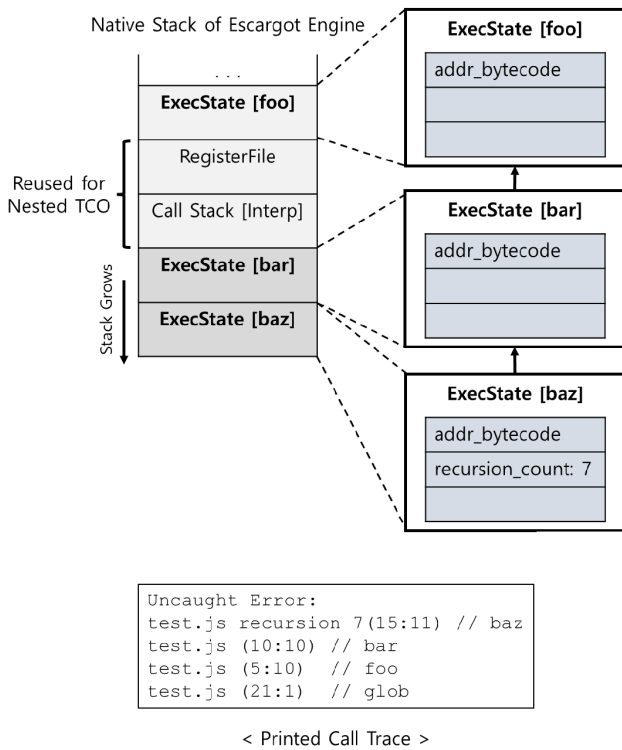


FIGURE 9. Illustration of call stack tracing with extended `ExecState` structures for nested TCO.

ensure that stack traces accurately reflect essential information.

Figure 9 demonstrates how the Escargot engine implements stack tracing for the source code depicted in Figure 8. `ExecStates` for functions `foo`, `bar`, and `baz` are sequentially allocated on the native stack, while other parts of the JavaScript call frame, including `RegisterFile` and interpreter’s call stack, are reused for all tail calls. The printed call history resembles that of Figure 8, where recursive calls of `baz` are briefly displayed along with a count indicating the occurrence of tail recursion. Notably, this debugging method is exclusively enabled in Escargot’s debug mode, whereas it remains disabled by default in release mode to uphold the performance advantages of TCO. Our debugging method provides only the minimal stack tracing information, focusing on the call sequence and function source, without call frame details. This approach allows for precise stack tracing within the Escargot engine under TCO, enabling developers to debug effectively without the need for a Shadow Stack and its associated overhead.

#### IV. EVALUATION

We conducted our evaluation on a system with an `x86_64` architecture, equipped with an Intel Core i7-7700 CPU and 16GB of RAM, running the Ubuntu 20.04 operating system. Our proposed TCO technique was implemented in the latest version of the Escargot engine, v4.1.0. We carried out comprehensive tests to assess the feasibility and performance

implications of our TCO implementation. The following sections outline our methodology, detailing the various aspects of our evaluation, and present a discussion of the achieved outcomes.

#### A. COMPLEXITY ANALYSIS

In assessing the complexity introduced to the Escargot engine by our TCO implementation, we conducted an analysis to quantify the changes. As described earlier, our approach involved enhancing the execution model with a limited set of additional bytecodes and associated routines. Complexity was evaluated by examining both the increase in the source code line count and the resulting binary size. An in-depth analysis of Escargot’s C++ codebase revealed that the TCO-related modifications accounted for a mere 0.59% (828 out of 139,562 lines) of the total lines of code. Furthermore, the binary size saw a marginal increase from 4.01MB to 4.02MB, representing a mere 0.2% growth in total. These metrics highlight the negligible footprint of the TCO code. The findings affirm that our TCO strategy seamlessly integrates into the engine, enhancing its capabilities without a proportionate increase in complexity. Consequently, the inherent efficiency and maintainability of the JavaScript runtime are preserved.

#### B. CONFORMANCE TESTING

The test262 suite [28], regularly updated to reflect both officially released and draft features of the ECMAScript specification, stands as the de facto standard for JavaScript engine conformance testing. Comprising over 50,000 tests covering various language behaviors and edge cases, it provides a comprehensive evaluation framework. For our evaluation, we specifically focused on tests tagged with `tail-call-optimization`, which directly evaluate TCO functionality. Additionally, we compared our work by testing other well-known JavaScript engines that support TCO, to see how they all perform on these tests.

As presented in Table 2, the tests were grouped according to specific features, with each engine’s support denoted by a pass (O), fail (X), or partial pass ( $\Delta$ ). Notably, all TCO-related tests focused on evaluating tail recursion scenarios and checking if a stack overflow occurred or not. Lightweight engines like Duktape [25], KinomaJS [26], and Moddable XS [27], tailored for resource-constrained environments similar to Escargot, were included in the evaluation. Among these, Moddable XS exhibited the broadest support for TCO features, while Duktape showed the least support. Both WebKit, the sole major browser engine equipped with TCO, and Escargot passed all tests except those involving `eval`.

While our approach did not achieve full compliance across the entire suite, it robustly supports the principal TCO features. The inability to pass `eval`-related tests holds minimal significance, given that `eval` is typically discouraged in practice and is absent from benchmark suites such as `web-tooling-benchmark`. It’s worth reiterating that,

**TABLE 2.** Comparative result of conformance testing on TCO features across TCO-enabled JavaScript engines.

TCO tests	Duktape 2.7.0 [25]	KinomaJS [26]	Moddable XS 4.2.1 [27]	WebKit 2.42.1 [9]	Escargot-TCO
Tail Call	O (3/3)	O (3/3)	O (3/3)	O (3/3)	O (3/3)
Coalesce Expression	X (0/2)	X (0/2)	O (2/2)	O (2/2)	O (2/2)
Comma Expression	O (1/1)	O (1/1)	O (1/1)	O (1/1)	O (1/1)
Conditional Expression	X (0/2)	O (2/2)	O (2/2)	O (2/2)	O (2/2)
Logical-And Expression	X (0/1)	O (1/1)	O (1/1)	O (1/1)	O (1/1)
Logical-Or Expression	X (0/1)	O (1/1)	O (1/1)	O (1/1)	O (1/1)
Tagged Template	X (0/2)	O (2/2)	O (2/2)	O (2/2)	O (2/2)
Block Statement	O (2/2)	O (2/2)	O (2/2)	O (2/2)	O (2/2)
DoWhile Statement	O (1/1)	O (1/1)	O (1/1)	O (1/1)	O (1/1)
For Statement	△ (2/4)	O (4/4)	O (4/4)	O (4/4)	O (4/4)
If Statement	O (2/2)	O (2/2)	O (2/2)	O (2/2)	O (2/2)
Switch Statement	O (3/3)	O (3/3)	O (3/3)	O (3/3)	O (3/3)
Try Statement	X (0/3)	△ (2/3)	△ (2/3)	O (3/3)	O (3/3)
While Statement	O (1/1)	O (1/1)	O (1/1)	O (1/1)	O (1/1)
Labeled Statement	O (1/1)	O (1/1)	O (1/1)	O (1/1)	O (1/1)
Eval	O (4/4)	X (0/4)	O (4/4)	X (0/4)	X (0/4)
<b>Total Pass Rate</b>	60.6% (20/33)	78.8% (26/33)	97.0% (32/33)	87.9% (29/33)	87.9% (29/33)

to the best of our knowledge, no other JavaScript runtimes simultaneously support TCO while leveraging the native stack. Other JavaScript engines listed in Table 2 allocate separate memory space for the JavaScript call frame and resize it to enable TCO. This demonstrates the effectiveness of our approach, enabling TCO utilization while adhering to JavaScript standard requirements and leveraging the native stack efficiently.

### C. PERFORMANCE ANALYSIS

Prior to evaluating the performance impact, we outline the specific benefits of TCO for performance enhancement as follows:

- **Reduced Call Overhead:** TCO significantly decreases the overhead associated with function calls in JavaScript engines. This reduction is accomplished by reusing the current call frame and interpreter instance, eliminating the need for separate call frame allocation and interpreter invocation for each tail call.
- **Improved Return Path:** TCO streamlines the return process for functions engaged in tail calls. It allows the engine to bypass intermediate return steps, directly transferring control back to the initial non-tail calling function. This advantage becomes more pronounced with increasing call chain depth, offering greater performance improvements for deeper recursive calls.
- **Enhanced Memory Utilization:** TCO enables the immediate clean up of local objects as a function enters a tail call. In the absence of TCO, local objects allocated in the call frame persist in memory for the duration of the function's execution, which can be prolonged in the presence of recursive calls. By employing TCO, however, the engine can immediately recycle the memory allocated to these local objects, as their lifetimes conclude with the function's tail call. This promptness in memory recovery not only optimizes memory usage

but also ensures that resources are available more quickly for subsequent allocations, thereby enhancing the overall memory management efficiency within the engine.

- **Increased Stack Space Efficiency:** By minimizing the native stack usage, TCO indirectly enhances memory cache performance. This optimization ensures that less cache is employed for call frame storage, potentially decreasing cache misses and improving overall system responsiveness.

To assess the impact of TCO on execution efficiency, we performed comparative evaluations using three distinct configurations of the Escargot engine:

- **Origin:** This configuration represents the original Escargot engine without incorporating TCO implementation. It allocates the JavaScript call frame on the native stack.
- **TCO-Stack:** In this configuration, the Escargot engine is equipped with TCO enabled, reusing the call frame (`ExecState` and `RegisterFile`) and the interpreter instance allocated in the native stack. This configuration corresponds to our proposed approach.
- **TCO-Heap:** Here, the Escargot engine is equipped with TCO enabled, but the call frame (`ExecState` and `RegisterFile`) is allocated in heap memory. This configuration was included to compare the efficiency of the native stack against heap management.

We evaluated performance using two test suites: the web-tooling-benchmark, which includes typical web applications, and a self-generated test suite containing four recursive algorithms. The web-tooling-benchmark was utilized to assess TCO performance for generic web applications. In contrast, we selected the four recursive algorithms where our TCO implementation could potentially provide the most significant performance improvements to evaluate the specific impact of TCO on performance.

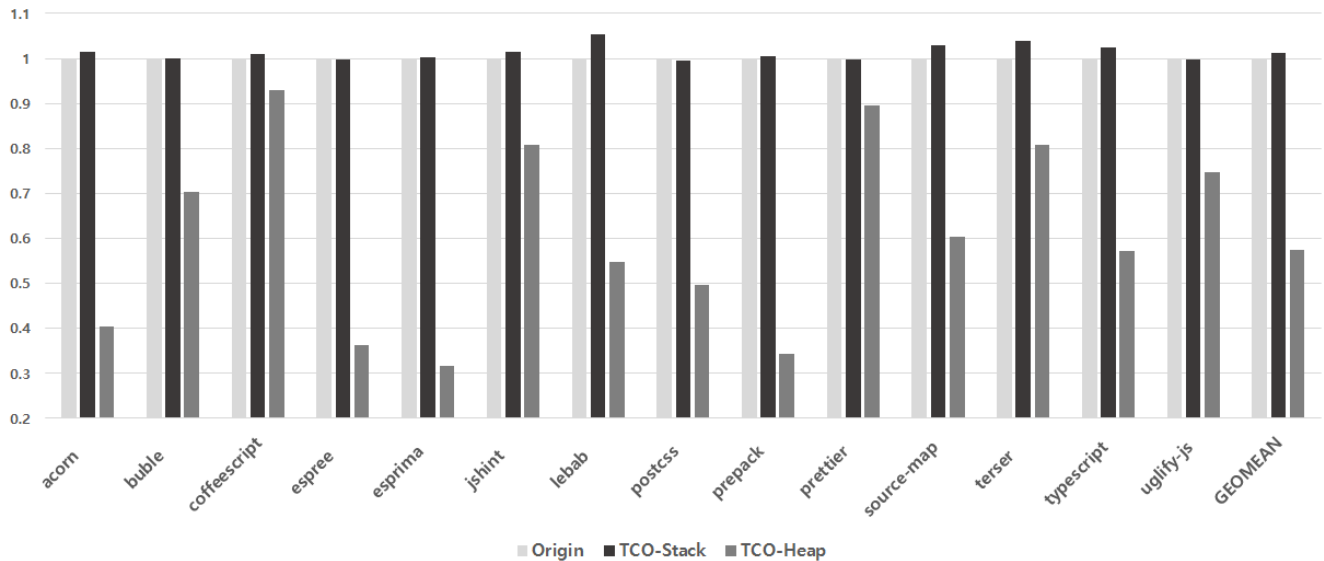


FIGURE 10. Performance gain of different Escargot configurations on the web-tooling-benchmark.

### 1) PERFORMANCE OF WEB-TOOLING-BENCHMARK

Figure 10 illustrates the performance results, normalized relative to the performance of the `Origin` configuration. Higher values on the graph indicate better performance. To ensure reliability, each test was executed ten times, and the graph represents the average performance outcome across these trials. Our proposed method (`TCO-Stack`) demonstrates a modest performance improvement of around 1.2% on average for the web-tooling benchmark. In contrast, `TCO-Heap` shows a significant performance decline, reaching only 57.4% of the `Origin` performance. This slight enhancement of `TCO-Stack` is expected, given that the proportion of tail calls among all function calls is insignificant (as depicted in Table 1), limiting the potential for performance gains through TCO. Nevertheless, it is crucial to emphasize that our TCO approach does not introduce any significant performance slowdowns by integrating tail call implementation into the Escargot engine, regardless of the frequency of tail calls.

On the other hand, `TCO-Heap` significantly degrades performance across all test cases. This downturn is largely attributed to the strategy of allocating JavaScript call frames on the garbage collector (GC)-managed heap, despite the adoption of TCO. GC identifies and reclaims unused memory objects (garbages), freeing up space for future use. The necessity for the GC to manage these heap-allocated call frames introduces a considerable overhead, which stands in contrast to the more streamlined management of call frames directly on the native stack. GC is typically triggered when the heap space is nearly full, leading to interruptions in the execution of the JavaScript engine and adversely affecting overall performance. As shown in Table 3, the total number of GC invocations substantially increases in `TCO-Heap` to reclaim the call frames, while `Origin`

TABLE 3. Number of garbage collections triggered during evaluation of the web-tooling-benchmark.

GC Count	Origin	TCO-Stack	TCO-Heap
acorn	52	46	451
buble	35	36	99
coffeescript	93	82	172
espre	53	53	428
esprima	47	46	408
jshint	46	42	144
lebab	74	64	350
postcss	102	101	383
prepack	46	45	336
prettier	170	168	257
source-map	47	45	237
terser	37	39	77
typescript	26	25	145
uglify-js	171	180	313

and `TCO-Stack` configurations exhibit comparable results, where call frames in the native stack are automatically reclaimed. This comparison highlights the superior efficiency of using the native stack for call frame management in the Escargot engine, reinforcing the benefits of native stack utilization over heap allocation.

Table 4 provides a comprehensive breakdown of the `TCO-Stack` execution for the web-tooling-benchmark, detailing the reused memory and tail call coverage. To assess the memory benefits of TCO, we measure the total size of reused stack memory and compare it to the maximum RSS memory of each test case execution to demonstrate its effectiveness. In several cases, significant memory savings are observed, such as in `acorn` (72.29%) and `espre` (62.22%), where the total size of reused memory rivals the max RSS memory. Although not all cases exhibit substantial

**TABLE 4.** Breakdown of the execution of TCO-Stack for the web-tooling-benchmark.

	Reused Memory Size (KB)	Reused Memory Size Compared to Max RSS (%)	Optimized Tail Call Count	TCO Coverage (%)
acorn	140962	72.29	840307	64.39
buble	50787	26.04	308163	73.84
coffeescript	1863	0.96	16384	23.66
espre	121326	62.22	720872	61.90
esprima	240	0.12	1469	0.32
jshint	2318	1.19	13629	50.01
lebab	35076	17.99	227489	31.61
postcss	24294	12.46	183455	19.31
prepack	39578	20.30	221614	22.69
prettier	6713	3.44	37206	10.48
source-map	2966	1.52	19879	15.22
terser	8324	4.27	48655	24.37
typescript	42285	21.68	270910	44.81
uglify-js	18328	9.40	112866	23.91

memory savings relative to the max RSS, the absolute value of reused memory remains significant. Moreover, all test cases unequivocally utilize tail calls and derive certain benefits from TCO. Given that the Escargot engine is designed for resource-constrained devices, this memory optimization could be more critical than performance enhancement.

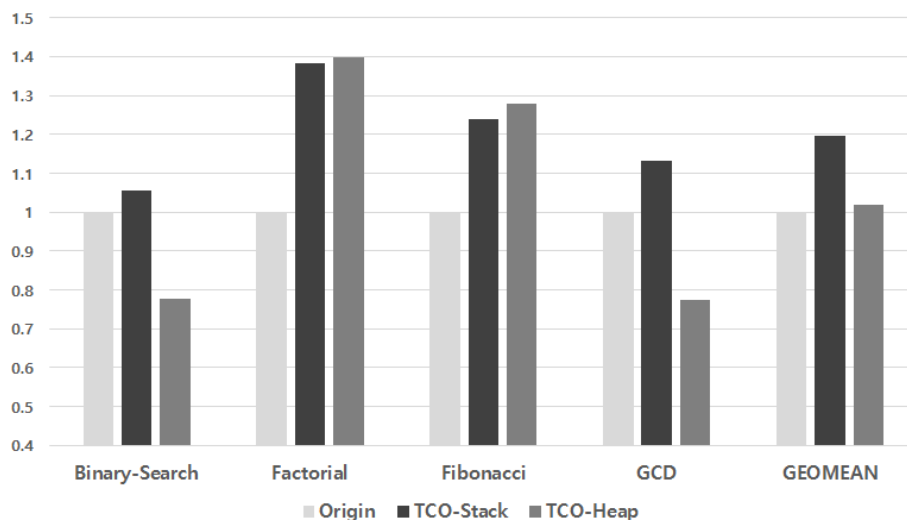
Additionally, Table 4 outlines the coverage of TCO to analyze the extent to which our approach optimizes tail calls. We measure the total count of tail calls optimized by our method and compare it with the overall tail call count from Table 1 to illustrate the TCO coverage. The results indicate varied TCO coverage, ranging from 0.32% to 73.84% across different test cases. The limited coverage primarily stems from an inherent constraint in our methodology, where the call frame of the callee must be smaller or equal to that of the caller. Notably, *esprima* exhibits exceptionally low TCO coverage (0.32%), due to certain call patterns where a simple function invokes a complex target function that requires large-sized call frames in a tail call manner. Consequently, we could not optimize these tail call patterns appropriately, resulting in suboptimal TCO coverage. However, when considering tail recursion, our approach effectively optimizes all tail recursion cases using the specialized bytecode *tail\_recursion*, with the exception of *buble*. In the case of *buble*, only 4.2% of recursive tail calls utilize *tail\_recursion*, while the remainder employ either the general TCO routine (*tail\_call*) or the direct return routine (*call\_direct\_return*). This disparity arises from partially recursive tail calls, where the target function of tail recursion is not always the caller itself, leading to a transformation to *tail\_call* or *call\_direct\_return* by the fail handler of *tail\_recursion*. Excluding this special case in *buble*, our approach fully optimizes tail recursion cases, which could significantly impact overall performance. To further examine the precise performance effects of TCO, we evaluate our approach on recursive algorithms in the subsequent section.

## 2) PERFORMANCE OF TAIL RECURSION

Tail recursion, characterized by its intensive self-involutions, stands as an ideal subject for realizing substantial performance improvements via TCO. To directly measure the potential impact of TCO on performance, we constructed four typical algorithmic tests that heavily rely on recursive tail calls. In order to execute these tests successfully without triggering stack overflow errors in the *Origin* configuration and yet place a significant computational load, we adjusted the recursion depth and the execution count for each test as described below:

- **Binary-Search:** This algorithm aims to find an item within a sorted array containing 5000 elements. It achieves this by recursively dividing the array's portion in half. The test is executed 10,000,000 times, with a maximum recursion depth of 13.
- **Factorial:** The factorial algorithm calculates the product of all positive integers less than or equal to the input number through recursive multiplication. The test is executed 10,000 times, with a maximum recursion depth of 10,000.
- **Fibonacci:** This algorithm computes a Fibonacci number by iteratively summing the two preceding ones. It is executed 10,000 times, with a maximum recursion depth of 10,000.
- **GCD (Greatest Common Divisor):** The GCD algorithm determines the largest number that divides two given numbers without leaving a remainder, employing the Euclidean algorithm. It is executed 10,000,000 times, with a maximum recursion depth of 16.

Figure 11 presents the performance outcomes of four recursive algorithms, normalized to the performance of the *Origin* configuration, with higher values indicating better performance. The TCO-Stack approach yielded improvements across all tested algorithms, reporting an average performance gain of 19.8%, aligning with our anticipations. This improvement directly results from our tailored TCO approach, which leverages the native stack for optimizing tail recursion.



**FIGURE 11.** Performance gain of different Escargot configurations on the four recursive algorithms.

Conversely, the TCO-Heap variant demonstrated a modest average improvement of 1.8%, though the results varied significantly across different tests. Specifically, TCO-Heap adversely affected the performance of Binary-Search and GCD by -22.3% and -22.6%, respectively. Both tests share a characteristic of high execution frequency. Even with TCO applied, the initial call in a tail recursion sequence requires allocating its call frame on the heap, which is then reused for subsequent recursive calls. The high execution count leads to an increase in heap-allocated call frames, triggering frequent GC invocations, as detailed in Table 5. On the other hand, TCO-Heap boosted the performance of Factorial and Fibonacci by 39.8% and 27.9%, respectively. These algorithms, which feature deep recursion with less frequent executions, benefitted from the TCO-Heap approach, even marginally outperforming TCO-Stack in terms of performance. This discrepancy arises because heap-based call frames are collected in bulk during the GC process rather than being individually reclaimed after each call. Meanwhile, the TCO-Stack strategy involves an immediate cleanup of stack-allocated call frames following tail call execution, which requires processing destructors for any stack-stored local objects, thereby adding extra overhead. This operational difference reveals that in scenarios with moderated heap allocation rates, such as those observed with Factorial and Fibonacci, TCO-Heap may deliver more efficient outcomes compared to TCO-Stack. Nonetheless, this efficiency is highly context-dependent, indicating that its benefits are not universally applicable across all use cases, as evidenced by varied results in web-tooling benchmarks.

The overall findings demonstrate that our TCO implementation not only facilitates modest performance improvements and significant memory savings in general web applications but also offers considerable performance enhancements for

**TABLE 5.** Number of garbage collections triggered during evaluation of the four recursive algorithms.

GC Count	Origin	TCO-Stack	TCO-Heap
Binary-Search	9	9	29860
Factorial	7	8	50
Fibonacci	7	8	52
GCD	7	7	24046

recursive-intensive workloads. This highlights the versatility and efficiency of our TCO strategy.

## V. RELATED WORK

The concept of TCO has already found successful implementation in various programming languages. Leading compilers like GCC [29] and LLVM/Clang [30] have integrated TCO into their optimization strategies, benefiting languages such as C and others when higher optimization levels are applied. Furthermore, the emerging WebAssembly standard [31], a key player in web technologies, is also beginning to embrace TCO as a draft feature [32].

Studies have explored the implementation of TCO in various language runtimes, with a particular focus on Java platforms. In a previous study [33], the Funnel compiler takes a unique approach to TCO. It augments all functions with an integer argument known as the Tail Call Counter (TCC), responsible for tracking consecutive tail calls on the call stack. When the count of tail calls exceeds a predefined threshold, the stack is dynamically shrunk, and the current continuation is passed back to a trampoline. This method offers control over worst-case stack space usage, effectively balancing speed and space considerations. Another approach [34] introduces a novel representation of first-class functions called Imperative Functional

Objects (IFOs). IFOs are abstract classes featuring argument and result fields, along with an apply function. The core of TCO in this context revolves around an auxiliary structure that tracks the next call to be executed, enabling delayed application of methods during tail calls and minimizing memory allocation overhead for each tail call. A continuous work [35] proposes a thread-safe variant of IFO for full tail call elimination on the JVM.

Previous approaches to achieving TCO in JavaScript environments, such as babel-plugin [36] and ClojureScript [37], typically transform tail recursive functions into loops to optimize them, rather than relying on the JavaScript engine's built-in capabilities. Additionally, Gambit-JS [38] offers a technique for compiling Scheme's tail calls and first-class continuations to JavaScript. In this approach, a trampolene function is utilized to control the flow of execution. This trampolene is a higher-order function that repeatedly invokes the function it receives as an argument until a specific condition is met. When a tail call occurs, the next function to be executed is returned instead of making a typical JavaScript function call. Subsequently, the trampolene function invokes this returned function, bypassing the calling function itself. In contrast, our TCO method directly jumps to the target function's instruction when a tail call is invoked, without the need for an intermediary trampolene function.

## VI. CONCLUSION

This research has effectively demonstrated the feasibility of an innovative TCO strategy tailored for optimal use of the native stack in JavaScript engines. We are pleased to report that our proposed TCO solution has been successfully integrated into the Escargot engine's main branch, marking a significant milestone towards optimizing JavaScript execution.

While our TCO implementation has been specifically developed for JavaScript environments, we firmly believe that this methodology can be adapted and applied to various other language runtimes as well. Since TCO represents a generalized optimization strategy that positively impacts both performance and memory usage, incorporating it into programming languages such as Python and Java—alongside native stack utilization—could lead to significant enhancements in their efficiency and effectiveness.

Looking ahead, we aim to refine our approach by incorporating an advanced heuristic for more precise detection of tail calls. Rather than applying the `tail_call` bytecode directly to all potential tail calls, we suggest a method that utilizes source code analysis to selectively identify and optimize those calls most likely to benefit, thereby unlocking additional avenues for performance improvement. The insights gained from the web-tooling-benchmark underscore the vast potential for TCO within the real-world web applications. By continuing to refine and enhance our TCO methodology, we anticipate not only wider implementation but also notable performance enhancements across a broader array of applications. Furthermore, our research is ongoing,

with a focus on uncovering prevalent coding patterns in actual applications. Our goal is to devise bespoke optimization techniques that cater specifically to these patterns, thus driving more significant and practical performance improvements. Through these efforts, we aspire to contribute further to the efficiency and effectiveness of JavaScript engines, enhancing the web development ecosystem at large.

## ACKNOWLEDGMENT

We greatly appreciate Haesik Jun and Dong-Heon Jung for their invaluable assistance in this research.

## REFERENCES

- [1] S. Tilkov and S. Vinoski, "Node.js: using Javascript to build high-performance network programs," *IEEE Internet Comput.*, vol. 14, no. 6, pp. 80–83, Nov. 2010.
- [2] G. Williams. (2014). *Espruino*. [Online]. Available: <https://www.espruino.com/>
- [3] D. Smilkov, N. Thorat, Y. Assogba, C. Nicholson, N. Kreeger, P. Yu, S. Cai, E. Nielsen, D. Soegel, S. Bileschi, M. Terry, A. Yuan, K. Zhang, S. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. Viegas, and M. M. Wattenberg, "TensorFlow.js: Machine learning for the web and beyond," in *Proc. SysML*, Stanford, CA, USA, 2019, pp. 309–321.
- [4] GitHub. (2023). *The State of Open Source and Rise of AI in 2023: The Most Popular Programming Languages*. [Online]. Available: <https://github.blog/2023-11-08-the-state-of-open-source-and-ai/#the-most-popular-programming-languages>
- [5] ECMA-TC39-Committee. (2023). *ECMAScript 2023 Language Specification*. [Online]. Available: <https://262.ecma-international.org/14.0>
- [6] ECMA-TC39-Committee. (2015). *ECMAScript 2015 Language Specification*. [Online]. Available: <http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf>
- [7] J. Zaytsev. (2024). *ECMAScript Compatibility Tables: ES6 Features*. [Online]. Available: <https://compat-table.github.io/compat-table/es6/>
- [8] B. Terlson and E. Faust. (2016). *Syntactic Tail Calls*. [Online]. Available: <https://tc39.es/proposal-ptc-syntax/>
- [9] Apple. (2024). *WebKitGTK*. [Online]. Available: <https://github.com/WebKit/WebKit>
- [10] Apple. (2024). *Safari Browser*. [Online]. Available: <https://www.apple.com/safari>
- [11] Google. (2019). *Web Tooling Benchmark*. [Online]. Available: <https://github.com/v8/web-tooling-benchmark>
- [12] Samsung. (2024). *Escargot Open-Source JavaScript Engine*. [Online]. Available: <https://github.com/Samsung/escargot>
- [13] Google. (2024). *V8 JavaScript Engine*. [Online]. Available: <https://github.com/v8/v8>
- [14] Apple. (2024). *JavaScriptCore Built-in JavaScript Engine for WebKit*. [Online]. Available: <https://github.com/WebKit/WebKit/tree/main/Source/JavaScriptCore>
- [15] Mozilla. (2024). *SpiderMonkey JavaScript/WebAssembly Engine*. [Online]. Available: <https://spidermonkey.dev/>
- [16] H. Abelson, G. J. Sussman, M. Henz, T. Wrigstad, and J. Sussman, "Building abstractions with functions," in *Structure and Interpretation of Computer Programs*. Cambridge, MA, USA: MIT Press, 2022, pp. 48–68.
- [17] G. Richards, S. Lebesne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proc. PLDI*, Toronto, ON, Canada, 2010, pp. 1–12.
- [18] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *Proc. ACSAC*, Austin, TX, USA, 2010, pp. 31–39.
- [19] S. Maffei, J. C. Mitchell, and A. Taly, "Isolating JavaScript with filters, rewriting, and wrappers," in *Proc. ESORICS*, Saint-Malo, France, 2009, pp. 505–522.
- [20] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Proc. DIMVA*, Milan, Italy, 2009, pp. 88–106.
- [21] F. Meawad, G. Richards, F. Morandat, and J. Vitek, "Eval begone! semi-automated removal of eval from JavaScript programs," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 607–620, Oct. 2012.



- [22] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. ASIA CCS*, Singapore, 2015, pp. 555–566.
- [23] T.-C. Chieh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proc. ICDCS*, Mesa, AZ, USA, 2001, pp. 409–417.
- [24] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. ASIACCS*, Hong Kong, 2011, pp. 40–51.
- [25] S. Vaarala. (2022). *Duktape 2.7.0*. [Online]. Available: <https://github.com/svaarala/duktape>
- [26] Marvell Semiconductor. (2016). *KinomaJS JavaScript Runtime*. [Online]. Available: <https://github.com/Kinoma/kinomajs>
- [27] Moddable. (2023). *Moddable SDK 4.2.1*. [Online]. Available: <https://github.com/Moddable-OpenSource/moddable>
- [28] ECMA-TC39-Committee. (2024). *Official ECMAScript Conformance Test Suite*. [Online]. Available: <https://github.com/tc39/test262>
- [29] Free-Software-Foundation. (2024). *GCC Manual: Optimize-Options*. [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-foptimize-sibling-calls>
- [30] LLVM-Developer-Group. (2024). *The LLVM Target-Independent Code Generator*. [Online]. Available: <https://llvm.org/docs/CodeGenerator.html#tail-call-optimization>
- [31] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proc. PLDI*, Barcelona, Spain, 2017, pp. 185–200.
- [32] A. Rossberg. (2024). *WebAssembly Specification With Tail Calls*. [Online]. Available: <https://webassembly.github.io/tail-call/core/>
- [33] M. Schinz and M. Odersky, "Tail call elimination on the Java virtual machine," *Electron. Notes Theor. Comput. Sci.*, vol. 59, no. 1, pp. 158–171, Nov. 2001.
- [34] T. Tauber, X. Bi, Z. Shi, W. Zhang, H. Li, Z. Zhang, and B. C. Oliveira, "Memory-efficient tail calls in the JVM with imperative functional objects," in *Proc. APLAS*, Pohang, South Korea, 2015, pp. 11–28.
- [35] M. Madsen, R. Zarifi, and O. Lhoták, "Tail call elimination and data representation for functional languages on the Java virtual machine," in *Proc. CC*, Vienna, Austria, 2018, pp. 139–150.
- [36] K. Kaczor. (2018). *babel-Plugin-Tailcall-Optimization*. [Online]. Available: <https://github.com/krzkaczor/babel-plugin-tailcall-optimization>
- [37] R. Hickey. (2022). *ClojureScript: Clojure to JS Compiler*. [Online]. Available: <https://github.com/clojure/clojurescript>
- [38] E. Thivierge and M. Feeley, "Efficient compilation of tail calls and continuations to JavaScript," in *Proc. Scheme*, Copenhagen, Denmark, 2012, pp. 47–57.



**HYUKWOO PARK** received the Ph.D. degree in electrical engineering and computer science from Seoul National University, Seoul, Republic of Korea, in 2018. Since 2018, he has been a Software Engineer with Samsung Research, which serves as an advanced research and development hub for Samsung's SET (end-products) business. His research interests include web platform optimization with a focus on developing and optimizing lightweight web engines like JavaScript runtimes and WebAssembly engines.



**SEONGHYUN KIM** received the B.S. degree in electronics and communications engineering from Kwangwoon University, Seoul, Republic of Korea, in 2013. Since 2013, he has been a Software Engineer with Samsung Research where he contributes to the organization's primary objectives, including leading advanced research efforts, developing key core technologies, and acquiring critical technologies via open innovation methods. His current research interests include diverse software architectural domains, with a specific focus on designing and implementing robust web platform components, such as web browsers and web programming languages.

• • •