HW-SW CO-DESIGN TECHNIQUES FOR MODERN PROGRAMMING LANGUAGES

BY

JIHO CHOI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Professor Wen-mei Hwu
Professor David Padua
Professor Grigore Roşu
Dr. Călin Caşcaval, Barefoot Networks

ABSTRACT

Modern programming languages raise the level of abstraction, hide the details of computer systems from programmers, and provide many convenient features. Such strong abstraction from the details of computer systems with runtime support of many convenient features increases the productivity of programmers.

Such benefits, however, come with performance overheads. First, many of modern programming languages use a dynamic type system which incurs overheads of profiling program execution and generating specialized codes in the middle of execution. Second, such specialized codes constantly add overheads of dynamic type checks. Third, most of modern programming languages use automatic memory management which incurs memory overheads due to metadata and delayed reclamation as well as execution time overheads due to garbage collection operations.

This thesis makes three contributions to address the overheads of modern programming languages. First, it describes the enhancements to the compiler of dynamic scripting languages necessary to enable sharing of compilation results across executions. These compilers have been developed with little consideration for reusing optimization efforts across executions since it is considered difficult due to dynamic nature of the languages. As a first step toward enabling the reuse of compilation results of dynamic scripting languages, it focuses on inline caching (IC) which is one of the fundamental optimization techniques for dynamic type systems. Second, it describes a HW-SW co-design technique to further improve IC operations. While the first proposal focuses on expensive IC miss handling during JavaScript initialization, the second proposal accelerates IC hit operations to improve the overall performance. Lastly, it describes how to exploit common sharing patterns of programs to reduce overheads of reference counting for garbage collection. It minimizes atomic operations in reference counting by biasing each object to a specific thread.

To my parents, for their love and support.
And to my wife, Heejeong, for everything.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Modern programming languages raise the level of abstraction, hide the details of computer systems from programmers, and provide many convenient features. Such strong abstraction from the details of computer systems with runtime support of many convenient features increases the productivity of programmers. As a result, modern programming languages are widely used by programmers in many application domains, both for clients [1, 2] and servers [3, 4, 5, 6].

Such benefits, however, come with performance overheads. First, many of modern programming languages use a dynamic type system which incurs overheads of profiling program execution and generating specialized codes in the middle of execution. Second, such specialized codes constantly add overheads of dynamic type checks. Third, most of modern programming languages use automatic memory management which incurs memory overheads due to metadata and delayed reclamation as well as execution time overheads due to garbage collection operations.

In this thesis, I address the overheads of dynamic type system and garbage collection in modern programming languages. The first two proposals analyze the key optimization technique for dynamic type system, *Inline Caching (IC)*, and use software and architectural techniques to tackle its two common operations, *IC miss* and *IC hit*, respectively. In the third proposal, I propose a novel algorithm for reference counting to reduce the overheads of garbage collection by minimizing the use of atomic operations.

1.1 CONTRIBUTIONS

1.1.1 Reusable Inline Caching for Dynamic Scripting Languages

JavaScript compilers have been developed with little consideration for reusing compilation and optimization efforts across executions (e.g., upon revisiting a website or visiting another website using the same JavaScript library). Currently, the only compilation step which can be saved by recycling information across executions is the initial bytecode generation from the source code. The rest of compilation steps, such as profiling program behavior and generating optimized codes, have to be repeated in every execution. The outputs from those steps are highly dependent on the execution context and deemed to be difficult to share across executions.

This paper analyzes the key optimization technique for dynamic type system, *Inline*

Caching (IC), to show that the majority of IC information collected during JavaScript initialization (e.g., webpage loading) can be represented in a context-independent way and shared across executions. Based on this observation, it proposes an enhanced design of IC called *Reusable Inline Caching (RIC)*. RIC extracts the context-independent portion of IC information after the first execution, and this information is used to significantly improve IC operations in future executions. We evaluate RIC with the state-of-the-art Google V8 JavaScript engine by measuring its impact on the initialization time of popular JavaScript libraries. On average, by recycling IC information collected from the previous execution, RIC reduces the number of IC misses by 53%, the dynamic instruction count by 15%, and finally the execution time by 17%.

1.1.2 ShortCut: Architectural Support for Fast Object Access in Scripting Languages

The same flexibility that makes dynamic scripting languages appealing to programmers is also the primary cause of their low performance. To access objects of potentially different types, the compiler creates a dispatcher with a series of if statements, each performing a comparison to a type and a jump to a handler. This induces major overhead in instructions executed and branches mispredicted.

This work proposes architectural support to significantly improve the efficiency of accesses to objects. The idea is to modify the instruction that calls the dispatcher so that, under most conditions, it skips most of the branches and instructions needed to reach the correct handler, and sometimes even the execution of the handler itself. Our novel architecture, called *ShortCut*, performs two levels of optimization. Its *Plain* design transforms the call to the dispatcher into a call to the correct handler — bypassing the whole dispatcher execution. Its *Aggressive* design transforms the call to the dispatcher into a simple load or store — bypassing the execution of both dispatcher and handler. We implement the ShortCut software in the state-of-the-art Google V8 JIT compiler, and the ShortCut hardware in a simulator. We evaluate ShortCut with the Octane and SunSpider JavaScript application suites. Plain ShortCut reduces the average execution time of the applications by 30% running under the baseline compiler, and by 11% running under the maximum level of compiler optimization. Aggressive ShortCut performs only slightly better.

1.1.3 Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection

Reference counting (RC) is one of the two fundamental approaches to garbage collection. It has the desirable characteristics of low memory overhead and short pause times, which

are key in today's interactive mobile platforms. However, RC has a higher execution time overhead than its counterpart, tracing garbage collection. The reason is that RC implementations maintain per-object counters, which must be continually updated. In particular, the execution time overhead is high in environments where low memory overhead is critical and, therefore, non-deferred RC is used. This is because the counter updates need to be performed atomically.

To address this problem, this paper proposes a novel algorithm called *Biased Reference Counting* (BRC), which significantly improves the performance of non-deferred RC. BRC is based on the observation that most objects are only accessed by a single thread, which allows most RC operations to be performed non-atomically. BRC leverages this by biasing each object towards a specific thread, and keeping two counters for each object — one updated by the owner thread and another updated by the other threads. This allows the owner thread to perform RC operations non-atomically, while the other threads update the second counter atomically.

We implement BRC in the Swift programming language runtime, and evaluate it with client and server programs. We find that BRC makes each RC operation more than twice faster in the common case. As a result, BRC reduces the average execution time of client programs by 22.5%, and boosts the average throughput of server programs by 7.3%.

CHAPTER 2: REUSABLE INLINE CACHING FOR DYNAMIC SCRIPTING LANGUAGES

2.1 INTRODUCTION

Dynamic scripting languages, such as JavaScript, Python and PHP, gain popularity because of their ease of use enabled by many convenient runtime supports including dynamic type systems, built-in objects, and automatic memory management. Such features increases the programmer productivity but comes with performance overheads.

To overcome such performance overheads, the industry and developer communities have put lots of efforts on improving the performance of compilers and runtime systems, such as Google V8 JavaScript engine [7] and HHVM [8]. These compilers, however, have been developed with little consideration for reusing compilation results across executions. Dynamic nature of these languages makes it very challenging for efficient code generation, and the optimized codes and profiling data end up closely tied to a specific execution context. As a result, in the current implementations, the only compilation step which can be saved by recycling information across executions is the initial bytecode generation from the source code. The rest of compilation steps, such as profiling program behavior and generating optimized codes, have to be repeated from scratch in every execution. This remains as an unsolved problem for the compilers of dynamic scripting languages.

At the same time, programs keep growing in size and complexity, and yet the user expectation is ever increasing as well as shown in Figure 2.1. The average number of JavaScript requests in the top 1000 websites have gone up from 12 in 2010 to 28 in 2015. Similarly, the total transfer size of JavaScript files has also increased sharply from 103 KB to 404 KB during the same period. Considering that initialization of each JavaScript library takes tens to hundreds of milliseconds [9], it puts much pressure on meeting ever increasing user expectation. According to a series of user surveys [10, 11, 12, 13], while the majority of users waited up to eight seconds for page load in 1999, this wait time had shrunk to two seconds by 2014. Furthermore, the industry is pushing the bar even higher by advocating a sub one-second limit so as to not interrupt a user's flow of thoughts [14]. Note that JavaScript initialization is only a part of works to be done for completing page load, and a sub one-second page load is especially challenging in mobile devices with slow network connections and limited computing resources.

We see a lack of sharing compilation efforts between executions as a missing piece to improve JavaScript performance to get closer to a sub one-second page load. There are many optimization techniques implemented in the current JavaScript engines. Among them, *inline*

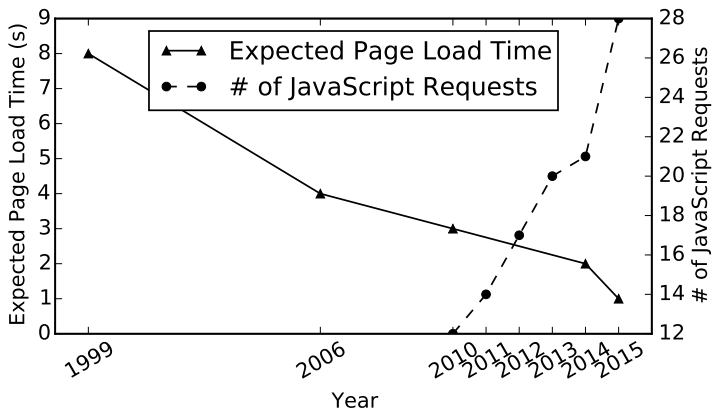


Figure 2.1: Contradictory trends of user expectation for page load time and webpage complexity.

caching (IC) is a fundamental technique to reduce the overheads of dynamic type systems based on an empirical evidence that the same type tends to be repeatedly encountered at the same program location. Similar to other optimization techniques for dynamic scripting languages, profiling data and generated codes for IC are tied to a specific execution context and difficult to reuse across executions.

As the first step toward reusing optimization efforts across executions, this paper proposes a new IC design called *Reusable Inline Caching (RIC)* which enables information sharing across executions. RIC extracts the context-independent portion of IC information after the first execution, and this information is used to significantly improve IC operations in future executions.

We evaluate RIC with the state-of-the-art Google V8 JavaScript engine by measuring its impact on the initialization time of popular JavaScript libraries. On average, by recycling IC information collected from the previous execution, RIC reduces the number of IC misses by 53%, the dynamic instruction count by 15%, and finally the execution time by 17%.

The contributions of this paper are as follows:

- Characterizes the overheads of IC during initialization of popular JavaScript libraries.
- Characterizes IC operations during JavaScript initialization to identify opportunities to share IC information across executions.
- Proposes RIC, a new IC design to enable the reuse of IC information across executions by extracting the context-independent portion of IC information.
- Implements RIC in the state-of-the-art Google V8 JavaScript engine.
- Provides a detailed evaluation of RIC's performance.

2.2 BACKGROUND

2.2.1 JavaScript Execution

While JavaScript standalone is as powerful as other dynamic scripting languages for various tasks, it was originally developed for browsers to provide application logic and user interactivity. As such, a JavaScript compiler has evolved to provide APIs well-suited for event-driven execution, and it is now widely used as an embedded language runtime in many systems including browsers, server-side framework [3], operating systems [2], etc.

The host system, in which a JavaScript runtime is embedded, is responsible for scheduling JavaScript execution. It triggers JavaScript execution mainly in two different ways: *Initialization* and *Event Handling*. Initialization happens when the host system first loads a JavaScript source code into the JavaScript runtime. For example, when a browser encounters a script tag during HTML parsing, it passes a JavaScript source code in the tag to the JavaScript runtime. Then, the runtime compiles the source code and executes statements in the global scope.

In this paper we focus on JavaScript initialization, as it directly affects page load time performance. Until all included JavaScript source codes are initialized, the page cannot be fully loaded. For example, the initialization can block DOM object construction and page rendering. In addition, typically user interaction is disabled until initialization is complete.

Both during and after initialization, the host system can also register JavaScript functions to handle events, such as user input. In contrast to initialization, JavaScript execution for event handling is highly dependent on user interactions. Hence, due to its unpredictable behavior, improving the performance of event-handling is not addressed in this paper and is left as future work.

2.2.2 V8 JavaScript Engine

V8 [7] is the JavaScript JIT compiler used in Google Chrome web browser. In the following, we describe the concepts in V8 related to our work. These concepts are not specific to V8, however, and are used in all popular JavaScript compilers.

Compilation Strategy Since there is no standard bytecode for JavaScript and every JavaScript compiler generates code differently, a JavaScript program is delivered in source code format. Therefore, the program has to be parsed and compiled before starting execution. It introduces significant execution time overheads and takes about 30% of total

execution time in real-world websites [15].

The early JavaScript compilers implemented *Eager Compilation* which compiles the entire program before beginning execution. However, since only a small fraction of the program is usually executed, eager compilation unnecessarily delays the start of execution by wasting time on compiling unused functions [16]. Furthermore, it increases memory pressure by generating code objects for functions that are ultimately never executed.

In order to accelerate the startup of execution, every major JavaScript compiler currently implements *Lazy Compilation*, which only compiles statements in the global scope and delays compilation of other functions until they are called during execution. As a result, it not only minimizes the delay caused by compilation before the first execution, but also improves the overall execution time and memory usage by only compiling necessary functions. Compared to eager compilation, lazy compilation introduces runtime overhead to schedule compilation dynamically, but such overhead is well justified in real-world programs where only a small fraction of functions are ever called [16].

Code Caching Although JavaScript files are delivered in source code format and have to be compiled before the first execution, some intermediate results are deterministic and may be cached for the later executions. For example, parsing the same source code produces the same AST, and then it is compiled into the same bytecodes. To avoid such redundant compilation across executions, V8 supports APIs to enable embedders to access some compilation results [17]. Using these APIs, Chrome browser maintains code cache containing compiled codes for frequently executed JavaScript files. Previously, in conjunction with lazy compilation, these APIs provided bytecodes for top-level codes only (i.e., statements in global scope). They have been improved to increase the coverage by also providing bytecodes for functions which are lazily compiled and executed [18]. With the increased coverage, code caching essentially eliminates the overheads of bytecode generation during JavaScript initialization.

While code caching accelerates JavaScript initialization, it has limitations as it can only store the initial compilation output, such as bytecode, which is deterministic and context-independent. First, it doesn't support any profiling data collected during the program execution. Such profiling data is critical for efficient bytecode execution and optimized code generation, and the time to collect the profiling data delays JavaScript initialization. Second, it doesn't support optimized codes. V8 profiles the program execution and generates optimized machine codes for frequently executed functions. It is very difficult to cache and reuse these optimized codes since they are tightly coupled to the execution context where they are generated and executed. For example, memory addresses for built-in objects are embedded as immediate operands in the optimized codes.

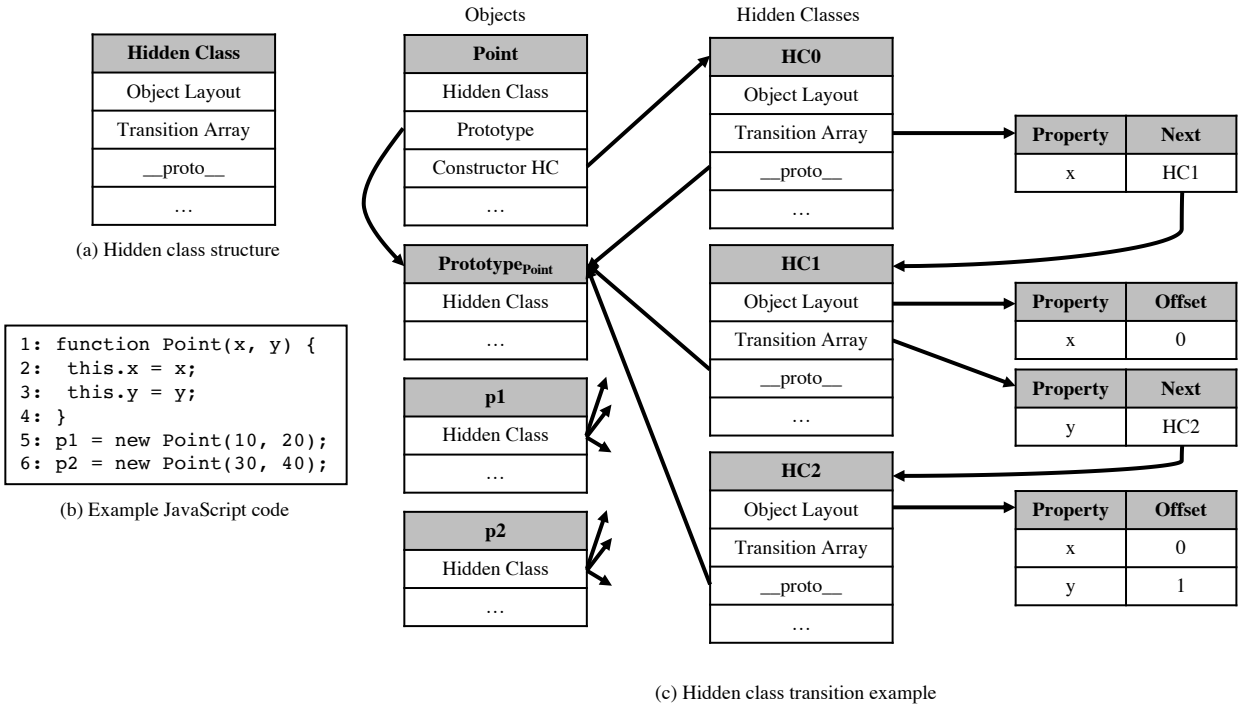


Figure 2.2: Hidden class structure and example of hidden class transition

Hidden Classes JavaScript is a dynamically-typed programming language where properties can be added to, and deleted from objects on the fly. Such dynamism prohibits JavaScript compilers from constructing a fixed object layout before execution. Type information, however, is crucial to efficient code generation, and V8 dynamically creates *Hidden Classes* behind the scenes to introduce the notion of types. The basic idea is similar to the maps in Self [19]. Objects that are created in the same way are grouped in the same hidden class. Note that the concept of types itself is an idea imposed by the compiler for the purposes of code generation, and is not present in the semantics of a dynamically-typed language like JavaScript. For this reason, types in dynamically-typed languages are called hidden classes, to stress that the types are hidden from the programmer. We use the terms type and hidden class interchangeably.

Figure 2.2(a) shows the hidden class structure. An object layout points to a table of `(property, offset)` pairs, and it is context-independent as its content is not tied to any specific execution context. On the other hand, the rest of the hidden class fields are context-dependent in that they are tied to JavaScript objects allocated in the heap of JavaScript execution context. A transition array points to an array of next hidden classes to transition into when a new property is added. A `__proto__` points to the prototype object associated with the constructor function. Note that JavaScript uses prototyping to emulate the class

inheritance of statically-typed languages such as C++ or Java. Every object in JavaScript inherits from a *parent* object, which is the prototype object of the constructor function used to create that object. Inheritance is supported by having the special property `__proto__` in the object pointing to the prototype object at object construction time. The prototype object is inherited by all the objects that are created using the corresponding constructor. Properties in the prototype can either be accessed via inheritance through the *child* object, or directly through the prototype property in the constructor function object. In addition, there are several other fields which are also context-dependent, and since a hidden class contains context-dependent fields, a hidden class itself becomes context-dependent as well.

Hidden classes are mainly created in two different ways in the program. First, there is a hidden class created for each function. When a function is used as a constructor for an object, the hidden class associated with the function (`Constructor HC` in Figure 2.2(c)) is assigned to the newly allocated object. For example, when the function `Point` is declared on line 1 of Figure 2.2(b), a new function object, `Point`, is created along with a new hidden class, `HC0`, and a prototype object, $\texttt{Prototype}_{\texttt{Point}}$, as shown in Figure 2.2(c). On line 5, when the function `Point` is used as a constructor, the newly created object, `p1`, is initially assigned the hidden class, `HC0`. Second, a new hidden class can be created when adding a property to an object triggers a new hidden class transition. For example, after being assigned `HC0` on line 5, the construction of `p1` continues to lines 2-3 with `p1` bound to this, and a new property `x` is added to `p1` on line 2. Initially, the transition array of `HC0` is empty, and the runtime creates a new hidden class, `HC1`, with the object layout with the new property `x` and updates `HC0`'s transition array. Similarly, when the property `y` is added on line 3, the new hidden class, `HC2`, is created and `HC1`'s transition array is updated. During the object construction, `p1`'s hidden class is initially set to `HC0` on line 5 and updated to `HC1` on line 2 and finally to `HC2` on line 3. Note that a new hidden class is created for a new transition only. When `p2` is constructed on line 6, it goes through lines 2-3 but does not create any new hidden classes because the transition arrays of `HC0` and `HC1` already have necessary transitions.

Inline Caching One of the fundamental optimization techniques enabled by type information collected through hidden classes is inline caching [20]. It is based on the empirical evidence that the objects received at a particular object access site are often of the same type.

V8 generates a specialized handler for each observed hidden class and records `(hidden class, handler)` pairs in `ICVector` (`TypeFeedbackVector` in V8 source code). An `ICVector` is maintained per function, and each object access site in the function has a dedicated slot

```
1  class ICVectorSlot {
2      HiddenClassAddr incoming_hc_addr;
3      HiddenClassAddr transition_hc_addr;
4      HandlerAddr handler_addr;
5  };
6
7  class ICVector {
8    public:
9      ...
10
11   private:
12     map<ObjectAccessSiteID, vector<ICVectorSlot>> vector;
13     ...
14 };
```

Figure 2.3: `ICVector`'s data structures

in the vector. When executing an object access site, it first compares the hidden class of the incoming object with the hidden classes recorded in the `ICVector`. If there is a match, it is an IC hit and jumps to the recorded handler. Otherwise, it has to jump to the runtime to search the object layout of the hidden class of the incoming object, generate a specialized handler, and update the `ICVector` accordingly for the future execution.

Figure 2.3 shows the details of the `ICVector`. It maps each object access site to a list of `ICVectorSlot`. Initially, the object access site is unitinitialized and its list is empty. After the first IC miss at the object access site, a new `ICVectorSlot` is appended to the list and the object access site becomes monomorphic. If a different hidden class is encountered at the object access site later, it causes an IC miss to append another `ICVectorSlot` to the list and make the object access site polymorphic.

There are mainly two different kinds of object access sites depending on the kind of memory access performed (i.e., whether it is load or store). Object access sites at the line 2 and 3 in Figure 2.2(b) both perform store operations. Furthermore, these object access sites updates the hidden class of the incoming object. We call such object access sites as *Transitioning Object Access Site*. When the program first executes a transitioning object access site, it will be an IC miss and the execution jumps to the runtime to look up the transition array of the incoming hidden class to figure out the hidden class to transition into. If the lookup fails, the runtime creates a new hidden class. `tranisition_hc_addr` (line 3 in Figure 2.2) is set for transitioning object access sites only.

Handlers are of various complexity depending on the object access site type (i.e., whether it is load or store) and the object lookup result. Some handlers are very simple and context-independent. For example, a handler could be as simple as a load instruction from a fixed offset. On the other hand, some handlers perform more complicated operations, such as

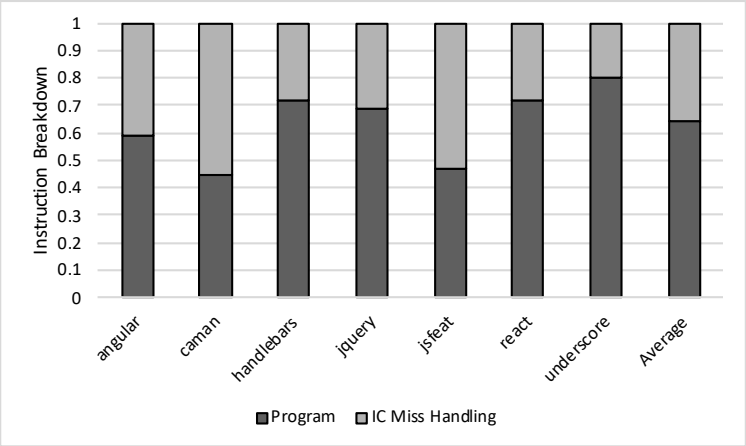


Figure 2.4: Overhead of IC miss handling in JavaScript libraries.

traversing the prototype chain or updating the incoming object's hidden class, and are tied to the execution context.

2.3 MOTIVATION

Overheads of IC To assess the overheads of IC, we count the number of instructions spent on handling IC misses during initialization of popular JavaScript libraries. As shown in Figure 2.4, IC miss handling incurs significant performance overheads during JavaScript initialization. On average, 36% of instructions are spent on handling IC misses in the runtime looking up the incoming object's object layout to find the requested property, generating a specialized handler, and updating the `ICVector` for the future execution.

Opportunities for Reuse To estimate the potential for reusing IC information between executions, we instrument V8 JavaScript engine to collect various statistics related to IC as shown in Table 2.1. The first column shows the number of hidden classes encountered during initializing the evaluated JavaScript libraries. The second column shows the number of IC misses. In all benchmarks, the number of IC misses is much higher than the number of hidden classes. The third column shows the ratio of the first two columns. It shows that the same hidden class is encountered in several different object access sites. On average, each hidden class is shared in about 5 IC misses. The last column shows the ratio of handlers which are not tied to a specific execution context so that they can be shared across executions. On average, 60% of handlers generated in our benchmarks are reusable.

To summarize, while IC miss handling adds significant performance overheads for JavaScript initialization, our characterization results show that there exist opportunities to avoid the

Benchmark	# of Hidden Classes	# of IC Misses	# of IC Misses per HC	% of Reusable Handlers
angular	138	799	5.79	62.50
caman	99	383	3.87	61.84
handlebars	88	6.15	541	63.20
jquery	271	1547	5.71	57.31
jsfeat	116	323	2.78	51.72
react	360	2356	6.54	82.27
underscore	123	295	2.40	38.12
Average	171	892	4.75	59.57

Table 2.1: Statistic related to IC during JavaScript initialization.

overheads by reusing IC information across executions. First, the fact that the hidden classes are shared in multiple IC misses opens up an opportunity to handle multiple IC misses at the same time. Second, the fact that the majority of the handlers are context-independent suggests that a large portion of IC information can be easily shared across executions.

2.4 REUSABLE INLINE CACHING

2.4.1 Main Idea

The goal of this paper is to reduce the overhead of IC miss handling by reusing the information passed from the previous execution. We do this with an enhanced IC design that we call *Reusable Inline Caching (RIC)*. RIC leverages the observation that each hidden class appears in multiple IC misses and many specialized handlers are context-independent. Hence, RIC records the groups of object access sites encountering the same hidden class in the previous execution (i.e., *record run*) so that they can be handled altogether in the future execution (i.e., *replay run*) combining each group's multiple IC misses into one.

RIC allows the reuse of IC information across executions by maintaining the information in a context-independent manner. First, RIC does not attempt to record hidden classes. Hidden classes are context-dependent since they point to prototype objects which are tied to a specific execution context as explained in Section 2.2.2. Instead, RIC records where a new hidden class is created in the program (e.g., when adding a new property to an object) and at which object sites in the program this hidden class is encountered. Second, RIC only deals with context-independent handlers.

RIC does not introduce any additional overheads to the record run as the information is

```
1  struct TOASTEntry {
2      HiddenClassID incoming_hc_id;
3      HiddenClassID transition_hc_id;
4  };
5
6  typedef map<ObjectAccessSiteID, TOASTEntry> TOAST;
7
8  struct HCGTEntry {
9      bool validated;
10     set<pair<ObjectAccessSiteID, HandlerAddr>> dependents;
11 }
12
13 typedef map<HiddenClassID, HCGEntry> HCGT;
14
15 class ICRecord {
16   public:
17     void Add(ObjectAccessSiteID site, HiddenClassID incoming_hc_id,
18              HiddenClassID transition_hc_id, HandlerAddr handler);
19
20     void Restore();
21
22   private:
23     HCGT hcgt;
24     TOAST toast;
25 };
```

Figure 2.5: `ICRecord`'s data structures

generated in background after the execution finishes. All necessary information is already collected during the execution in the `ICVector`. RIC scans the `ICVector` to generate the *IC Record* which contains three pieces of information: (i) at which object site each hidden class is created, (ii) which object sites depend on each hidden class, and (iii) context-independent handlers generated for each object site and hidden class pair.

In the replay run, RIC validates the IC record as it creates a hidden class. A hidden class is created at the transitioning object site when adding a property to an object triggers a new hidden class transition as explained in Section 2.2.2. If the IC record contains the transition at the object site and the incoming hidden class is already validated, RIC validates the transition hidden class. Finally, RIC looks up the IC record to find out all the object sites using the newly validated hidden class and updates the `ICVector` for those object sites to avoid IC misses.

2.4.2 IC Record

The `ICRecord` consists of two data structures, *Transitioning Object Access Site Table (TOAST)* and *Hidden Class Group Table (HCGT)*. As its name suggests, the `TOAST` contains

the table of transitioning object access sites which cause a new hidden class created upon their first execution. The `HCGT` maps each hidden class to a group of object access sites encountering the hidden class.

Figure 2.5 shows the details of `TOAST` and `HCGT`. The `TOAST` maps each transitioning object access site to an entry containing a pair of incoming hidden class and transition hidden class. The `HCGT` maps each hidden class to an entry containing a flag whether this hidden class is validated and a set of object access sites dependent on this hidden class and their context-independent handlers.

Note that `ICRecord` distinguishes hidden classes with context-independent integer IDs. On the other hand, `ICVector` records hidden classes using their addresses as shown in Figure 2.3.

Generating `ICRecord` after Record Run The `ICRecord` is generated from `ICVector` after the execution finishes. Figure 2.6 shows the methods for generating `ICRecord` after the record run. `GenerateICRecord` (lines 34-52) is invoked after the execution finishes to iterate the `ICVector` (lines 37-50) to extract the IC information in a context-independent manner. For each `ICVectorSlot`, it first converts hidden class addresses to integer IDs (lines 41 and 42) using `HiddenClassRecordMap` (lines 12-32). Then, if the handler is context-independent, it adds this `ICVectorSlot`'s information to the `ICRecord` (lines 46-49). When adding information to the `ICRecord`, it first checks if it is for transitioning object access site (line 3), and if so, it adds the information to the `TOAST`. Otherwise, it updates the `HCGT`.

Utilizing `ICRecord` in Replay Run In the replay run, the `ICRecord` passed from the previous execution is utilized to handle multiple IC misses at once. Figure 2.7 shows the methods for utilizing `ICRecord` in the replay run. The runtime function to handle IC miss handling, `HandleICMiss` (lines 22), is extended to use these methods to update `ICVector` for other object access sites dependent on the transition hidden class. `HandleICMiss` retrieves a transition hidden class (line 27) and invokes `ICRecord::Restore` to update `ICVector` according to `ICRecord` (line 29). `ICRecord::Restore` first checks if the current object access site has information about transition in `ICRecord`'s `TOAST` (line 10). If so, it reads the hidden class IDs from `TOAST` (lines 11-12). If the incoming hidden class ID is not validated, it returns without validating the transition hidden class (line 13). It means that the execution takes a different path than the record run and `ICRecord` for this object access site might be wrong. If the incoming hidden class ID is already validated, the transition hidden class ID is validated (line 14) and it iterates `HCGT` (line 15) to update `ICVector` of object access sites dependent on the transition hidden class (line 18). This saves IC misses at those object access sites.

```
1  void ICRecord::Add(ObjectAccessSiteID site, HiddenClassID incoming_hc_id,
2                     HiddenClassID transition_hc_id, HandlerAddr handler) {
3      if (transition_hc_id != 0) {
4          toast[site] = TOASTEntry(incoming_hc_id, transition_hc_id);
5      } else {
6          if (hcgt.count(incoming_hc_id) == 0)
7              hcgt[incoming_hc_id] = entry(false, set());
8          hcgt[incoming_hc_id].dependents.insert(make_pair(site, handler));
9      }
10 }
11
12 class HiddenClassRecordMap {
13   public:
14     HiddenClassID GetID(HiddenClassAddr addr) {
15         if (addr == nullptr) return 0;
16         else if (Exist(addr)) return hc_map(addr);
17         else return Add(addr);
18     }
19
20   private:
21     bool Exist(HiddenClassAddr addr) {
22         return (hc_map.count(addr) > 0);
23     }
24
25     HiddenClassID Add(HiddenClassAddr addr) {
26         HiddenClassID new_id = hc_map.size() + 1;
27         hc_map[addr] = new_id;
28         return new_id;
29     }
30
31     map<HiddenClassAddr, HiddenClassID> hc_map;
32 };
33
34 void ICVector::GenerateICRecord() {
35     ICRecord ic_record;
36     HiddenClassMap hc_map;
37     for (auto& kvp in vector) {
38         ObjectAccessSiteID site = kvp.first;
39         vector<ICVectorSlot> slot_vector = kvp.second;
40         for (auto& slot in slot_vector) {
41             HiddenClassAddr incoming_hc_addr = slot.incoming_hc_addr;
42             HiddenClassAddr transition_hc_addr = slot.transition_hc_addr;
43             HandlerAddr handler = slot.handler;
44             HiddenClassID incoming_hc_id = hc_map.GetID(incoming_hc_addr);
45             HiddenClassID transition_hc_id = hc_map.GetID(transition_hc_addr);
46             if (IsContextIndependent(handler))
47                 ic_record.Add(site, incoming_hc_id,
48                               transition_hc_id, handler);
49         }
50     }
51     return ic_record;
52 }
```

Figure 2.6: Methods for generating `ICRecord` in the record run

```
1  void ICVector::Add(ObjectAccessSiteID site,
2                     HiddenClassAddr incmoing_hc_addr, HandlerAddr handler) {
3      vector[site].push_back(ICVectorSlot(incoming_hc_addr, nullptr, handler));
4  }
5
6  void ICRecord::Restore(ObjectAccessSiteID site,
7                         HiddenClassAddr incoming_hc_addr,
8                         HiddenClassAddr transition_hc_addr,
9                         ICVector* ic_vector) {
10     if (toast.count(site) == 0) return;
11     HiddenClassID incoming_hc_id = toast[site].incoming_hc_id;
12     HiddenClassID transition_hc_id = toast[site].transition_hc_id;
13     if (!hcgt[incoming_hc_id].validated) return;
14     hcgt[transition_hc_id].validated = true;
15     for (auto& p in hcgt[transition_hc_id].dependents) {
16         ObjectAccessSiteID dependent_site = p.first;
17         HandlerAddr handler = p.second;
18         ic_vector->Add(dependent_site, transition_hc_addr, handler);
19     }
20 }
21
22 void Runtime::HandleICMiss(ObjectAccessSiteID site,
23                            Object* incoming_obj, PropertyName name) {
24     ...
25     HiddenClassAddr incoming_hc_addr = incoming_obj->hc_addr;
26     HiddenClassAddr transition_hc_addr
27                         = GetTransitionHiddenClass(incoming_hc_addr, name);
28     ...
29     ic_record->Restore(site, incoming_hc_addr, transition_hc_addr,
30                         ICVector* ic_vector);
31     ...
32 }
```

Figure 2.7: Methods for utilizing `ICRecord` in the replay run

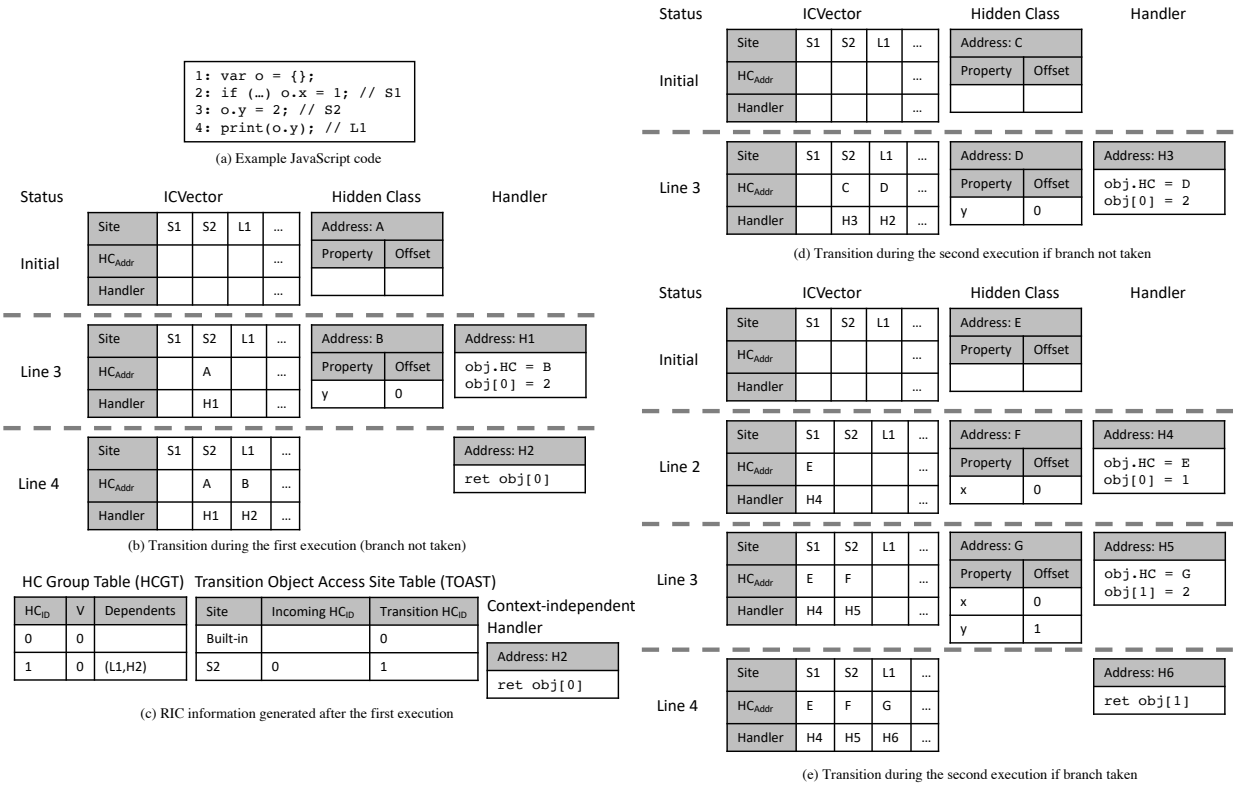


Figure 2.8: Example walk-through of how RIC extracts and utilizes context-independent portion of IC information.

2.4.3 RIC Examples

In this section, we show an example about how IC information is originally generated during execution, how RIC extracts context-independent portion of the IC information, and finally how RIC utilizes it to avoid IC misses while ensuring correctness.

Figure 2.8(a) shows a JavaScript source code example which creates an empty object (line 1), adds a property `x` if a branch is taken, adds a property `y`, and finally prints out the value of the property `y` which will be always 2 in this simple example. There are 3 object access sites in the example. Lines 2 and 3 have object access sites which adds a property and stores the value to it which we will call as `S1` and `S2`, respectively. Also, line 4 has an object access site which loads the value of the property which we will call as `L1`.

Figure 2.8(b) shows how `ICVector` is updated and which hidden classes and handlers are created as the example code in Figure 2.8(a) is executed in the baseline. `ICVector` has slots for three object access sites in the code (`S1`, `S2` and `L1`) and is initially empty. The only relevant built-in hidden class is an empty hidden class and its hidden class object is located at the memory address `A`. We assume that the branch on line 2 is not taken so that `S1` is not

accessed. On line 3, since ICVector is empty, it is an IC miss on S2 and the runtime creates a new hidden class with the property y at the offset 0 and a new handler which makes a hidden class transition to the new hidden class and stores the value to the offset 0 in the incoming object. ICVector is updated with the incoming hidden class and the new handler. On line 4, it is an IC miss on L1. This time, there is no need to create a new hidden class. The runtime creates a new handler loading the value from the offset 0 and updates ICVector with the incoming hidden class and the new handler.

Figure 2.8(c) shows how RIC extracts the context-independent portion of IC information. HCGT enumerates hidden classes encountered during the execution, gives them integer IDs, and records object access sites and corresponding context-independent handlers dependent on each hidden class. In this example, there are 2 hidden classes observed and the only hidden class associated with a context-independent handler is the hidden class B which is created on line 3. HCGT assigns an integer ID 1 to it and records that the handler H2 may be used at the object access site L1 if validated successfully. TOAST records where new hidden classes are created and provides information to validate HCGT's information. Lastly, RIC keeps context-independent handlers. In this example, it only keeps the handler H2.

Figure 2.8(d) shows how RIC information is utilized to avoid IC misses when the execution follows the same control flow (branch not taken on line 2) and RIC information is validated successfully. ICVector is empty initially. The empty built-in hidden class object is located at the memory address C this time. When built-in hidden classes are created at the beginning of execution, it consults with RIC information to validate the related portion and updates ICVector when necessary. In this case, it links the hidden class C to the integer ID 0 according to TOAST and validates the corresponding entry in HCGT. Since it does not have any dependents, it does not update ICVector. The branch on line 2 is not taken so that S1 is not accessed. On line 3, it is an IC miss on S2 and the runtime creates a new hidden class D and a new handler H3 similar to the first execution. Note that this IC miss cannot be avoided because the handler H3 is context-dependent. In addition to updating ICVector for the object S2 with the hidden class C and the handler H3 as in Figure 2.8(b), it checks RIC information. TOAST expects the incoming hidden class to have the integer ID 0 and have it validated. As it is already validated when built-in hidden classes are created, it links the hidden class D to the integer ID 1 according to TOAST and validates the corresponding entry in HCGT. In this case, the entry also has dependents information and it updates ICVector for the object access site L1 with the hidden class D and the handler H2. Thanks to this update, on line 4, it is an IC hit as opposed to an IC miss in Figure 2.8(b).

Figure 2.8(e) shows how RIC's validation scheme ensures the correctness when the execution takes a different control flow (branch taken on line 2). Again, ICVector is empty

initially. The empty built-in hidden class is allocated at the memory address `E` and linked to the integer ID 0 according to `TOAST`. The branch on line 2 is taken this time and the object access site `S1` is accessed. It is an IC miss and the runtime creates a new hidden class and a new handler. However, RIC information lacks any information about `S1` and the newly created hidden class does not validate anything. On line 3, it is an IC miss on the object access site `S2` and the runtime creates a new hidden class and a new handler. Note that the new hidden class `G` has the property `y` at the offset 1 while the hidden classes `B` and `D` have it at the offset 0 in Figure 2.8(b) and Figure 2.8(d), respectively. `TOAST` has an entry for the object access site `S2` but it expects the incoming hidden class to have the integer ID 0 and have it validated. While the integer ID 0 is validated, remember that the integer ID 0 is linked to the hidden class `E` which is different from the incoming hidden class `F`. As a result, we cannot validate the integer ID 1 and it prevents us from updating `ICVector` for the object access site `L1`. Finally, on line 4, unlike in Figure 2.8(d), it is an IC miss on `L1`. Note that we would end up accessing the object with the wrong offset (i.e., 0 instead of 1) if RIC's validation scheme prevents us from updating `ICVector` incorrectly.

2.5 EXPERIMENTAL SETUP

To characterize IC operations and evaluate our proposal, we use the version 6.8 of Google V8 JavaScript engine [7]. We modify the V8 compiler to generate verbose log messages to gather statistics related to IC operations, such as the number of IC misses, hidden class addresses, etc. To simulate the performance impact of our proposal, we instrument the V8 runtime and generated codes to communicate with Pin [21] to report the number of instructions for each IC miss and other parts of execution. We gather the execution time for each part using the high precision timer available in the V8 runtime.

In our experiments, for each benchmark, we first execute it to collect information to be reused in the future executions (i.e., *record run*). Then, we execute it again with the collected information (i.e., *replay run*) to take the actual measurements. We do not introduce any additional overheads to the record run since information is processed in background after the execution finishes.

We test two configurations to measure the performance impact of our techniques. The baseline (B) the original V8 compiler with the code cache enabled. In the record run, B generates bytecodes from the source code and store them in the code cache. In the replay run, bytecode generation is compeltely avoided by reusing bytecodes provided by the code cache. The reusable IC (R) is B with our enhanced IC design. In the record run, in addition to bytecodes, the IC information is collected. In the replay run, bytecode generation is still

skipped thanks to the code cache, and the IC information is utilized by our enhanced IC design to reduce the overhead of IC miss handling.

To evaluate our enhancements, we select 7 popular JavaScript libraries from various domains and measure their initialization performance. AngularJS [22] and React [23] are web application frameworks to support the development of single-page applications. CamanJS [24] is a JavaScript library for image manipulation. Handlebars [25] and Underscore [26] are client-side template engines. jQuery [27] is the most widely used JavaScript library for DOM manipulation. JSFeat [28] is a JavaScript library for computer vision.

We insert a fake window object to the original source codes to mimic the browser environment and run benchmarks in the standalone V8 shell. Octane benchmark suite [29] handles the library execution in a standalone JavaScript shell in a similar way.

Lastly, in order to mimic the real webpages and evaluate the robustness of RIC, we create two synthetic webpages loading the libraries in different orders. We use the first one for the record run to generate RIC information. Then, we use the second one for the replay run where we measure the performance. This effectively simulates the common scenario when RIC information is generated and utilized in different webpages. With this setup, we disable RIC for global objects since IC information for global objects varies for each website depending on the order in which the libraries are loaded.

2.6 EVALUATION

In this section, we analyze various statistics related to IC opeartions to study the effectiveness of RIC, and compare dynamic instruction count and execution time in the configurations B and R to evaluate the performance impact of RIC.

2.6.1 IC Statistics

Table 2.2 shows the statistics related to RIC. The first four columns are repeated from Table 2.1 for the explanation. The third column shows the ratio between the number of hidden classes and the number of IC misses. On average, 4.75 IC misses share the same hidden class. It means that RIC can potentially reduce the number of IC misses by the factor of up to 4.75 (i.e., by about 80%). The fourth column shows the ratio of reusable handlers. Reusable handlers are not tied to a specific execution context and easily transferable across executions. On average, 60% of handlers are resuable which means that RIC can be applied to the majority of handlers.

Benchmark	# of Hidden Classes	# of IC Misses	# of IC Misses per HC	% of Reusable Handlers	ICRecord Stats			
					% of Hits	% of Misses		
						Handler	Global	Other
angular	138	799	5.79	62.50	52.44	12.52	4.13	30.91
caman	99	383	3.87	61.84	49.87	1.31	3.92	44.91
handlebars	88	541	6.15	63.20	64.88	8.32	1.85	24.95
jquery	271	1547	5.71	57.31	39.63	13.38	2.33	44.67
jsfeat	116	323	2.78	51.72	56.97	0.93	9.60	32.51
react	360	2356	6.54	82.27	79.50	10.19	1.66	8.66
underscore	123	295	2.40	38.12	30.85	3.39	4.07	61.69
Average	171	892	4.75	59.57	53.45	7.15	3.93	35.47

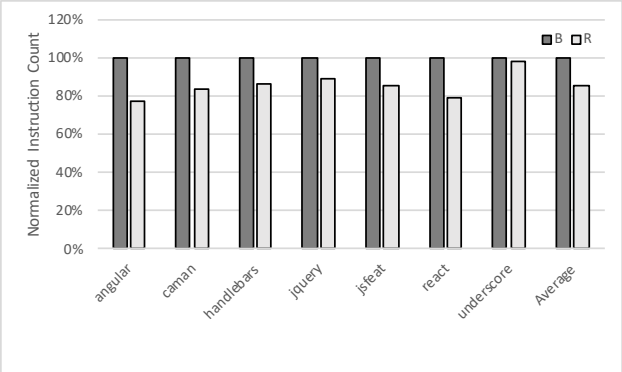
Table 2.2: Statistic related to IC during JavaScript initialization.

The last four columns show the statistics related to the operation of RIC on how `ICRecord` utilized to avoid IC misses. The fifth column is the percentage of IC misses which are saved by utilizing `ICRecord`. The number of hits in `ICRecord` is counted when updating `ICVector` at line 18 in Figure 2.7. On average, 53.45% of IC misses are avoided this way. The next three columns show the percentage of IC misses which `ICRecord` cannot help due to different reasons. The sixth column shows the percentage of IC misses which have context-dependent handlers which is not the target of RIC. The seventh column shows the percentage of IC misses of which incoming object is a global object. We disable RIC for global objects because we believe it is more realistic configuration considering the common case when the same JavaScript library is shared by multiple webpages. The last column shows the percentage of IC misses which `ICRecord` cannot help for other reasons. The majority of these are the cold misses on transitioning object access sites where RIC creates a transition hidden class and validates dependent object access sites.

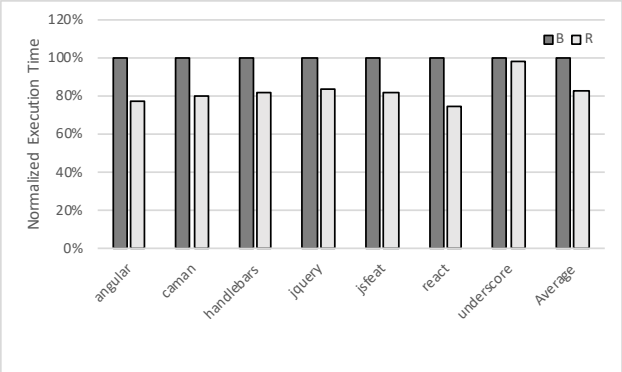
2.6.2 Performance Improvement

Figure 2.9(a) shows the number of dynamic instructions normalized to the configuration *B*. On average, *R* reduces the number of instructions by 15%. Similarly, Figure 2.9(b) shows the execution time normalized to the configuration *B*. On average, *R* reduces the execution time by 17%.

RIC improves the performance of benchmarks differently and its impact is roughly proportional to the number of IC misses saved as shown in Table 2.2.



(a) Instruction count normalized to *B*.



(b) Execution time normalized to *B*.

Figure 2.9: Performance improvement normalized to *B*.

2.7 RELATED WORK

Ahn et al. [30] propose a modified type system which decouples the prototype pointer from hidden classes and enables the hidden class and IC information to be reused when refreshing the same website. Its applicability is, however, somewhat limited as it lacks the mechanism to make the information persistent and improves the performance only under the specific conditions (e.g., reloading the same website before garbage collection and without restarting the browser). In addition, changing the hidden class structure may hurt other optimization techniques which require stronger type guarantee provided by the original hidden class with the prototype pointer. On the other hand, our proposal maintains the information in a persistent manner to be more generally applicable and does not interfere with other optimization techniques as it does not change the hidden class structure.

Oh and Moon [9] proposes a snapshot technique to accelerate JavaScript initialization for mobile applications. V8 also has a similar API to create a custom startup snapshot [31]. Such approaches take a snapshot of the heap after the initialization of JavaScript frameworks and libraries. When the same application is invoked later, it restores the objects from the snapshot to the heap instead of recreating them by executing JavaScript code for initialization. While the snapshot technique may greatly reduce the startup time by completely avoiding JavaScript initialization, it has several limitations. First, it is too rigid for different applications to share information. Although two applications use the same JavaScript library, a snapshot is application-specific and each application has to create its own snapshot. In contrast, in our reusble IC design, the information is maintained for each JavaScript file so that different applications can share it. Second, it has a correctness issue if the initialization involves any undeterministic behavior, such as the use of random function, date function, IO operations, etc., of which output may be different in each invocation of the

same application. Again, in our proposal, the initialization code is still fully executed but only accelerated with the hints from the previous execution so that it produces the correct results for undeterministic behavior in every invocation.

There exist some server-side techniques to improve the page load time performance [32, 33]. Kedlaya et al. propose server-side type profiling for optimizing client-side JavaScript engines. They identify functions with dynamic behaviors which are frequently deoptimized and waste computing power through server-side profiling. Then, they mark such functions so that the client-side JavaScript engines do not attempt to optimize them. Compared to our proposal, they convey the information at the function level and are orthogonal to our approach. Google's PageSpeed Module [32] is an open-source Apache HTTP Server module which automatically applies best practices in the form of optimization filters to the website source codes including CSS, JavaScript and images. More than 40 available optimization filters include CSS and JavaScript concatenation, inlining, image optimization, resizing, etc. PageSpeed can be used to complement our approach to improve the page load time but does not address the issue of recycling compilation efforts, such as IC information.

2.8 CONCLUSION

JavaScript compilers have not been developed with the aim to resue compilation results across executions (i.e. upon revisiting a website or visiting another website which uses the same JavaScript libraries). In this paper, the analysis of the initialization of popular JavaScript libraries confirmed that there exist opportunities to reuse compilation efforts across executions: (i) the same hidden class is shared by many object access sites and (ii) the majority of handlers are context-independent. Based on the observations, we proposed a mechanism to record and reuse IC information across executions. We evaluated our enhanced IC design called *Reusable Inline Caching (RIC)* with the state-of-the-art Google V8 JavaScript engine by measuring its impact on the initialization time of popular JavaScript libraries. On average, by recycling IC information collected from the previous execution, the reusable IC reduced the number of IC misses by 53%, the dynamic instruction count by 15%, and finally the execution time by 17%.

CHAPTER 3: SHORTCUT: ARCHITECTURAL SUPPORT FOR FAST OBJECT ACCESS IN SCRIPTING LANGUAGES

3.1 INTRODUCTION

Dynamic scripting languages such as JavaScript [34], Python [35], and Ruby [36] are widely used in many application domains, both for clients [1, 2] and servers [3, 4]. Programmers like the portability, flexibility, and ease of programming of these languages, where everything — including functions and primitives — can be treated as objects, and objects can add and remove properties at runtime.

This same flexibility makes the task of using a compiler to optimize programs written in these languages challenging. A given read or write in the program may access different types of objects at different times. Since the compiler does not know the object types ahead of execution, it has to augment the code with many runtime checks which slow down execution.

The place in the code where an object property is to be read or written is called an *access site*. Access sites are organized to record information about the object types recently encountered at the access site. The code performs a series of checks to determine if an incoming object has the same type as any of the ones seen before. If so, the code jumps to the appropriate handler to perform the access. If all checks fail, the code jumps to the language runtime, which performs an expensive hash table lookup. The process of identifying the correct type and invoking the correct handler is called the *Dispatch* operation. The actual structure with checks and jumps is called the *Inline Cache* (IC) [20].

Our analysis of the code generated by the state-of-the-art Google V8 JavaScript JIT compiler [7] leads to some insights. Execution at an access site starts with a call to a dispatcher. The dispatcher code has a series of *if* statements, each of which includes a comparison to a type and a jump to a handler. This code executes many instructions, including hard-to-predict control-flow instructions. Specifically, our experiments with V8's baseline compiler shows that, on average for two suites of applications, the dispatch operation accounts for 22% of the applications' instruction count. Moreover, the branches in the dispatch operation increase the applications' average branch Mispredictions Per Kilo Instruction (MPKI) from 5.8 to 10.8.

Since inline caching is a central feature in dynamic scripting language implementations, this paper examines architectural support to optimize its operation. Based on the insights from the V8 analysis, our idea is to modify the instruction that calls the dispatcher so that, under typical conditions, it skips most of the instructions in the IC execution. This is possible thanks to a new hardware table that records the state observed in prior invocations

of the code.

Our proposed architecture is called *ShortCut*, and performs two levels of optimization. In the *Plain* design, it transforms the call to the dispatcher into a call to the correct handler — bypassing the whole dispatcher execution. In the *Aggressive* design, it transforms the call to the dispatcher into a simple load or store — this time bypassing the execution of both dispatcher and handler.

We implement the ShortCut software changes in the state-of-the-art Google V8 JIT compiler, and the ShortCut hardware modifications in a Pin-based simulator. We use the Octane and SunSpider JavaScript application suites. Our evaluation shows that Plain ShortCut reduces the average execution time of the applications by 30% running under the baseline compiler, and by 11% running under the maximum level of compiler optimization. Aggressive ShortCut performs only slightly better.

3.2 BACKGROUND AND MOTIVATION

3.2.1 Inline Caching in Scripting Languages

Many scripting languages, including JavaScript [34], Python [35], and Ruby [36], implement dynamic type systems. This means that, at runtime, a program can dynamically create new types by adding or subtracting properties (i.e., data fields or methods) to an object. Such flexibility promotes ease of programming.

With this support, however, a given access site can now encounter objects of different types at different times. As a result, in a naïve design, the code at the access site has to perform an expensive dictionary lookup to locate the property being accessed. This is in contrast to statically-typed languages, such as C++ or Java, where the fields of a type do not change at runtime, and are located at fixed offsets determined by the type definition. In this case, an access site simply performs a memory access.

To overcome this limitation of dynamically-typed languages, the Smalltalk language introduced a technique called *Inline Caching* (IC) [20]. The idea is to augment an access site with a *software* cache that contains the outcome of the most recent dictionary lookup(s). When an access site accesses a property of an object, the code first searches the cache for this object type. If the object type is found, the code accesses the property with low overhead; otherwise, it has to perform a dictionary lookup and update the inline cache.

This approach is widely used in modern virtual machines for dynamic scripting languages. An access site is called monomorphic, polymorphic, or megamorphic if it sees a single type, a few different types, or many different types, respectively.

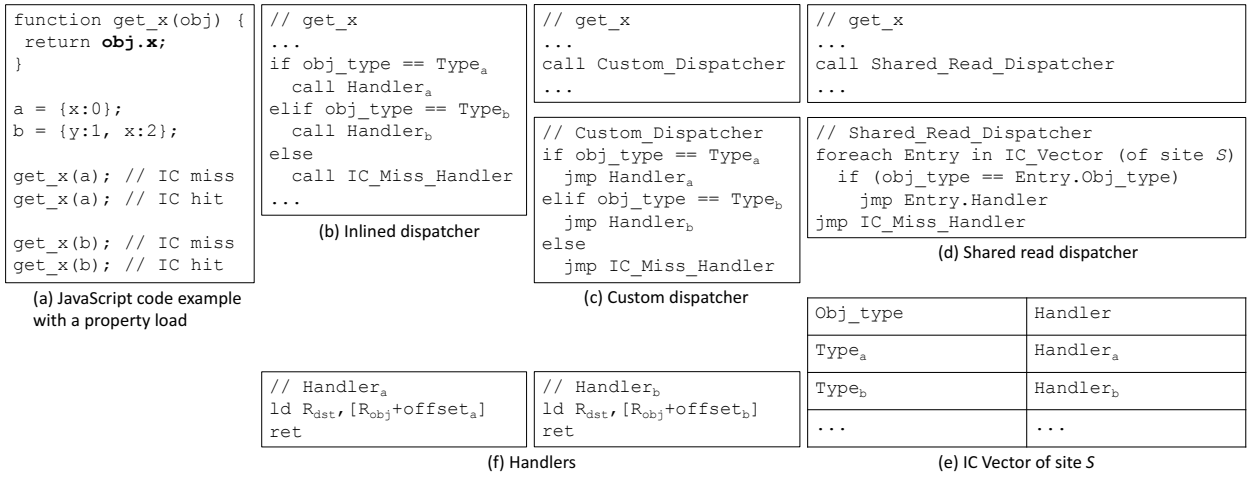


Figure 3.1: Code generation example with different inline cache implementations.

As an example, consider Figure 3.1(a). It shows function `get_x`, which returns the value of property `x` of the object argument. We have two objects, `a` and `b`. They have different types because object `a` only has a property `x`, while object `b` has properties `y` and `x`. The example code performs two calls to `get_x` with object `a`, and two with object `b`. In both cases, the first one misses in the IC and the second one hits.

Intuitively, ICs maintain a lookup result as a pair {object type, handler}. The handler is code specialized to access a particular property of a particular object type. It can be as simple as a memory access, or it can perform complex operations according to the language semantics. The handler is generated by the language runtime. The process of identifying the correct type and invoking the correct handler is called the *Dispatch* operation.

3.2.2 Approaches to Inline Caching

There are three different implementations of inline caching based on the dispatch mechanism used.

Inlined Dispatcher In this design, the dispatcher is inlined at the access site. This is the original design in Smalltalk [20], which only supported monomorphic access sites. It can be generalized to support polymorphic access sites by generating code with a series of *if* statements, where each one checks for a different type and, if there is a match, calls the correct handler.

Figure 3.1(b) shows an example. This code performs the actual reading of `obj.x` in function `get_x`. In Figure 3.1(b), the code checks for $\texttt{Type}_\texttt{a}$ and $\texttt{Type}_\texttt{b}$ in sequence. The handlers access the property and then return, as shown in Figure 3.1(f). If no match occurs,

the code in Figure 3.1(b) calls an IC miss handler in the language runtime, which performs the access and then extends the IC with a new comparison and handler call.

The main advantage of this design is a relatively low dispatching overhead. A disadvantage is that every time that a new type is encountered, the entire procedure that contains the access site has to be extended and, therefore, recompiled. Frequent code recompilation adds up to the execution time and degrades instruction cache performance. Thus, this design is typically used only by the highest optimizing tier in multi-tier JIT compilers, which generates code for hot functions with stable type information. For example, the highest compiler tier in V8 [7] uses this design, and even inlines the handlers at the access site.

Custom Dispatcher In this design, the dispatcher is taken out from the access site. The access site simply has a call to a dispatcher specific for this site [37]. In the dispatcher, the code has the usual set of *if* statements with comparisons and jumps to handlers.

Figure 3.1(c) shows an example. When a new type is encountered, a new custom dispatcher is generated that includes the additional comparison and jump. Since the new dispatcher has a new address, the call at the access site is updated to transfer execution to the new address. This does not require recompilation of the procedure that contains the access site. However, it involves writing to the code, to invoke the dispatcher at a different address, which causes the invalidation of instruction cache state.

The advantage of this approach is that it does not recompile the procedure with the access site when a new type is encountered. Still, a new custom dispatcher needs to be generated, and the modification of the call target hurts instruction cache performance. Also, the memory overhead of maintaining a custom dispatcher per access site is not negligible in resource-constrained devices.

This design was used by the *baseline* compiler in V8 until version 4.8 released in November 2015. Recall that V8 has two tiers of compilation. The baseline compiler performs the initial compilation for all executed code. After warm-up, selected regions of performance-critical code are recompiled by an optimizer (i.e., Crankshaft).

Shared Dispatcher In this design, instead of having a custom dispatcher for each access site, there is one dispatcher shared by all the read access sites, and one dispatcher shared by all the write access sites. The read dispatcher maintains a data structure with individual information for each read access site. Such individual information is a set of 2-tuples called *Inline Cache (IC) Vector*, where each tuple contains a type encountered by that site and its handler. The write dispatcher maintains a similar data structure with an IC Vector for each write access site.

Let us call Site S the read access site in Figure 3.1(a) — i.e., `obj.x`. Figure 3.1(d) shows the shared read dispatcher code, as it iterates over the entries of the IC Vector corresponding to Site S. The shared read dispatcher code uses a register initialized at Site S to automatically index the correct IC Vector. Figure 3.1(e) shows the IC Vector for Site S. The code in the shared dispatcher iterates over the entries in this IC Vector, searching for the matching type. If the matching type is found, the code jumps to the corresponding handler. Otherwise, it jumps to the IC miss handler in the language runtime, which performs the load and then adds a new 2-tuple to the IC Vector for S.

The advantage of this design is that it eliminates any recompilation upon an IC miss. This is because it stores the previous lookup results as *data*, rather than hard-coding types and handler addresses in the code. Thus, IC miss handling is quick. Also, the memory overhead of the dispatcher code is small. However, getting to the handler is now more expensive: it requires more instructions, many of which are memory accesses. This design has been used by the baseline compiler in V8 since November 2015. In addition, as part of V8's continuous enhancement, its upcoming new interpreter tier will use a shared dispatcher.

Our Focus Our focus in this paper is on optimizing an IC with a shared dispatcher. The reason is that this is the design currently used by the state-of-the-art V8's baseline compiler, but has performance shortcomings. Moreover, similar optimization insights will also apply to an IC with a custom dispatcher. An IC with an inlined dispatcher, such as the one used by the V8 optimizing compiler tier, could also benefit from our optimization, albeit to a lesser extent because it is already highly optimized.

The performance of an IC with a shared or custom dispatcher is important regardless of the availability of an optimizing tier. Indeed, even if the optimizing compiler is available, a significant fraction of the execution of programs uses code generated by the baseline compiler — and, hence, uses a shared or custom dispatcher. The reason is threefold. First, it takes a while for the optimizing tier to engage. Second, if any assumption made by any optimization fails (e.g., an unexpected object type is encountered), the baseline tier is re-invoked. Lastly, there are some functions in a program that the optimizing tier abstains from compiling, often based on heuristics; they include *eval* constructs and other complicated cases.

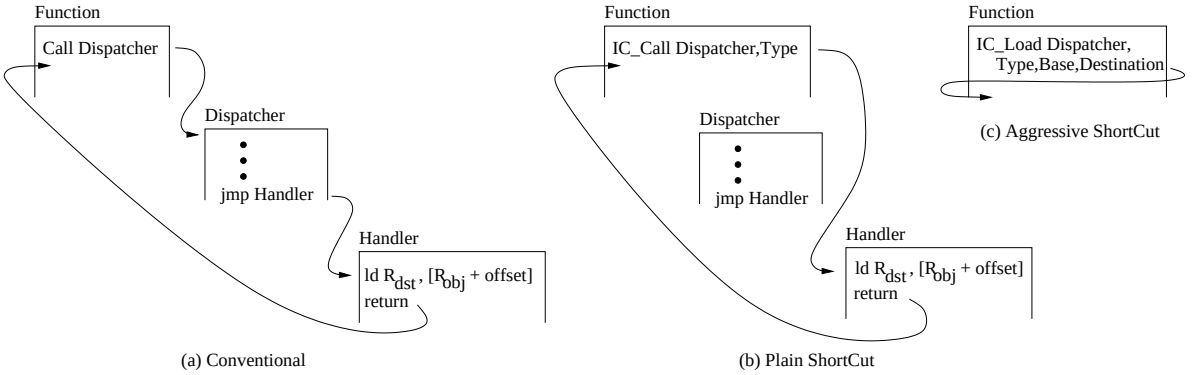


Figure 3.2: Operations of a conventional IC (a), Plain ShortCut (b), and Aggressive ShortCut (c).

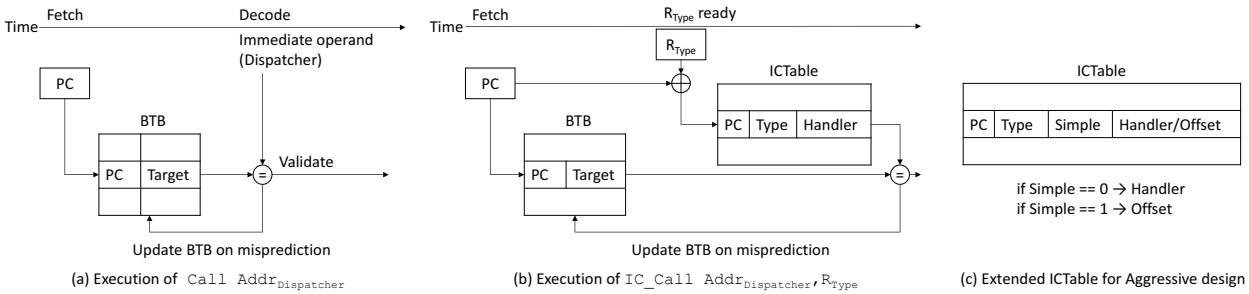


Figure 3.3: Structures used in a conventional IC (a), Plain ShortCut (b), and Aggressive ShortCut (c).

3.3 SHORTCUT ARCHITECTURE

3.3.1 Main Idea

Inline caching is a central feature in dynamic scripting language implementations. Unfortunately, an examination of custom or shared dispatchers reveals that ICs still have a lot of overheads. The general structure of these ICs is shown in Figure 3.2(a). The access site (e.g., `obj.x` in Figure 3.1(a)) uses a procedure call to invoke the dispatcher. The dispatcher, after some checks with conditional branches, uses an unconditional jump to reach the handler. In this process, many instructions are executed, including many control-flow instructions, which often flush the pipeline.

The goal of this paper is to make inline caching in dynamic scripting languages significantly more efficient. We propose two designs, which we call *Plain ShortCut* and *Aggressive ShortCut*. In Plain ShortCut, we want to bypass the dispatcher and transfer execution directly to the correct handler as shown in Figure 3.2(b). Since a given access site may use

different handlers at different times, we have to predict the correct handler. Then, we need a way to validate the prediction and rollback execution if the prediction was incorrect.

Figure 3.2(b) shows the simplified control flow of Plain ShortCut. We replace the original call with a new instruction called *IC_Call*. Since the correct handler to use depends on the type of the object accessed, *IC_Call* takes, as an additional operand, the type of the object accessed. If the prediction fails, execution falls back to the conventional path of Figure 3.2(a).

In practice, a handler often performs nothing more than a simple load from or store to an object's property. This is shown in Figure 3.1(f). Calling the handler and returning has substantial overhead. Hence, Aggressive ShortCut takes a more aggressive approach, as shown in Figure 3.2(c). The idea is to perform the load or the store as part of the original instruction that used to call the dispatcher.

These simple handlers with a load or store operation use a base register, an offset, and a destination register. We add these two registers as additional operands to the instruction. Moreover, the offset is fixed for a given access site and object type. Hence, if we predict the type of the object correctly, the hardware can perform the load or store operation as part of this instruction.

As shown in Figure 3.2(c), we replace the call with a new instruction called *IC_Load* (or *IC_Store*). These instructions are used in read and write access sites, respectively. When an *IC_Load* or *IC_Store* executes, if the prediction succeeds, the instruction either performs the actual load/store operation in hardware (if the handler is a simple load or store operation) or defaults to an *IC_Call* (if the handler is more complicated). If the prediction fails, execution falls back to the conventional path of Figure 3.2(a).

To see how we support these ideas, consider first how a conventional `Call Dispatcher` instruction works in Figure 3.3(a). V8 uses a call instruction with the PC-relative address of the dispatcher as an immediate. At instruction fetch, the BTB is accessed, and provides the predicted target address. The pipeline starts fetching instructions at the predicted target. When the dispatcher address is finally generated, the target is validated. If a misprediction occurred, the pipeline is flushed, the BTB is updated, and the pipeline starts fetching instructions at the dispatcher.

Plain ShortCut extends this procedure as shown in Figure 3.3(b). The `Call Dispatcher` instruction is now replaced with our *IC_Call* instruction. The *IC_Call* instruction takes an additional register ($\texttt{R}_{\texttt{Type}}$) with the type of the object accessed. In addition, we add a new hardware table called *Inline Cache Table (ICTable)*. Each entry in the table contains the address of an *IC_Call* instruction, an object type, and the address of the handler for that object type.

At instruction fetch in Plain ShortCut, the BTB is accessed and proceeds as usual — in the best case, as we will see, predicting the *handler* address as the target (rather than the dispatcher address). The pipeline starts fetching instructions at the predicted target. When $\mathtt{R}_{\mathtt{Type}}$ becomes available, the address of *IC_Call* and the object type are hashed together to index into the ICTable. On a hit, the ICTable provides the handler address. If the BTB had provided the correct prediction, execution continues; otherwise, the pipeline is flushed, the BTB is updated, and the pipeline starts fetching instructions at the correct handler address provided by the ICTable. On an ICTable miss, the dispatcher has to be executed, and we flush the pipeline unless the BTB had provided the dispatcher address.

Aggressive ShortCut upgrades the ICTable to the design of Figure 3.3(c). The table has one extra field, called *Simple*, and the old *Handler* field becomes *Handler/Offset*. If *Simple* is zero, the *Handler/Offset* field contains the handler address as in Plain ShortCut; if *Simple* is set, the *Handler/Offset* field contains the offset of the requested property in the structure of the object being accessed. Then, the hardware reads the offset, adds it to the base register provided by the instruction, and performs the load/store operation.

3.3.2 Detailed Design of Plain ShortCut

The Plain ShortCut architecture is shown in Figure 3.3(b). It uses the set-associative hardware table called ICTable. Each ICTable entry contains three virtual addresses (VA). The first one is the VA of the *IC_Call* instruction, which we refer to as the address of the access site. The second is the VA of an object type that has been previously seen at this access site. This is because V8 represents object types as memory addresses. The third is the VA of the handler for this type.

The ICTable may contain multiple entries for the same access site. In this case, each entry contains a different type. As a result, the ICTable is indexed by a hash of the access site address and the type.

The ICTable provides the address of the handler to execute. The table *cannot* provide incorrect results, as it is not a prediction. If an access to the ICTable hits, it contains the VA of the handler that should be executed.

On an ICTable miss, the ICTable is updated in software by the dispatcher, once the dispatcher determines the correct handler that will be executed. Such an update occurs with a special instruction called *IC_Update*. There is also a way to flush the whole ICTable. This is done with the *IC_Flush* instruction. We describe these instructions in Section 3.4.

The ICTable is accessed by the *IC_Call* instruction described above. The access occurs as soon as the access site address and the type are known during the execution of the

instruction, after the access to the BTB at fetch time.

The BTB and the ICTable work together but have different roles. The BTB is accessed in the pipeline's front end. It provides a prediction, which is used to direct the fetching of instructions. However, the BTB prediction may be incorrect. The ICTable validates or refutes the prediction of the BTB later in the pipeline. If the prediction is refuted, the pipeline is flushed and the BTB is updated.

The BTB has at most one entry for a given access site. If the entry exists, it can have one of three different types of target addresses: the address of the dispatcher, the address of the correct handler, or the address of an incorrect handler. The last case occurs when the access site encounters a type that is different from the one last seen at this site.

The Plain ShortCut operation, therefore, involves two steps (Figure 3.3(b)). At fetch time, the BTB is accessed and provides a target. The pipeline starts fetching instructions from there. Later, when the ICTable is accessed, there are two possible outcomes: either the ICTable misses or hits. If it misses, the hardware uses the dispatcher address generated in the decode stage as the address of the next instruction to fetch. Hence, if the BTB had predicted this target, the pipeline continues execution; otherwise, the pipeline is flushed and fetching starts at the dispatcher address. If, instead, the ICTable hits, the hardware provides the handler address in the ICTable entry as the correct target. As before, if the BTB had predicted this target, the pipeline continues execution; otherwise, the pipeline is flushed and fetching starts at the handler address provided by the ICTable.

The entries in the BTB and in the ICTable can be evicted independently due to conflicts in their structures. For simplicity, when an ICTable entry is evicted, the BTB is not modified, and vice versa. As a result, when an ICTable entry for an access site and type is evicted, it is possible that the BTB is left with an entry that will not be useful even if the same access site and type are encountered. This is because while the target in the BTB entry is the correct handler address, the ICTable miss will trigger a pipeline flush and the redirection of instruction fetching to the dispatcher address. With additional hardware, we could update the BTB with the dispatcher address upon the ICTable entry eviction to avoid such misprediction. We choose not to do it for simplicity.

ICTable Operations Our Plain ShortCut design works best when a given access site keeps encountering the same type repeatedly. In this case, the ICTable will store an entry for this access site and type, and the corresponding BTB entry will set its target field to be the correct handler address. The pipeline is never flushed and the dispatcher is always avoided.

There are three cases when things go wrong. One case is when the access site encounters

#	When the Case Happens	BTB Outcome	ICTable Outcome	Pipeline Flush?	What is Executed	Other Actions Taken
1	First execution of access site	Miss	Miss	Yes	Dispatcher + Handler	Add entry to BTB w/ PC + handler Add entry to ICTable w/ PC + type + handler
2	Encounter same type again	Hit. Correct handler	Hit	No	Handler	
3	Encounter diff. type (not yet seen before)	Hit. Incorrect handler	Miss	Yes	Dispatcher + Handler	Update BTB entry w/ handler Add entry to ICTable w/ PC + type + handler
4	Encounter diff. type (already seen before)	Hit. Incorrect handler	Hit	Yes	Handler	Update BTB entry w/ handler
5	After ICTable eviction: same type accessed	Hit. Correct handler	Miss	Yes	Dispatcher + Handler	Update BTB entry w/ handler Add entry to ICTable w/ PC + type + handler
6	After BTB eviction: same type accessed	Miss	Hit	Yes	Handler	Add entry to BTB w/ PC + handler

Table 3.1: Interaction between the BTB and the ICTable.

different object types that keep alternating. This will cause the BTB to mispredict and the pipeline will be flushed; however, the dispatcher will not be executed, as the ICTable maintains the handlers for the multiple types. Another unfavorable case is when a useful ICTable entry is evicted. In this case, when the evicted type is encountered again, the dispatcher will have to be executed. Finally, a useful BTB entry may be evicted. This will cause a BTB miss when the corresponding access site is executed again with the same type; however, it will not cause dispatcher execution.

Table 3.1 summarizes all the possible cases. It lists when each case happens, the BTB outcome, the ICTable outcome, whether there is a pipeline flush, what is executed, and other actions taken. *Case 1* is the first execution of the access site, which induces misses in both structures and causes both dispatcher and handler execution. *Case 2* is an execution of the access site that encounters the same type as in the prior execution. This is the best possible case. The next two cases occur when the execution encounters a different type than in the prior execution — a type that has never been seen before (*Case 3*) or that has already been seen before (*Case 4*).

Consider now that the ICTable entry for an access site and type is evicted while it had a corresponding valid BTB entry. A new case occurs when the same access site and type are encountered again before the BTB entry has changed. In this case, the pipeline is flushed because both the dispatcher and the handler need to be executed. The ICTable and the BTB are updated, even though the BTB had the correct handler address (*Case 5*). Note that if after the ICTable entry eviction the same access site is encountered with a different type, we have one of the situations discussed in *Case 2*, *Case 3*, or *Case 4*.

Finally, assume that a BTB entry for an access site is evicted. *Case 6* is the case when the access site is accessed again with a type that has an entry in the ICTable.

3.3.3 Detailed Design of Aggressive ShortCut

Some of the read and write access sites, when they encounter certain object types, execute handlers that perform nothing more than a simple load from or store to a property of the object. They do not perform any other operation, such as traversing the object's prototype chain. We call these handlers *simple* handlers. In Aggressive ShortCut, we want to perform the load or store without having to jump to the handler. Hence, we propose to functionally transform the instruction that calls the dispatcher into a load or a store when the conditions allow it. The goal is to emulate the low overhead of a statically-typed language.

As shown in Figure 3.1(f), these simple handlers execute an instruction of the form `ld` R_{dst}`,[`R_{base}`+offset]` (or `st` R_{src}`,[`R_{base}`+offset]` at a write access site). R_{base} contains the base address of the object accessed by the handler (hence, we use R_{obj} in Figure 3.1(f)). It is set by the V8 compiler before calling the dispatcher. `offset` is the offset of the desired property from the object's base. V8 hardcodes `offset` in the handler for that access site and type. Finally, R_{dst}/R_{src} is the register that receives the datum from the memory or that provides it to the memory. It is always the same register as part of the calling convention.

The Aggressive ShortCut architecture augments the ICTable as shown in Figure 3.3(c). It has an additional one-bit field called *Simple*. If a given entry corresponds to a simple handler, *Simple* is set to one, and the field that used to contain the handler address now contains the offset used in the load or store. If the entry does not correspond to a simple handler, *Simple* is set to zero, and the *Handler/Offset* field contains the handler address.

On the software side, we modify the IC miss handler so that, when it generates a handler, it checks if it is a *simple* handler. If so, as it inserts the entry in the ICTable, it sets the *Simple* bit and stores the offset in the *Handler/Offset* field. In addition, as it adds the new type and handler address in the software IC Vector for the site, it also records that this is a simple handler and its offset.

Finally, we replace the *IC_Call* instruction in read and write access sites with *IC_Load* and *IC_Store*, respectively. These instructions take two additional registers as operands, which are used as R_{base} and R_{dst}/R_{src}. When these instructions execute, if they hit in the ICTable, the hardware checks if the *Simple* bit is set. If so, the hardware reads the *Handler/Offset* field and performs the `ld` R_{dst}`,[`R_{base}`+offset]` or `st` R_{src}`,[`R_{base}`+offset]` operation. Neither the dispatcher nor the handler is called. In addition, the BTB target is updated to point to the instruction that follows the *IC_Load* or *IC_Store*. This is because we have eliminated all calls and jumps, and there is no control flow change. We have performed the load or store as part of the *IC_Load* or *IC_Store* instruction.

IC_Load and *IC_Store* for a simple handler follow a similar algorithm as that in Table 3.1.

For example, if the ICTable misses, the dispatcher executes, and inserts into the ICTable an entry from the IC Vector that has a set *Simple* bit and an offset.

When the handler is not simple (i.e., *Simple* is clear), *IC_Load* and *IC_Store* operate exactly like *IC_Call*.

3.4 ADDITIONAL DESIGN ASPECTS

3.4.1 ISA Extensions

ShortCut extends the ISA to expose the ICTable to the software. It adds the five instructions shown in Table 3.2. The first three instructions can replace the call to the dispatcher at IC access sites (`call` $\texttt{Addr}_{\texttt{D}}$).

Instruction	Functionality
IC_Call (Plain)	Calls the handler if it hits in ICTable; calls the dispatcher otherwise
IC_Load (Aggressive)	Performs a load if it hits in ICTable and *Simple* is set; calls handler or dispatcher otherwise
IC_Store (Aggressive)	Performs a store if it hits in ICTable and *Simple* is set; calls handler or dispatcher otherwise
IC_Update (Plain/Aggr.)	Installs an entry in the ICTable; updates the BTB
IC_Flush (Plain/Aggr.)	Flushes the ICTable

Table 3.2: Instructions added by ShortCut.

`IC_Call` $\texttt{Addr}_{\texttt{D}}$,$\texttt{R}_{\texttt{Type}}$. It takes as operands the relative address of the dispatcher and a register with the type of the object accessed. It indexes into the ICTable with a hash of the PC and $\texttt{R}_{\texttt{Type}}$. If it hits, control is transferred to the handler provided by the ICTable; otherwise, control is transferred to the dispatcher.

`IC_Load` $\texttt{Addr}_{\texttt{D}}$,$\texttt{R}_{\texttt{Type}}$. It takes as operands the relative address of the dispatcher, one explicit register with the object type, and two implicit registers. One of the implicit registers contains the base of the object ($\texttt{R}_{\texttt{base}}$) and the other will receive the value of the object property ($\texttt{R}_{\texttt{dst}}$). It replaces *IC_Call* at read access sites in Aggressive ShortCut.

IC_Load indexes into the ICTable with a hash of the PC and $\texttt{R}_{\texttt{Type}}$. If it hits and the ICTable provides an offset, the instruction performs `ld` $\texttt{R}_{\texttt{dst}}$`,[`$\texttt{R}_{\texttt{base}}$`+offset]`; if it hits and the ICTable does not have an offset, control is transferred to the handler; if it misses, control is transferred to the dispatcher.

IC_Store Addr$_D$,R$_{Type}$. It takes as operands the relative address of the dispatcher, one explicit register with the object type, and two implicit registers. The implicit registers contain the base of the object (R_{base}) and the value that will be stored into the object property (R_{src}). It replaces *IC_Call* at write access sites in Aggressive ShortCut. *IC_Store* operates like *IC_Load* except that, if it hits in the ICTable and the ICTable provides an offset, the instruction performs `st` R_{src}`,[`R_{base}`+offset]`.

IC_Update R$_{PC}$,R$_{Type}$. It takes as operands a register with the access site address (R_{PC}), a second register with the object type (R_{Type}), and either one or two implicit registers (in Plain and Aggressive, respectively). The implicit register in Plain ShortCut contains the handler address ($R_{handler}$); those in Aggressive ShortCut contain the simple bit (R_{simple}), and either the handler address or the offset (depending on the value of the simple bit) (R_{value}). *IC_Update* indexes into the ICTable with a hash of access site address and object type, and creates an entry in the table, filling it with its three or four register operands.

IC_Update also updates the BTB entry indexed by R_{PC}. In Plain ShortCut, the entry's target is set to the handler; in Aggressive ShortCut, if R_{simple} is set, the entry's target is set to R_{PC} `+ 4`; otherwise, it is set to the handler.

IC_Flush. It invalidates all entries in the ICTable. It is used by the language runtime after garbage collection, and by the OS at context switches to prevent the use of stale or incorrect data.

3.4.2 Integration with the Compiler

We modify V8 to use ShortCut's new instructions. In the Plain ShortCut design, we replace the call to the dispatcher at each access site with *IC_Call*. Recall that *IC_Call* takes the object type as an operand. In the original V8, the object type is read in the dispatcher. Consequently, we move the instruction that reads the type from the dispatcher to before the *IC_Call*.

In the Plain and Aggressive ShortCut designs, we modify V8's dispatcher and IC miss handler to use *IC_Update*. Specifically, when the dispatcher finds the correct handler in the IC Vector, we invoke *IC_Update* to upload the information into an ICTable entry. Similarly, when the IC miss handler creates a handler, we also invoke *IC_Update* to upload the information into an ICTable entry. These instructions add little overhead because dispatcher and IC miss handler are invoked infrequently.

In the Aggressive ShortCut design, we replace the *IC_Call* at each read and write access site with *IC_Load* and *IC_Store*, respectively. These instructions need R_{base} (and R_{src} in the

case of *IC_Store*) to be set in advance. The original V8 already sets these registers before the call to the dispatcher. Hence, we do not need to make changes.

3.4.3 ShortCut Overheads

ShortCut adds both software and hardware overheads. The software overheads are very small, and come from slightly higher memory pressure and from the instructions added.

The memory pressure increases slightly for two reasons. First, moving the instruction that sets $\mathtt{R}_{\mathtt{Type}}$ from the shared dispatcher to before the *IC_Call* marginally increases the code size. Second, extending the IC Vector in the Aggressive design to include the *Simple* bit increases the data size a bit.

To assess the cost of the new instructions, we consider the following. *IC_Call* is a call instruction that, as it executes, checks the ICTable and confirms or refutes the prediction. *IC_Load* and *IC_Store* additionally perform a load or a store based on data accessed from the table. A possible latency for these instructions is one more cycle than a call (for *IC_Call*) and two more cycles than a load or store (for *IC_Load* and *IC_Store*). However, these additional latencies are mostly hidden in an out-of-order pipeline, and are dwarfed by the instructions' positive impact on prediction and avoidance of control flow change.

For the other two instructions, we assume that *IC_Update* takes 6 execution cycles, and *IC_Flush* 20 execution cycles. These instructions are too rare to cause any noticeable overhead.

The instruction count increases slightly because we add the *IC_Update* instruction to the dispatcher and to the IC miss handler, and need to use *IC_Flush* at context switches and after garbage collection invocations. However, the dispatcher and the IC miss handler are invoked relatively infrequently, and context switches and garbage collection invocations are even less frequent.

The main ShortCut hardware overhead is the ICTable. Each entry in the ICTable contains three memory addresses — plus one bit in the Aggressive design. Current x86-64 processors use 48 bits for a virtual address. Hence, the size of a 512-entry ICTable is about 9 KB, which is a modest overhead.

Finally, although *IC_Call*, *IC_Load*, and *IC_Store* interact with the BTB, they do not place additional size requirements on the BTB. This is because each of these instructions replaces an original instruction that called the dispatcher and already occupied a BTB entry. Consequently, the total number of instructions competing for BTB entries is unchanged.

3.5 DISCUSSION

3.5.1 Implications for Other Languages

While we use V8 to demonstrate the effectiveness of ShortCut, the idea can be ported to other JavaScript compilers, and to compilers of other dynamic scripting languages. If such compilers implement an IC in their lower tier, a shared or custom dispatcher is a much better design choice than an inlined dispatcher, which has substantial recompilation overhead. For example, WebKit's baseline compiler implements a custom dispatcher [38]. Moreover, the ShortCut hardware can be largely reused with minimal variations for any different implementation of a shared or custom dispatcher. Finally, while our software changes are based on V8's shared dispatcher design, we speculate that compiler teams for other languages and implementations would find it relatively easy to support ShortCut.

It is also possible to apply ShortCut to other code structures. For example, it can be applied to virtual function calls in statically-typed object-oriented languages [39], to avoid *vtable* lookups and improve indirect branch prediction. Similarly, ShortCut can be used to improve the performance of switch statements, by storing case labels and the corresponding handlers in the ICTable.

3.5.2 Flushing the ICTable

The ICTable needs to be flushed upon garbage collection and upon context switches to prevent the use of stale or incorrect data. It is possible to avoid unnecessary flushes after garbage collection by tracking the types of objects collected, and executing *IC_Flush* only if the type structure is altered by the garbage collection. In addition, we can extend the ICTable with a bloom filter [40] to track the types stored in the ICTable. In this way, we can flush only if the types altered by the garbage collection hit on the bloom filter.

3.5.3 Applicability to Interpreters

As many scripting languages rely on interpreters, it would be interesting to extend ShortCut to support interpreters. Interpreters lack the capability of dynamic code generation; they record profiling information as data instead of inlining it in codes. Hence, when interpreters implement an IC, they use a design like the shared dispatcher. For example, to record profiling information, WebKit's LLInt [38] uses bytecode data, and the upcoming V8 Ignition [7] relies on the IC Vector of the shared dispatcher. Consequently, ShortCut has

the potential to improve the IC operation of interpreters.

In interpreters, however, all access sites use the same instruction to call the dispatcher from the bytecode handler. Consequently, all access sites are predicted with a single BTB entry that transfers execution to the shared dispatcher. As a result, if we used ShortCut, we would have a low BTB prediction accuracy, because a single BTB entry now needs to transfer execution to different handlers. To overcome this limitation, we could extend ShortCut to index the BTB with bytecode addresses instead of PCs. A similar approach is used in previous BTB proposals [41, 42].

3.6 EXPERIMENTAL SETUP

To support ShortCut, we implement the compiler changes discussed in Section 3.4.2 to the state-of-the-art Google V8 JavaScript JIT compiler [7]. Our implementation uses V8 version 5.1, which was the most recent release at the time of performing our experiments. V8 consists of two compiler tiers; every function starts with the baseline tier, and only hot functions are recompiled by the optimizing tier. We turn off garbage collection to achieve deterministic results. We run the well-known Octane 2.0 [29] and SunSpider 1.0.2 [43] application suites.

To model and evaluate the ShortCut hardware, we extend the Sniper simulator [44], which is a widely-used Pin-based [21] architecture simulator. The parameters of the processor architecture are shown in Table 3.3. The baseline processor uses a 4K-entry BTB to predict indirect branches [45]. The ICTable modeled has 512 entries. While Sniper is an application-level simulator, we model the effect of context switches and garbage collection invocations on ICTable by flushing the ICTable every 15 milliseconds with *IC_Flush*es.

Core	4-wide out-of-order, 128-entry ROB, 2.66GHz
Branch Predictor	Hybrid predictor BTB: 4K entries, 4-way, RR replacement, 96b/entry Branch misprediction penalty: 15 cycles
ICTable	512 entries, 4-way, RR replacement, 145b/entry
Caches	L1-I: 32KB, 4-way, 4-cycle latency L1-D: 32KB, 4-way, 4-cycle latency L2: 256KB, 4-way, 12-cycle latency L3: 8MB, 16-way, 30-cycle latency Block size: 64B, LRU replacement
Memory	120-cycle minimum latency 16 DRAM banks

Table 3.3: Processor architecture. RR means round robin.

Name	Configuration
BO, B	Baseline: Conventional processor using the unmodified V8
IO, I	Ideal: Baseline enhanced with a perfect BTB for the IC
PSO, PS	Plain ShortCut using the modified V8
ASO, AS	Aggressive ShortCut using the modified V8

Table 3.4: Architecture and compiler configurations evaluated.

We evaluate the four pairs of configurations shown in Table 3.4. Baseline is a conventional processor using the unmodified V8 compiler (*BO, B*). Ideal is the baseline configuration enhanced with a perfect BTB that always provides the correct target for branches in the IC code structure (*IO, I*). This serves as an upper bound for existing proposals for BTBs that improve indirect branch prediction [41, 46, 47]. Finally, we model Plain ShortCut with its modified V8 compiler (*PSO, PS*), and Aggressive ShortCut with its modified V8 compiler (*ASO, AS*).

Within each pair, the two configurations are: one with the V8 optimizing tier enabled (configurations terminated in *O*), and one with such tier disabled (configurations not terminated in *O*). The configurations without the optimizing tier estimate the impact of ShortCut in dynamic scripting languages that do not have an advanced optimizing tier. Also, recall that we do not apply ShortCut to the IC with an inlined dispatcher used by the optimizing tier. Applying ShortCut to it could potentially improve performance more.

Recall from Section 3.3.3 that the Aggressive ShortCut design supports *simple* load and store handlers. Due to the complexity of the V8 compiler, however, our compiler support for Aggressive ShortCut (*ASO, AS*) is currently limited to *IC_Load* only (no *IC_Store*). To assess the potential of Aggressive ShortCut, we note that of all the handler invocations in our applications, 75.7% are loads and 24.3% are stores. Further, 15.1% of the load handler invocations and 17.2% of the store handler invocations, respectively, are *simple*.

3.7 EVALUATION

3.7.1 Characterization

We start by investigating the overhead of the IC in the unmodified V8 compiler. We measure the dynamic instruction count and the branch misprediction count. We categorize the dynamic instructions in an application execution into three categories: *IC*, *Code*, and *Runtime*. *IC* is instructions spent in the IC shared dispatcher, executing the code in Figure 3.1(d); *Code* is instructions spent in the rest of the application code generated by the

compiler; finally, *Runtime* is instructions spent in the language runtime — e.g., to support compilation, string operations, and regular expressions.

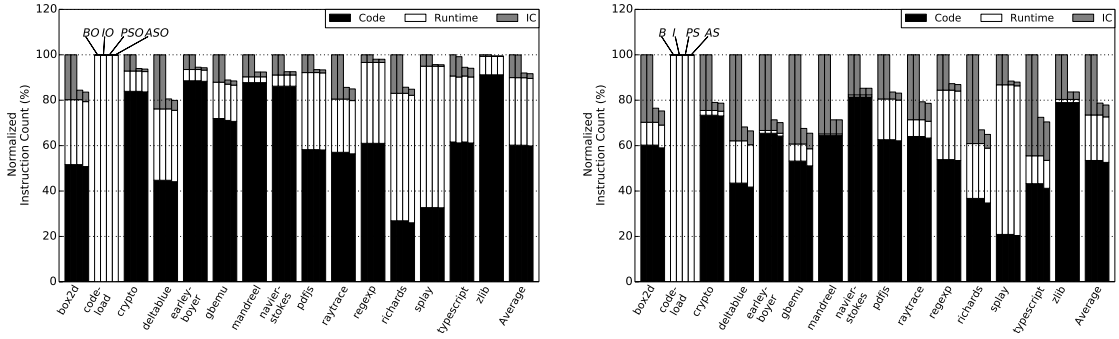


Figure 3.4: Breakdown of dynamic instruction count in Octane with (left) and without (right) the optimizing tier.

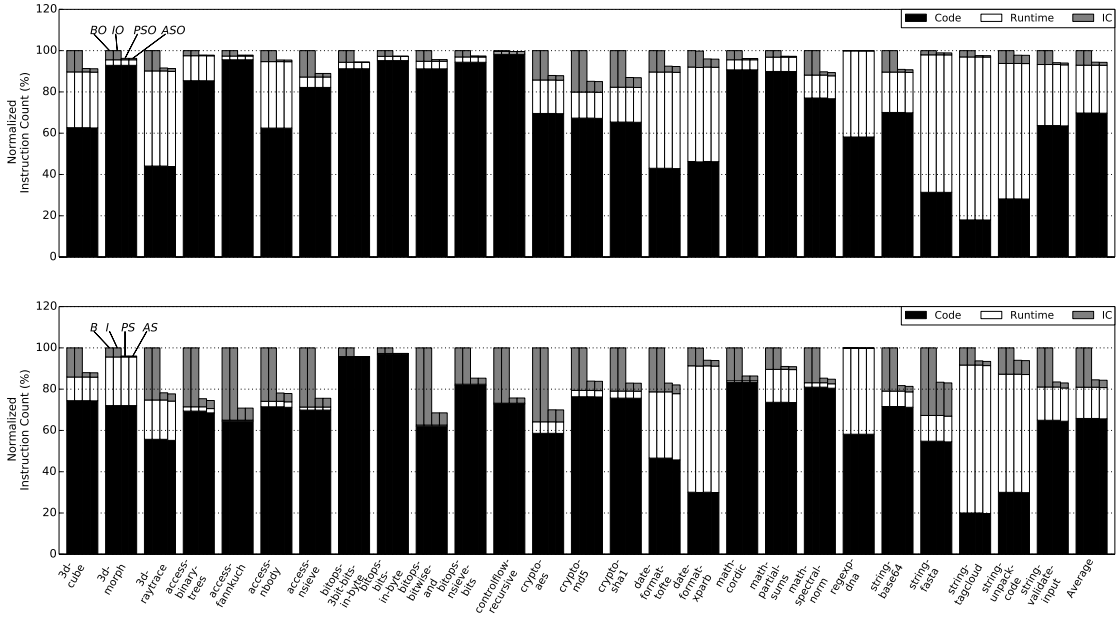


Figure 3.5: Breakdown of dynamic instruction count in SunSpider with (top) and without (bottom) the optimizing tier.

Figures 3.4 and 3.5 show the breakdown of dynamic instruction count in Octane and SunSpider, respectively. Each figure has two charts: one with the optimizer tier on, and one with the optimizer tier off. In a chart, each application has four bars: Baseline configuration (normalized to 1), Ideal, Plain ShortCut, and Aggressive ShortCut.

If we have the optimizing tier enabled (Figures 3.4-left and 3.5-top), the unmodified V8 system (*BO*) executes, on average, 10% and 7% of the instructions in the IC dispatcher for

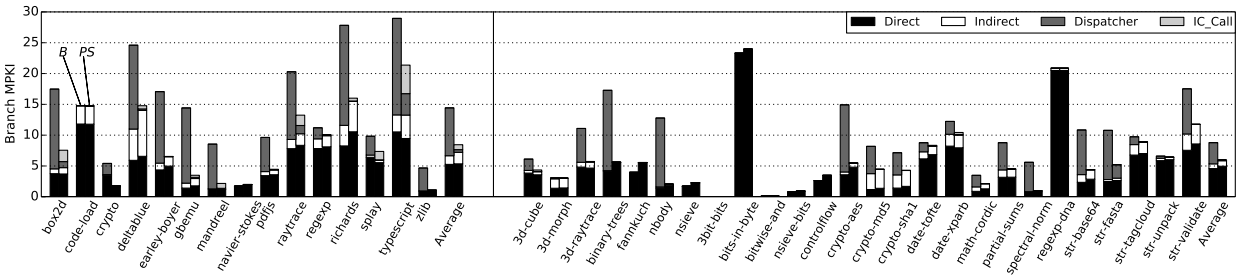


Figure 3.6: Branch MPKI in Octane (left) and SunSpider (right) for *B* and *PS*.

Octane and SunSpider, respectively. Moreover, if we do not have the optimizing tier enabled, (Figures 3.4-right and 3.5-bottom), the unmodified system (*B*) executes, on average, 26% and 19% of the instructions in the IC dispatcher for Octane and SunSpider, respectively. The IC dispatcher executes at least 14 dynamic instructions every invocation, and consequently, the overhead of executing the IC dispatcher is significant. These instructions are the main target of ShortCut.

It can be shown that the indirect jump in the shared dispatcher (`jmp Entry.Handler` in Figure 3.1(d)) is very hard to predict. This is because it has as many different targets as the number of handlers. We measure that the average prediction accuracy of the BTB for this branch is only 42% and 52% for Octane and SunSpider, respectively. As a result, for the unmodified V8's baseline compiler, this branch increases the application-wide average Mispredictions Per Kilo Instruction (MPKI) substantially. Specifically, it can be shown that it increases the application-wide average MPKI in Octane from 6.7 to 14.4, and in SunSpider from 5.4 to 8.8. We will examine the branch behavior of the IC in more detail later.

3.7.2 Impact of ShortCut

Dynamic Instruction Count We now compare the different bars in Figures 3.4 and 3.5. First, the *IO* and *I* configurations have the same instruction count as *BO* and *B*, respectively.

We now consider Plain ShortCut. Without the optimizing tier, (Figures 3.4-right and 3.5-bottom), *PS* reduces the average number of instructions by 21% and 15% in Octane and SunSpider, respectively. This is a substantial reduction. In addition, with the optimizing tier enabled (Figures 3.4-left and 3.5-top), *PSO* still reduces the average number of instructions by 8% and 6% in Octane and SunSpider, respectively.

Aggressive ShortCut improves little over Plain ShortCut. Specifically, *AS* shaves off an average of 2% and 1% of the instructions in *PS* for Octane and SunSpider, respectively. The reason for this small impact is that, as indicated in Section 3.6, Aggressive ShortCut only

optimizes 15% of the load handlers and none of the store handlers. The reduction from *ASO* to *PSO* is even smaller.

Looking at the breakdown of instructions, we see that the reduction from *B* to *PS* (or from *BO* to *PSO*) comes from the *IC* category. ShortCut is avoiding the execution of the dispatcher. *PS* and *PSO*, however, do not completely remove the dispatching overhead (*IC*). The reason is that they still have to execute the dispatcher when they miss in the ICTable. It can be shown that the average miss rate in the 512-entry ICTable is 5.7% .

The magnitude of the reduction in dynamic instructions varies by application, depending on the frequency and predictability of inline caching operations. The `code-load` application (the second application in Figure 3.4) is an extreme case where there is almost no inline cache operation, as it measures compilation overhead and executes in the language runtime (*Runtime*).

Branch Prediction Figure 3.6 shows the branch MPKI in Octane and SunSpider for the *B* and *PS* configurations. We categorize branch instructions into four types: *Direct*, *Indirect*, *Dispatcher* and *IC_Call*. *Direct* is the traditional direct branches, where the target is provided with an immediate operand. *Indirect* is the traditional indirect branches, where the target is provided with a register operand. We count the indirect branch in the shared dispatcher separately, in the *Dispatcher* category, to expose the overhead of inline caching. Lastly, *IC_Call* is ShortCut's new type of indirect branch instruction. Note that *B* has no *IC_Call* category. The figure does not show *I* because *I* is identical to *B* without the *Dispatcher* category. Also, *AS* is not shown because it is practically identical to *PS*.

As shown in the figure, *PS* reduces the average branch MPKI from 14.4 to 8.5 in Octane and from 8.8 to 6.0 in SunSpider. Looking at the breakdown, we see that the reduction in branch mispredictions comes from *Dispatcher*. *PS* rarely executes the *Dispatcher* branch because it hits in the ICTable and avoids the execution of the dispatcher most of the time.

PS introduces *IC_Call* in the figure by executing the *IC_Call* instruction at each access site. This instruction replaces a direct call instruction to the dispatcher in *B*, whose target can be easily predicted by the BTB. On the other hand, *IC_Call* can be mispredicted in some cases, as explained in Table 3.1 — e.g., for polymorphic access sites. However, the average prediction accuracy for *IC_Call* is as high as 98%. Hence, the reduction in *Dispatcher* well justifies the additional mispredictions caused by *IC_Call*.

In the figure, *Direct* and *Indirect* seem to increase for some applications. This is due to the reduction in dynamic instructions, but the absolute number of mispredictions remains same. The BTB's overall hit rate remains roughly the same for all configurations, confirming that ShortCut does not increase the pressure on the BTB.

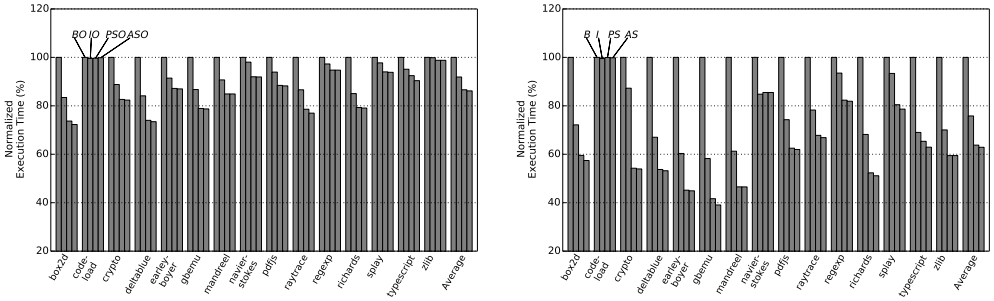


Figure 3.7: Normalized execution time of Octane with (left) and without (right) the optimizing tier.

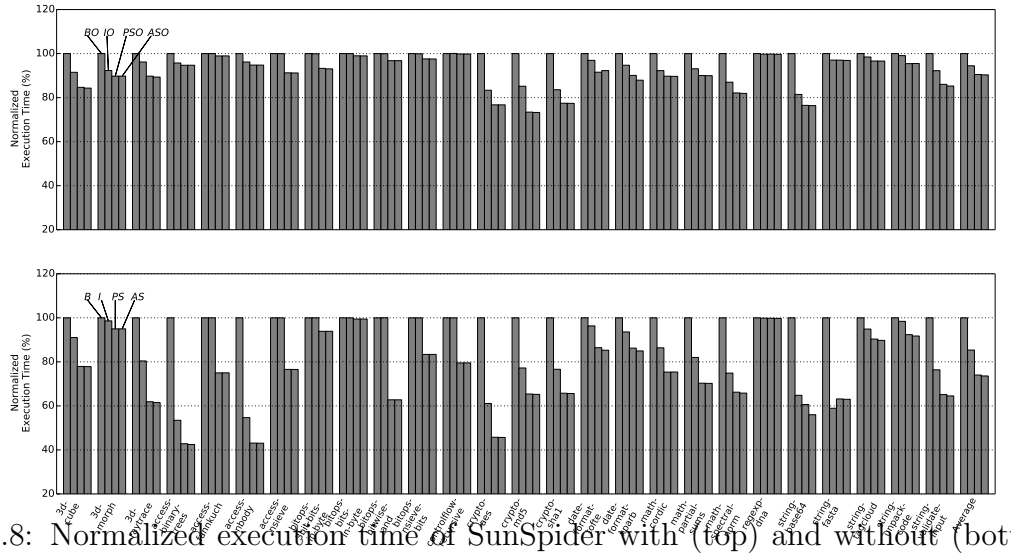


Figure 3.8: Normalized execution time of SunSpider with (top) and without (bottom) the optimizing tier.

Execution Time Figures 3.7 and 3.8 show the execution time of Octane and SunSpider, respectively, for all configurations with and without the optimizing tier. The figures are organized as Figures 3.4 and 3.5. All bars are normalized to the baseline configuration (*BO* in the charts with the optimizing tier, and *B* in the charts without it).

Without the optimizing tier (Figures 3.7-right and 3.8-bottom), *PS* improves the average execution time by 37% and 26% for Octane and SunSpider, respectively, relative to *B*. The improvement comes mostly from the combination of reduced instruction count and enhanced branch prediction, as explained in the previous paragraphs. *AS* further decreases the execution time in some applications (up to 2.6%). However, its average impact is very small. As indicated before, the reason for this small impact is that the current implementation of Aggressive ShortCut only optimizes a small fraction of the handlers.

When the optimizing tier is enabled (Figures 3.7-left and 3.8-top), *PSO* reduces the average execution time by 13% and 10% for Octane and SunSpider, respectively, relative to *BO*. This

is smaller than without the optimizing tier, but still a substantial improvement — especially, given the very highly optimized nature of the Google V8 compiler. The average reduction from *PSO* to *ASO* is negligible.

Lastly, *PSO* and *PS* substantially outperform *IO* and *I*. This shows that having a perfect BTB that always provides the correct target for branches in the IC code structure is no match for our ShortCut optimization.

3.7.3 Sensitivity Study

We perform a sensitivity study by varying the ICTable size at a fixed associativity of 4, for Plain ShortCut. Figure 3.9 shows the average execution time of Octane and SunSpider, respectively, for *PS* with different ICTable sizes relative to *B*.

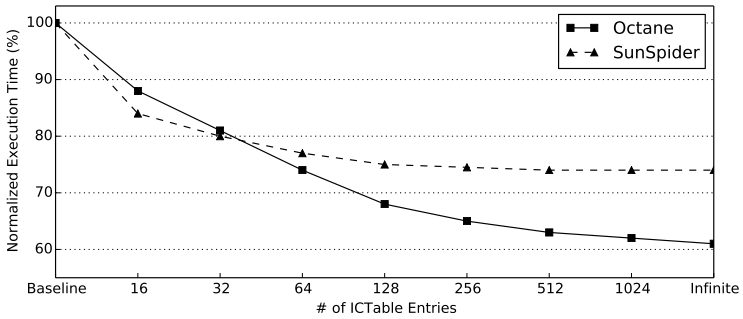


Figure 3.9: Sensitivity of the execution time of *PS* to varying ICTable size. The execution time is normalized to *B*.

Figure 3.9 shows that the performance benefits of Plain ShortCut decrease with smaller ICTable sizes. However, ShortCut outperforms the baseline even with only 16 ICTable entries. Small ICTable sizes such as these are relevant to resource-constrained embedded devices. In these devices, the area overhead is a critical issue.

In addition, we measure the maximum performance benefit possible due to ShortCut hardware by assuming an ICTable with infinite entries. Figure 3.9 shows that it reaches near the maximum performance with 1024 entries. This result justifies our use of a 512-entry ICTable, as it achieves a high speedup while having a reasonable area overhead.

3.8 RELATED WORK

Modern dynamic scripting languages derive key ideas from Smalltalk [20] and Self [19] on how to support dynamic type systems and generate efficient code, most notably inline caching [20, 37].

Ahn et al. [30] reduce the overhead of inline cache miss handling in real-world JavaScript workloads by proposing an alternative type system. In our paper, we instead focus on architectural support to reduce the overhead of inline caching. Our ICTable design is inspired by BTB proposals that improve indirect branch prediction [41, 46, 47]. In particular, our approach of using both the access site address and the object type to index the ICTable resembles the VBBI BTB design [41]. However, we separate the ICTable from the BTB to minimize pipeline intrusion.

There are two hardware proposals to skip the execution of instructions that calculate a dynamic jump target in a way similar to ShortCut. In the first proposal, Agrawal et al. [48] propose a technique to optimize dynamic linking by avoiding the execution of the trampolines for library function calls. Similar to ShortCut, it relies on the BTB to make a prediction of the trampoline target. However, unlike ShortCut, each trampoline in dynamic linking always jumps to the same address. In ICs, instead, the destination of the IC dispatcher code depends on the incoming object type.

The second proposal by Kim et al. [42] reduces the overheads of bytecode dispatching in interpreters by overlaying the bytecode jump table on the BTB. This scheme, however, is not flexible enough to support ICs. First, it operates on a single variable, which is the opcode of the bytecode. An IC operation, instead, requires three variables, namely object type, property name, and access type (load/store). Second, Kim's proposal is limited to improving branch prediction of only one branch instruction, whose address is specified in a special register. In contrast, ShortCut covers all access sites in a program, which include many branches. Overall, to the best of our knowledge, there is no existing proposal with hardware flexible enough to support ICs.

Several hardware proposals exist that address other overheads of dynamic scripting languages. For example, Checked Load [49] extends the ISA to reduce the overhead of checking primitive types. Some proposals improve instruction cache performance on client-side [50] and server-side [51] JavaScript programs. ParaGuard [52] and ParaScript [53] enable parallel execution of JavaScript programs. These proposals are orthogonal to ShortCut, and they could be used together with it.

Dot et al. [54] propose a hardware-software approach to accelerate property loads without using the IC. The idea is to dynamically create a structure in memory with the offset of all the properties of all the objects. A hardware cache caches commonly-used entries from this structure. On a load to a property, special instructions access the cache and return the property offset. While this technique can speed up loads, it is very specific to the design considered. For example, it is unclear how it supports changes in prototype chains, and the various types of handlers required by JavaScript semantics.

Kedlaya et al. propose a technique to combine static type inference and dynamic type feedback mechanisms to optimize dynamically-typed languages [55]. They also propose a technique to enable deoptimization for dynamic language runtimes implemented on top of typed, stack-based virtual machines [56]. These two techniques focus on type specialization based on primitive types (e.g., generating specialized codes for "+" operator differently based on whether operands are integers or strings). They do not support dynamism in object access. On the other hand, ShortCut specifically focuses on supporting IC operations in hardware to minimize the overheads due to dynamic behaviors in object access sites.

Since the submission of this paper, the V8 team has announced their intention to change the compilation tiers within V8 [57]. They plan to replace the baseline compiler with an interpreter named Ignition, and their optimizing compiler with a new implementation named Turbofan. The new version will also extensively use the shared dispatcher design and, hence, would benefit from ShortCut.

3.9 CONCLUSION

Inline caching (IC) is a central feature in dynamic scripting language implementations. This paper proposed architectural support to make IC more efficient. The architecture we proposed, called *ShortCut*, performs two levels of optimization. Its *Plain* design transforms the call to the dispatcher into a call to the correct handler — bypassing the whole dispatcher execution. Its *Aggressive* design transforms the call to the dispatcher into a simple load or store — this time bypassing the execution of both dispatcher and handler. We implemented the ShortCut software modifications in the state-of-the-art Google V8 JIT compiler, and the ShortCut hardware modifications in a Pin-based simulator.

Our evaluation using V8's baseline compiler showed that Plain ShortCut reduces the average application's instruction count by 17%, and its branch MPKI from 10.8 to 6.9. The result is a reduction in the average execution time of the applications by 30%. Under the maximum level of compiler optimization, with the V8 optimizing tier enabled, Plain ShortCut reduces the average execution time of the applications by 11%. Aggressive ShortCut performs only slightly better. The reason is that our current implementation of Aggressive ShortCut only optimizes a small fraction of the handlers. Our future work involves enhancing the capability of Aggressive ShortCut.

CHAPTER 4: BIASED REFERENCE COUNTING: MINIMIZING ATOMIC OPERATIONS IN GARBAGE COLLECTION

4.1 INTRODUCTION

High-level programming languages are widely used today. They provide programmers intuitive abstractions that hide many of the underlying computer system details, improving both programmer productivity and portability. One of the pillars of high-level programming languages is automatic memory management. It frees programmers from the obligation to explicitly deallocate resources, by relying on the runtime to automatically handle memory management.

Garbage collection is the process of runtime monitoring the lifetime of objects and freeing them once they are no longer necessary. There are two main approaches to garbage collection: tracing [58] and reference counting (RC) [59]. Tracing garbage collection maintains a root set of live objects and finds the set of objects reachable from this root set. Objects not reachable from the root set are considered dead and their resources can be freed. RC garbage collection, on the other hand, maintains a counter for each object, which tracks the number of references currently pointing to the object. This counter is actively updated as references are added and removed. Once the counter reaches zero, the object can be collected.

Most implementations of managed languages use tracing garbage collection, as RC is believed to be slower. This belief stems from the fact that most of the tracing garbage collection can be done in the background, off the critical path, while RC is on the critical path. However, many optimization techniques exist to limit the number of RC operations and reduce the overhead on the critical path. Furthermore, RC has the desirable characteristics of low memory overhead and short pause times.

In garbage collection, memory overhead comes from two sources, namely garbage collector metadata, and objects that are dead but not yet reclaimed by the garbage collector. While RC adds, as metadata, one counter per object, RC can have low overall memory overhead because it can be designed to free up objects very soon after they become dead.

Pause times are times when the application is stopped, to allow the garbage collector to perform maintenance operations. RC can be designed to have only short pause times — when individual objects are freed up.

Overall, the combination of low memory overhead and short pause times makes RC suitable for today's interactive mobile platforms. For this reason, some languages such as Swift [60] use RC.

Unfortunately, RC can have significant execution time overhead when using algorithms

that reclaim objects immediately after they become dead — i.e., *non-deferred* RC algorithms. For example, we find that the non-deferred RC algorithm used in Swift causes Swift programs to spend 32% of their execution time performing RC operations. Still, using non-deferred RC is highly desirable when it is critical to keep memory overhead to a minimum, as in many mobile platforms.

We find that a major reason for this execution time overhead of non-deferred RC is the use of atomic operations to adjust the reference counters of objects. Note that even if a Swift program has little sharing and, in fact, even if it is single-threaded, it may have to use atomic operations. This is because, like many programming languages, Swift compiles components separately to allow for maximum modularity. Separate compilation forces the compiler to be conservative. Furthermore, Swift is compiled ahead of time, so the compiler cannot leverage program information gathered throughout execution to limit the use of atomic operations.

The goal of this paper is to reduce the execution time overhead of non-deferred RC. To accomplish this goal, we propose to replace the atomic RC operations with *biased* RC operations. Similar to biased locks [61], our biased operations leverage uneven sharing to create asymmetrical execution times for RC operations based on the thread performing the operation. Each object is biased toward, or favors, a specific thread. This allows an object's favored thread to perform RC operations without atomic operations, at the cost of slowing down all the other threads' RC operations on that object.

While biased RC operations are very effective in most cases, sometimes multiple threads do try to adjust the reference counter of the same object. To handle this, we exploit the fact that, unlike locking, RC does not require strong exclusivity. While only one thread is allowed to acquire a lock at any given time, it is possible for multiple threads to perform RC operations on an object concurrently — if they use multiple counters and eventually merge the counters.

Based on these ideas, we propose a novel algorithm called *Biased Reference Counting* (BRC), which significantly improves the performance of non-deferred RC. BRC maintains two counters per object — one for the owner thread and another for the other threads. The owner thread updates its counter without atomic operations; the other threads update the other counter with atomic operations.

We implement BRC in the Swift runtime. We run various client and server Swift programs and analyze both their performance and sharing patterns. Overall, we find that, on average, BRC improves the execution time of client programs by 22.5%, and the throughput of server programs by 7.3%.

The contributions of this paper are as follows:

• Characterizes the overheads of RC in Swift programs.
• Characterizes the memory behavior and sharing patterns of Swift programs.
• Proposes BRC, a new algorithm to reduce the overhead of non-deferred RC with an efficient biasing technique.
• Implements BRC in the state-of-the-art Swift runtime.
• Provides a detailed evaluation of BRC's performance.

4.2 BACKGROUND

4.2.1 Reference Counting

The fundamental idea of RC [59] is to maintain a per-object counter denoting the current number of references to the object. These per-object counters are updated as references are created, reassigned, and deleted.

Figure 4.1 shows a simple program which highlights all possible RC operations. The normal program commands are on the numbered lines. The RC operations required for each command are on the lines directly above the command in gray and are not numbered. A RC operation on an object `obj` is described by `rc(`*obj*`)`. In this example, the reference counts of objects obj_1, obj_2, and obj_3 are adjusted as different assignments execute.

```
     rc(obj1) = 1
1    var a = new obj1
     rc(obj1)++
2    var b = a
     rc(obj1)--, rc(obj2) = 1
3    b = new obj2
     rc(obj1)--,rc(obj3) = 1
     free(obj1)
4    a = new obj3
```

Figure 4.1: Basic RC operations.

While the idea of RC is straightforward, it should be implemented carefully to ensure correctness. This is because it is possible to have data races while adjusting reference counters in otherwise correct programs. Consider the code example in Figure 4.2. Note that in the traditional sense there is no data race in this code. Each thread only reads shared variable `g`, and all writes are performed to thread-local variables. However, due to both threads adding another reference to *obj*, it is possible for the reference counter of *obj* to be updated incorrectly without synchronization. In other words, it is possible for the `rc(obj)++`

corresponding to the commands on lines 2_A and 2_B to race and produce incorrect results. Hence, updates to an object's reference counter must be done in a synchronized manner.

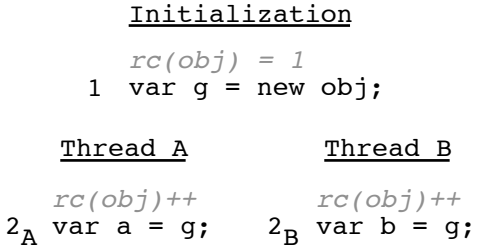


Figure 4.2: Data race due to RC.

There are several approaches to synchronizing RC operations. The most obvious approach is to add locks around the RC operations. This approach has two main drawbacks. First, the runtime must decide the number of locks to create. At one extreme, the runtime can use a single lock for all objects, but this would incur a lot of contention for the lock by threads trying to adjust the reference counters of different objects. At the other extreme, each object can have its own lock. However, this adds an extra overhead to each object header which can result in less locality and more cache misses. Another problem is that there is very little work done in between the lock's acquisition and release – i.e., one simple arithmetic operation to adjust the reference counter. This makes processor stalls likely.

An alternative approach is a lock-free approach using atomic operations as shown in Algorithm 4.1. In this approach, the reference counter is updated using an atomic compare-and-swap (CAS) operation. If the CAS is successful, then the operation is complete. Otherwise, the process is repeated until the operation can complete successfully. This lock-free approach addresses the problems described above for the lock-based approach. First, since there are no locks, there is no tradeoff between the contention for locks versus the memory overhead of locks. Second, this algorithm has only one synchronization operation (the CAS). Because of these benefits, modern RC implementations use lock-free algorithms.

Algorithm 4.1 CAS-based increment operation

```
1: procedure INCREMENT(obj)
2:     do
3:         old := obj.rc_counter                          ▷ read old value
4:         new := old
5:         new += 1                                       ▷ set new value
6:     while !CAS(&obj.rc_counter, old, new)
7:                                                        ▷ Atomic update of counter
8: end procedure
```

4.2.2 RC Optimization

Previous works to optimize RC can be categorized into two groups: works that defer reclamation of objects (deferred RC) and works that do not (non-deferred RC).

Deferred RC These works postpone dead object reclamation, and divide execution into distinct mutation and collection phases. During mutation phases, RC operations are simplified and there is no reclamation of dead objects; during collection phases, the dead objects are reclaimed.

There are two groups of techniques: deferral and coalescing. In deferral [62, 63, 64, 65, 66, 67, 68], the mutation phase does not perform RC operations for local pointer variables stored in the stack or registers. Then, the collection phase scans the stack and registers, and determines the objects that have a reference count equal to zero, and therefore can be reclaimed.

In coalescing [69, 70], the mutation phase only records the modified pointer variables and their initial values. Then, the collection phase compares the initial and final values of the modified pointer variables, and performs RC operations only on the objects pointed to initially and at the end. Dead objects are found and reclaimed during the collection phase.

A hybrid approach [65] uses simple tracing garbage collection for young objects and RC for old objects.

Optimizations of Non-deferred RC These techniques remove unnecessary RC operations through static compiler analysis. Some proposals [71, 72, 73, 74, 75] eliminate the RC operations for a reference R to an object when R's lifetime is completely nested in the lifetime of another reference to the same object. Figure 4.1 shows a simple example of a candidate for this optimization. In this figure, the lifetime of the reference to obj_1 created on line 2 is completely nested in the lifetime of the reference to obj_1 created on line 1. Because of this, it is unnecessary to adjust obj_1's reference counter to reflect `var b`'s effect, so lines two and three need not adjust `rc(`obj_1`)`.

Another optimization [71] is to look at sequential chains of RC operations on the same object, and find matching pairs of increments and decrements (potentially created by different references). These RC operations can also be removed, as they negate one another.

4.2.3 Swift Programming Language

The Swift Programming Language was introduced by Apple in 2014 [76] as an alternative programming language to Objective-C for development for the Apple platform. Due to its

support by Apple and incorporation into the Apple software ecosystem, Swift has quickly become popular and is now the preferred programming language for development on the Apple platform.

Like most modern programming languages, Swift has automatic memory management. Because of its popularity in the mobile environment, where memory overhead is a primary concern, Apple's implementation of Swift uses non-deferred RC and uses the optimizations described in Section 4.2.2. Swift uses weak references to avoid cyclic references, an approach popular in previous literature [77, 78].

4.3 MOTIVATION

4.3.1 Overhead of Reference Counting

To assess the overhead of state-of-the-art non-deferred RC, we measure the time spent performing RC operations in Swift programs. Figure 4.3 shows, for a set of programs, the fraction of time spent on RC operations for each program. The programs we evaluate are explained in detail in Section 4.5. They include client programs (Swift Bench, CryptoSwift, SwiftyJSON, Raytrace, GCBench-Single, GCBench-Multi, and Regex-Redux) and server programs (Perfect-JSON, Perfect-Blog, Kitura-JSON, and Kitura-Blog).

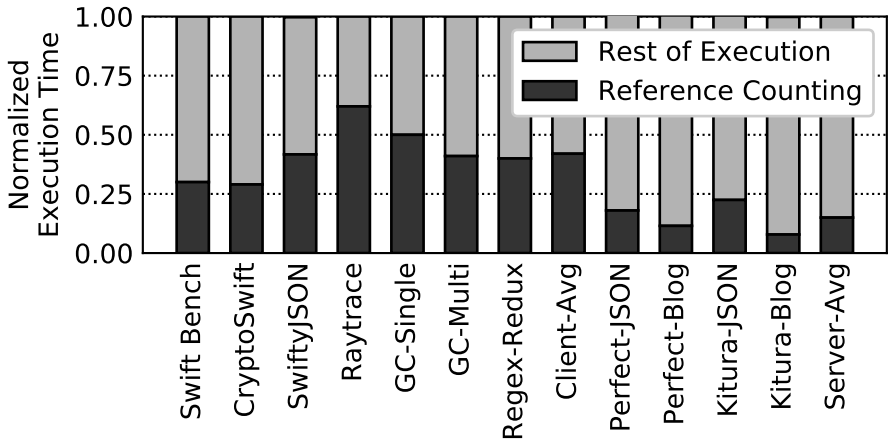


Figure 4.3: Overhead of RC in Swift programs.

As shown in the figure, performing RC operations takes on average 42% of the execution time in client programs, and 15% in server programs. The average across all programs can be shown to be 32%. The Swift compiler does implement optimization techniques to reduce the number of RC operations similar to those described in Section 4.2.2. Without them,

the overhead would be higher. The RC overhead is lower in server programs than in client programs. This is because server programs spend relatively less time in Swift code and RC operations; they spend relatively more time in runtime functions for networking and I/O written in C++.

To estimate the contribution of using atomic operations to this overhead, we remove the CAS operation from the RC code. Hence, the Swift runtime performs RC without safeguards. As explained in Section 4.2.1, not using atomic operations is incorrect, and may result in either memory leaks or objects being prematurely freed. As a result, we are able to run only a subset of the programs without crash. We will show later that, on average, not using atomic operations reduces the average execution time of the programs by 25%. This means that the large majority of the RC overhead in these programs is due to the use of the CAS operations.

This large overhead is due to two reasons. First, CAS instructions are more expensive than normal updates. The reason is that there is a memory fence associated with a CAS instruction. This limits the amount of instruction overlapping performed by the hardware, preventing the out-of-order capabilities of modern cores from being effectively utilized. Second, due to contention, it may be necessary to execute a CAS instruction multiple times before completing successfully. Note that we run all of our experiments on a modern Intel Haswell processor, which has an efficient CAS implementation.

4.3.2 Sharing Patterns of Swift Programs

Since the use of atomic operations greatly affects performance, we evaluate how often in practice they are necessary for correctness. To do so, we modify the runtime so that, for each RC operation, we record which thread is invoking the operation and which object's reference counter is being updated. We classify objects as private or shared. An object is classified as private if all of its reference counter updates throughout its lifetime come from one single thread. Otherwise, the object is classified as shared.

Table 4.1 shows our results. Each row corresponds to a different program. Columns 3 and 4 show the percentage of objects that we classify as private or shared. Columns 5 and 6 show the percentage of reference counter updates that go to objects classified as private or shared. We can see that, on average, over 99% of the objects in client programs, and over 93% of those in server programs are private objects. Similarly, about 93% of the RC operations in client programs, and about 87% of those in server programs are to private objects. This means that the large majority of RC operations are to private objects, and one could skip the corresponding atomic operation. However, as argued in Section 4.1, the Swift compiler

does not know this because Swift compiles components separately. Hence, Swift is forced to use atomic operations always for correctness.

	Program Name	Objects		RC operations	
		Priv (%)	Shar (%)	Priv (%)	Shar (%)
Client	Swift Benchmark	100.00	0.00	100.00	0.00
	CryptoSwift	100.00	0.00	100.00	0.00
	SwiftyJSON	100.00	0.00	100.00	0.00
	Raytrace	100.00	0.00	100.00	0.00
	GCBench-Single	100.00	0.00	100.00	0.00
	GCBench-Multi	99.84	0.16	99.68	0.32
	Regex-Redux	99.99	0.01	51.13	48.87
	Average	99.98	0.02	92.97	7.03
Server	Perfect-JSON	94.74	5.26	83.99	16.01
	Perfect-Blog	94.58	5.42	95.33	4.67
	Kitura-JSON	91.41	8.59	84.29	15.71
	Kitura-Blog	91.59	8.41	83.32	16.68
	Average	93.08	6.92	86.73	13.27

Table 4.1: Sharing patterns of Swift programs.

4.4 BIASED REFERENCE COUNTING

4.4.1 Main Idea

The goal of this paper is to reduce the overhead of non-deferred RC by minimizing the use of atomic operations. We do this with a novel algorithm for RC that we call *Biased Reference Counting (BRC)*. BRC leverages the observation that many objects are only accessed by a single thread. Hence, BRC gives the ownership of, or *biases*, each object to a specific thread. BRC provides two modes of updating an object's reference count: the object's owner thread is allowed to update the reference count using non-atomic operations, while non-owner threads must use atomic operations to update the reference count.

BRC allows these two modes of execution by maintaining *separate counters* for the owner (or biased) thread and for the non-owner threads. The first counter, called the *Biased* counter, counts the number of references to the object added by the owner thread minus those removed by the owner thread. The second counter, called the *Shared* counter, maintains the active reference count for all non-owner threads combined. Since the first counter is only accessed by the owner thread, it can be accessed without atomic operations. The second counter may be accessed by multiple threads concurrently. Therefore, it requires atomic operations to prevent data races. The biasing information is maintained on a per-object

basis. This allows each object to be biased toward the thread most likely to update its reference counter.

In BRC, an object can be deallocated only when the sum of its two counters is zero. Hence, the two counters first need to be merged. Since only the owner thread can read the biased counter reliably, the owner thread is responsible for merging the counters. To merge the counters, the owner thread first atomically accumulates the biased counter into the shared counter. Next, the owner sets a flag to indicate that the two counters have been merged. Once the counters are merged, if the shared counter is zero, the object may be deallocated; otherwise, the owner unbiases the object, and all subsequent reference counter updates will be performed on the shared counter. When the shared counter reaches zero, the object may be deallocated.

In the following, we describe the changes that BRC introduces to the object header, list the invariants in the BRC algorithm, show a few examples of counter transitions, and then describe the BRC algorithm in detail.

4.4.2 Object Header Structure

To support RC, the compiler reserves one word in each object's header, called *RCWord* (for Reference Counting Word). The Swift runtime uses a 64-bit word. Figure 4.4 shows the structure of the RCWord. It has a 30-bit counter to keep track of the number of references to the object. The remaining 34 bits are reserved for weak reference counting and flags to describe the state of the object. Weak references are used to prevent cycles, and are outside of the scope of this paper. To prevent race conditions, accesses to RCWord always use atomic operations.

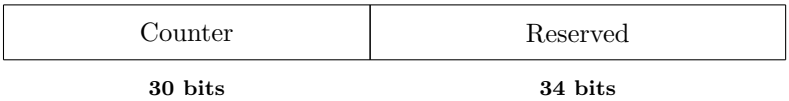


Figure 4.4: Original RCWord.

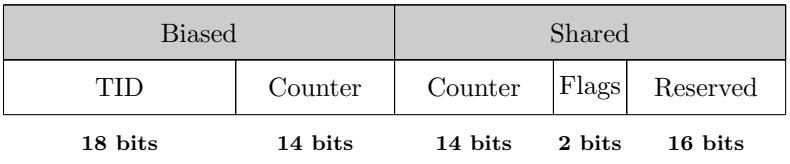


Figure 4.5: BRC's RCWord.

BRC modifies RCWord as shown in Figure 4.5. The new RCWord is now divided into two half-words: *Biased* and *Shared*. The biased half-word contains two fields: the owner thread

identifier (TID) and the biased counter. The TID indicates which thread, if any, the object is currently biased to. This thread has the exclusive right to modify the biased counter.

The shared half-word contains fields shared by all the threads. The shared counter field tracks RC activity by non-owner threads. Then, there are two flags to support the BRC algorithm: `Merged` and `Queued`. The `Merged` flag is set by the owner thread when it has merged the counters. The `Queued` flag is set by a non-owner thread to explicitly request the owner thread to merge counters. More details about the counter operations are explained later.

To prevent race conditions, BRC uses atomic operations to access the shared half-word. In some situations, the BRC algorithm requires multiple fields of the shared half-word to be updated together atomically. This is why the flags must be inside of the shared half-word.

BRC reduces the number of bits per counter from 30 bits to 14 bits, which is more than enough for RC. Many Java programs need only 7 bits [66], and we observe similar behavior in our Swift programs. BRC also reduces the number of the reserved bits used for weak reference counting and existing flags to 16 bits. Weak references occur significantly less frequently than regular RC operations, so this size is acceptable. Alternatively, we could keep the number of bits per counter unchanged to 30 by increasing the size of RCWord at the cost of adding more memory overhead. We evaluate the memory overhead of this alternative design as well in Section 4.6.4.

4.4.3 Algorithm Invariants

To understand the BRC algorithm, we start by describing its main invariants. They are described in Table 4.2. Recall that the value of the counter in the original RCWord (Figure 4.4) reflects the number of current references to the object, and must always be zero or higher. For the same reason, in BRC, invariant *I1* in Table 4.2 says that the sum of the biased and shared counters must always be zero or higher.

Invariant *I2* in Table 4.2 says that the biased counter must always be zero or higher. This is because, as we will show, as soon as the biased counter reaches zero, the owner unbiases the object. This action makes the biased counter inaccessible, and we say it *implicitly merges* the two counters into the shared counter. The owner thread also sets the `Merged` flag.

On the other hand, *I3* says that the shared counter can be negative. This is because a pair of positive and negative updates may be split between the biased and shared counters, pushing the shared counter below zero. As an example, consider two threads T1 and T2. Thread T1 creates an object and sets itself as the owner of it. It points a global pointer to the object, setting the biased counter to one. Then, T2 overwrites the global pointer,

Invariant Description
I1: biased + shared = total number of references to object * Must be zero or higher * If zero, object can be deallocated
I2: biased = (references added - references removed) by owner * Must be zero or higher * When it reaches 0, owner unbiases object, implicitly merging counters
I3: shared = (references added - references removed) by non-owners * Can be negative * If negative, biased must be positive, and object is placed in owner's `QueuedObjects` list so that owner can unbias it
I4: Owner only gives up ownership when it merges counters, namely: * When biased reaches zero (implicit merge) * Or when the owner finds the object in its `QueuedObjects` list (explicit merge)
I5: Object can only be placed into `QueuedObjects` list once * Placed when shared becomes negative for first time * Removed when counters are explicitly merged

Table 4.2: Invariants of the BRC algorithm.

decrementing the shared counter of the object. As a result, the shared counter becomes negative.

When the shared counter for an object becomes negative for the first time, the non-owner thread updating the counter also sets the object's `Queued` flag. In addition, it puts the object in a linked list belonging to the object's owner thread called `QueuedObjects`. Without any special action, this object would leak. This is because, even after all the references to the object are removed, the biased counter will not reach zero — since the shared counter is negative. As a result, the owner would trigger neither a counter merge nor a potential subsequent object deallocation.

To handle this case, BRC provides a path for the owner thread to explicitly merge the counters called the `ExplicitMerge` operation. Specifically, each thread has its own thread-safe `QueuedObjects` list. The thread owns the objects in the list. At regular intervals, a thread examines its list. For each queued object, the thread merges the object's counters by accumulating the biased counter into the shared counter. If the sum is zero, the thread deallocates the object. Otherwise, the thread unbiases the object, and sets the `Merged` flag. Then, when a thread sets the shared counter to zero, it will deallocate the object. Overall, as shown in invariant *I4*, an owner only gives up ownership when it merges the counters.

Invariant *I5* in Table 4.2 says that an object can be placed into `QueuedObjects` list only once. It is placed there when its shared counter becomes negative for the first time. After that, while its shared counter may continue to change, since the object is already marked as

queued, no action is required. It will remain in the owner's `QueuedObjects` list until the owner unbiases it.

4.4.4 Examples of Counter Transitions

Figure 4.6 shows some examples of RCWord transitions in BRC. To start with, Figure 4.6(a) shows the RCWord structure without the Reserved field. Then, in Figure 4.6(b), we show the RCWord transitions for a private (i.e., thread-local) object. In this example, thread T1 allocates the object and becomes the owner thread. Next, T1 creates up to N references to the object, incrementing the biased counter up to N. Finally, T1 removes these references, decrementing the biased counter to zero, and deallocates the object.

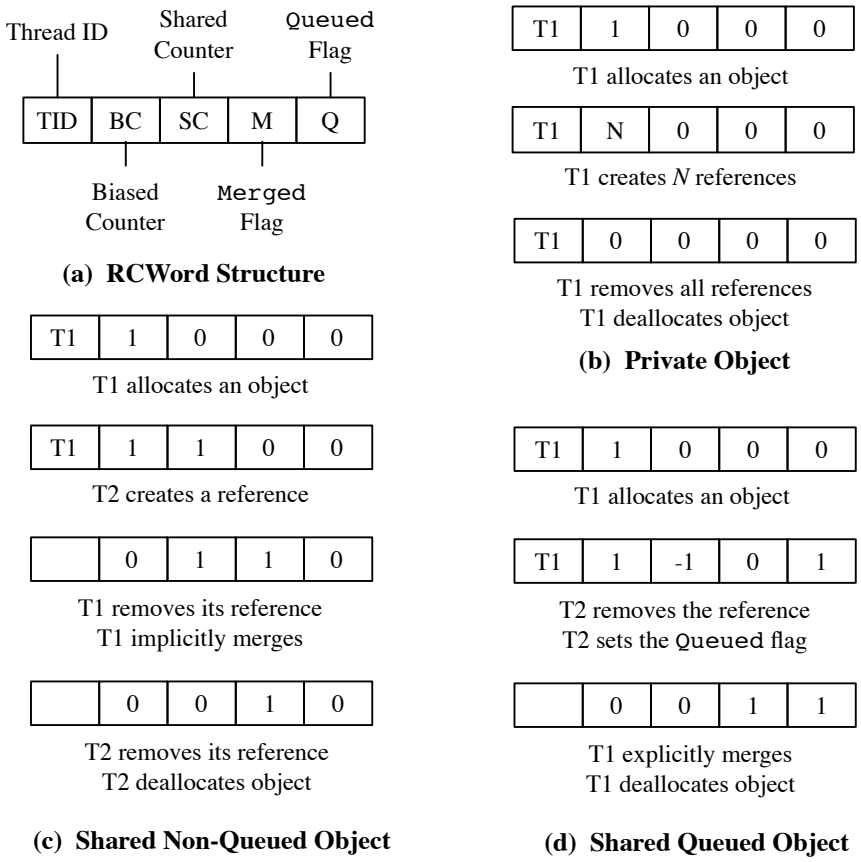


Figure 4.6: Examples of RCWord transitions.

In Figure 4.6(c), we show the RCWord transitions for a shared object that is not queued in a `QueuedObjects` list during its lifetime. Thread T1 first allocates the object and sets itself as the owner of it. Next, a second thread T2 creates a reference to the object, incrementing the shared counter. Then, T1 removes its reference to the object, decrementing

the biased counter. As the biased counter becomes zero, T1 performs an implicit counter merge: it sets the `Merged` flag and unbiases the object. Later, T2 removes its reference to the object, decrementing the shared counter. Since the shared counter is zero and the `Merged` flag is set, T2 deallocates the object.

In Figure 4.6(d), we show the RCWord transitions for a shared object that is queued in a `QueuedObjects` list during its lifetime. Thread T1 first allocates the object and sets itself as the owner of it. Then, thread T2 overwrites the reference to the object and hence decrements the shared counter. Since the shared counter becomes negative, T2 also sets the `Queued` flag and places the object in T1's `QueuedObjects` list. Later, T1 invokes the `ExplicitMerge` operation and explicitly merges the counters, setting the `Merged` flag and unbiasing the object. Since the sum of the counters is zero, T1 deallocates the object.

4.4.5 BRC Algorithm

The BRC algorithm introduces several changes to a conventional RC algorithm. First, when an object is allocated, BRC saves the ID of the thread allocating the object in the RCWord, effectively biasing the object. Second, BRC modifies the RC operations (i.e., `Increment` and `Decrement`) to update one of the two RCWord counters based on which thread an object is biased to. Finally, BRC adds two new operations, `Queue` and `ExplicitMerge`, to handle a special case introduced by using two counters. In the following paragraphs, we explain these operations in detail. We use a dot notation to access the biased and shared half-words, and their fields in the algorithms. Note that the algorithms given below sacrifice performance for maximum clarity. BRC's implementation on the Swift runtime is more efficient than what is shown here.

Algorithm 4.2 shows BRC's `Increment` operation. It begins by checking whether the new reference is being created by the object's owner thread (line 4). If so, the owner thread continues to the `FastIncrement` procedure to increment the biased counter (line 11). Otherwise, a non-owner thread calls `SlowIncrement` and uses an atomic CAS operation to increment the shared counter (line 19).

Algorithm 4.3 shows BRC's `Decrement` operation. Similar to `Increment`, it first checks whether the reference is being removed by the object's owner thread (line 4). If so, the owner thread continues to the `FastDecrement` procedure to decrement the biased counter (line 11). If the resulting value of the counter is higher than zero (line 13), no further action is required. Otherwise, the biased counter is zero, and the owner thread performs an implicit merge of the counters. Specifically, it sets the `Merged` flag (line 19) by atomically updating the shared half-word (line 20). Next, the shared counter is read. If its value is

Algorithm 4.2 Increment operation

```
 1: procedure INCREMENT(obj)                          ▷ Increment the reference count of obj
 2:     owner_tid := obj.rcword.biased.tid
 3:     my_tid := GetThreadID()
 4:     if owner_tid == my_tid then
 5:         FastIncrement(obj)                        ▷ Owner access
 6:     else
 7:         SlowIncrement(obj)                        ▷ Non-owner access
 8:     end if
 9: end procedure

10: procedure FASTINCREMENT(obj)
11:     obj.rcword.biased.counter += 1
12:                                                  ▷ Non-atomic increment of biased counter
13: end procedure

14: procedure SLOWINCREMENT(obj)
15:     do
16:         old := obj.rcword.shared                  ▷ Read shared half-word
17:         new := old
18:         new.counter += 1
19:     while !CAS(&obj.rcword.shared, old, new)
20:                                                  ▷ Atomic increment of shared counter
21: end procedure
```

zero (line 22), the object is deallocated. Otherwise, BRC unbiases the object by clearing the owner TID (line 25). Now, all future RC operations to this object will invoke either the `SlowIncrement` or the `SlowDecrement` procedures. In addition, any thread can make the decision to deallocate the object. Note that the `Deallocate` call in line 23 does not need to lock the object. This is because the last reference has been removed so no other thread can access the object.

If the `Decrement` operation is invoked by a non-owner thread, it continues to the `SlowDecrement` procedure (line 28). BRC decrements the shared counter (line 32) and, if the counter's new value is negative, BRC also sets the `Queued` flag (line 34). The shared half-word is updated atomically (line 36). If the `Queued` flag has been set for the *first* time by this invocation (line 38), BRC invokes function `Queue` to insert the object in a list to be handled later by the owner (line 40) — note that this case implies that the counters have not been merged yet, as the shared counter's value is negative. Otherwise, if the `Merged` flag is set and the shared counter is zero (line 41), BRC deallocates the object.

BRC adds two new operations, `Queue` and `ExplicitMerge` (Algorithm 4.4), to support a special case introduced by having two counters. Specifically, the first time that the shared counter attains a negative value, `Queue` is invoked. As indicated in Section 4.4.3, at this point, the biased counter has a positive value. If BRC did not take any special action, the

Algorithm 4.3 Decrement operation

```
1: procedure DECREMENT(obj)                                  ▷ Decrement the reference count of obj
2:     owner_tid := obj.rcword.biased.tid
3:     my_tid := GetThreadID()
4:     if owner_tid == my_tid then
5:         FastDecrement(obj)                                ▷ Owner access
6:     else
7:         SlowDecrement(obj)                                ▷ Non-owner access
8:     end if
9: end procedure

10: procedure FASTDECREMENT(obj)
11:     obj.rcword.biased.counter −= 1
12:                                                          ▷ Non-atomic decrement of biased counter
13:     if obj.rcword.biased.counter > 0 then
14:         return
15:     end if
16:     do                                                   ▷ biased counter is zero
17:         old := obj.rcword.shared                         ▷ Read shared half-word
18:         new := old
19:         new.merged :=True                                ▷ Set merged flag
20:     while !CAS(&obj.rcword.shared, old, new)
21:                                                          ▷ Atomic update of shared half-word
22:     if new.counter == 0 then
23:         Deallocate(obj)
24:     else
25:         obj.rcword.biased.tid := 0                       ▷ Give up ownership
26:     end if
27: end procedure

28: procedure SLOWDECREMENT(obj)
29:     do
30:         old := obj.rcword.shared                         ▷ Read shared half-word
31:         new := old
32:         new.counter −= 1
33:         if new.counter < 0 then
34:             new.queued :=True                            ▷ Set queued flag
35:         end if
36:     while !CAS(&obj.rcword.shared, old, new)
37:                                                          ▷ Atomic decrement of shared counter
38:     if old.queued ≠ new.queued then
39:                                                          ▷ queued has been first set in this invocation
40:         Queue(obj)
41:     else if new.merged ==True and new.counter == 0 then
42:                                                          ▷ Counters are merged and shared counter is zero
43:         Deallocate(obj)
44:     end if
45: end procedure
```

biased counter might never be decremented to zero and, thus, the counters might never be merged, and the object might never be deallocated. This is a memory leak.

To guard against such scenarios, BRC keeps track of objects that may leak. As shown in the `Queue` procedure of Algorithm 4.4, the non-owner thread that first finds that the shared counter becomes negative, inserts the object in a thread-safe list belonging to the object's owner thread. The list is part of a structure called `QueuedObjects` (line 3), which is organized as per-thread lists of potentially leaked objects. Potentially leaked objects are added to the `QueuedObjects` list belonging to the object's owner thread.

Algorithm 4.4 Extra operations

```
 1: procedure QUEUE(obj)
 2:     owner_tid := obj.rcword.biased.tid
 3:     QueuedObjects[owner_tid].append(obj)
 4:                                          ▷ Adds object to list belonging to owner_tid
 5: end procedure

 6: procedure EXPLICITMERGE
 7:     my_tid := GetThreadID()
 8:     for all obj ∈ QueuedObjects[my_tid] do
 9:         do
10:             old := obj.rcword.shared                    ▷ Read shared half-word
11:             new := old
12:             new.counter += obj.rcword.biased.counter
13:                                                         ▷ Merge counters
14:             new.merged :=True
15:         while !CAS(&obj.rcword.shared, old, new)
16:                                          ▷ Atomic update of shared half-word
17:         if new.counter == 0 then
18:             Deallocate(obj)
19:         else
20:             obj.rcword.biased.tid := 0                  ▷ Give up ownership
21:         end if
22:         QueuedObjects[my_tid].remove(obj)
23:     end for
24: end procedure
```

At regular intervals, a thread checks its `QueuedObjects` list, to explicitly merge counters and enable object deallocation. The `ExplicitMerge` procedure of Algorithm 4.4 performs this operation. The procedure searches through the thread's `QueuedObjects` list and, for each object, explicitly merges its two counters. Note that this merging can only be done by the owner thread, so the procedure only accesses the `QueuedObjects` list owned by the thread invoking the procedure (line 8). For each object in the list, BRC accumulates the biased counter into the shared counter (line 12) and sets the `Merged` flag (line 14). This change is atomic (line 15). If the merged counter becomes zero, the owner deallocates the object (line 18). Otherwise, it unbiases the object (line 20) so that all future RC operations

Name	Configuration
O	Original: The unmodified Swift runtime
I	Ideal: O with no atomic operations
B	Biased: O enhanced with BRC

Table 4.3: Configurations evaluated.

are performed on the shared counter. Once this merging is completed, it is no longer possible for the object to be leaked, and thus the owner removes the object from `QueuedObjects` in a thread-safe manner (line 22).

A given object can only be put in the `QueuedObjects` list once. This is because, before an object is taken out of the list, its counters are merged. Such merging eliminates the possibility that the shared counter become negative anymore.

Lastly, when a thread terminates, it processes the objects remaining in its `QueuedObjects` list, and de-registers itself from the `QueuedObjects` structure. Theoretically, an object can outlive its owner thread if its biased counter is positive, and has not been queued in the `QueuedObjects` list when the owner thread terminates. We handle this case as follows. When a non-owner thread makes the shared counter of an object negative, it first checks whether the object's owner thread is alive by looking-up the `QueuedObjects` structure — which implicitly records the live threads. If the owner thread is not alive, the non-owner thread merges the counters instead of queuing the object, and either deallocates the object or unbiases it.

4.5 EXPERIMENTAL SETUP

To evaluate BRC, we implement it in the Swift version 3.1.1 runtime. We evaluate the three configurations shown in Table 4.3. The *Original* configuration (O) is the unmodified Swift runtime, which implements RC with lock-free atomic operations. The *Ideal* configuration (I) takes O and eliminates all the atomic operations. In this configuration, due to data races, counters may have incorrect values. In particular, an object may be accessed after being deallocated, which may lead to a crash. We collect data from I only when the program runs to completion, and its output and number RC operations are same as in O's execution. This ensures that I did not change semantics. Lastly, the biased configuration (B) is O enhanced with BRC. As a result, all of Swift's RC optimizations (which are present in O) are enabled in B by default.

Table 4.4 shows the client and server programs that we evaluate. The official Swift Benchmark Suite [79] consists of a set of tests which cover important Swift workloads. The suite

	Program Name	Multi-threaded?	Description
Client	Swift Benchmark	No	A set of 212 benchmarks covering a number of important Swift workloads designed to track Swift performance and catch performance regressions
	CryptoSwift	No	Performance tests of a Swift package for cryptography algorithms
	SwiftyJSON	No	Performance tests of a Swift package for JSON handling
	Raytrace	No	Ray tracing application
	GCBench-Single	No	Single-threaded implementation of an artificial garbage collection benchmark that creates perfect binary trees
	GCBench-Multi	Yes	Multi-threaded implementation of GCBench
	Regex-Redux	Yes	Benchmark that uses regular expressions to match and replace DNA 8-mers
Server	Perfect-JSON	Yes	JSON generator running on the Perfect framework
	Perfect-Blog	Yes	Blog engine running on the Perfect framework
	Kitura-JSON	Yes	JSON generator running on the Kitura framework
	Kitura-Blog	Yes	Blog engine running on the Kitura framework

Table 4.4: Client and server programs used.

is designed to track Swift performance and catch performance regressions. CryptoSwift [80] and SwiftyJSON [81] are popular Swift packages for cryptography and JSON handling, respectively. We also use a Swift version of ray tracing [82]. GCBench is an artificial garbage collection benchmark which creates and discards perfect binary trees to estimate the collector performance. We use single-threaded and multi-threaded implementations of GCBench. Lastly, Regex-Redux is a regular expression benchmark that uses regular expressions to match and replace DNA sequences.

Our server programs are based on two popular server-side frameworks for Swift, namely Perfect [5] and Kitura [6]. For each framework, we run a blog engine that returns random images and blog posts for each request, and a JSON generator that returns a JSON dictionary of random numbers for each request [83]. For the server programs, we measure throughput instead of execution time.

We run our experiments on a desktop machine with an Intel Core i7 processor and 16 GB of memory running Ubuntu 16.04 LTS. The processor has four cores cycling at 3.50 GHz. Each experiment is run 10 times and the average is reported.

	Program Name	% of Shared Obj.	Original			Biased					
			Obj. Allocs. per μs	RC Ops. per μs	RC Ops. per Obj.	RC Ops. per μs	% of RC Ops. to Shared obj.	% of RC Ops. to Biased Counter	% of RC Opts. to Shared Counter	% of Queued Obj.	% of RC Ops. Setting `Queued` Flag
Client	Swift Benchmark	0.00	1.91	29.18	15.24	33.16	0.00	100.00	0.00	0.00	0.00
	CryptoSwift	0.00	2.39	56.54	23.61	68.16	0.00	100.00	0.00	0.00	0.00
	SwiftyJSON	0.00	2.60	68.19	26.24	93.77	0.00	100.00	0.00	0.00	0.00
	Raytrace	0.00	0.00	101.70	27258.42	173.94	0.00	100.00	0.00	0.00	0.00
	GCBench-Single	0.00	12.87	64.07	4.98	86.03	0.00	100.00	0.00	0.00	0.00
	GCBench-Multi	0.16	50.69	150.39	2.97	195.44	0.32	99.97	0.03	0.01	0.00
	Regex-Redux	0.01	2.58	92.61	35.91	123.99	48.87	88.02	11.98	0.00	0.00
	Average	0.02	10.44	80.38	39338.19	110.64	7.03	98.28	1.72	0.00	0.00
Server	Perfect-JSON	5.26	0.55	4.56	8.25	4.95	16.01	88.64	11.36	1.99	0.24
	Perfect-Blog	5.42	0.45	12.57	27.95	12.90	4.67	96.72	3.28	1.95	0.07
	Kitura-JSON	8.59	0.40	7.27	18.05	7.55	15.71	87.38	12.62	2.76	0.15
	Kitura-Blog	8.41	0.39	6.12	15.81	6.34	16.68	86.68	13.32	2.70	0.17
	Average	6.92	0.45	7.63	17.51	7.93	13.27	89.85	10.15	2.35	0.16

Table 4.5: Reference counting statistics.

4.6 EVALUATION

4.6.1 Characterization

We start by investigating the overhead of RC in the Original (O) and Biased (B) configurations. Table 4.5 shows various metrics of RC behavior during execution for both configurations. For reference, Column 3 repeats the data shown in Table 4.1 about the percentage of shared objects in each program. Recall that we consider an object as shared if its reference counter updates come from more than one thread. Next, Columns 4-6 refer to the O configuration, while columns 7-12 refer to the B configuration.

Columns 4 and 5 show the number of object allocations per μsecond and the number of RC operations per μsecond, respectively. The latter are counter increments and decrements. Based on the data in these two columns, Column 6 shows the average number of RC operations per object. We can see that, discounting Raytrace, there are 3–36 RC operations per object in client programs, and 8–28 in server programs.

Column 7 shows the number of RC operations per μsecond in the B configuration. Due to the improved performance of B, these numbers are higher than in O for all the programs.

Column 8 shows the percentage of RC operations to shared objects, and Columns 9 and 10 the percentage of RC operations to the biased and shared counters, respectively. We see that only a small percentage of the RC operations are performed on shared objects (7.03% in client programs and 13.27% in server programs), and an even smaller percentage are performed on shared counters (1.72% in client programs and 10.15% in server programs). The outlier is Regex-Redux, where nearly 50% of the RC operations are on shared objects, and 12% use the shared counter. Overall, the small fraction of the RC operations that use the shared counter is the reason for the speed-ups of B over O; only such operations use atomic instructions.

Configuration	Time (ns)
Original	13.84
Ideal	5.77
Biased (operation by owner)	6.28
Biased (operation by non-owner)	15.57

Table 4.6: Time of counter increment operations.

Column 11 shows the percentage of the total objects that are queued. On average, this number is 0.00% in client programs, and 2.35% in server programs. This number is very small, in part because the percentage of objects that are shared (Column 3) is already small. Finally, Column 12 shows the percentage of RC operations that set the `Queued` flag and add the object to the `QueuedObjects` list. We see that this is a rare event, which occurs 0.00% of the time in client programs, and 0.16% in server programs. Overall, queuing in the `QueuedObjects` list is a negligible overhead.

4.6.2 Latency of RC Operations

We measure the time it takes to increment a reference counter in the different configurations. For this measurement, we create kernels that repeatedly increment the counter in a loop. Therefore, the operations have a near-perfect cache behavior. In addition, these kernels are single-threaded and, therefore, the measured times do not include contention. Overall, our experiments measure best-case timings.

Table 4.6 shows the time to perform a counter increment in our different configurations: O, I, and B. For the B configuration, we show the operation time for the owner thread and for non-owner threads. As shown in the table, the increment operation takes 13.84 ns in O and 5.77 ns I. Hence, the use of atomic operations slows down the operation by 2.40x. In B, the owner's increment takes only 6.28 ns, while the non-owner increment takes 15.57 ns. Ideally, the former should be as fast as I, while the latter should take as long as O. In practice, BRC adds some overhead to each of these operations, as the TID and various flags are checked before performing the increment. Consequently, B owner takes 8.8% longer than I, and B non-owner takes 12.5% longer than O.

4.6.3 Performance Improvement

In this section, we evaluate the performance improvements attained by BRC. Figure 4.7 shows the execution time of the client programs for the O, I, and B configurations, normal-

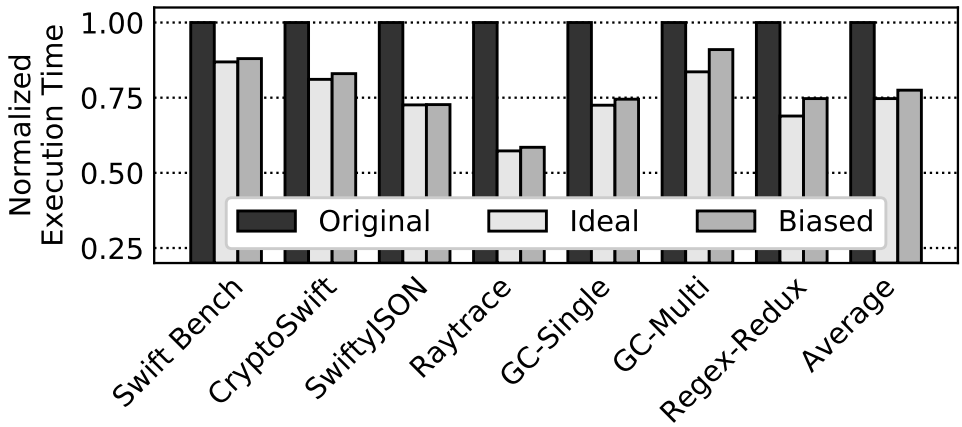


Figure 4.7: Execution time of client programs under the O, I, and B configurations.

ized to the O configuration. On average, B reduces the execution time by 22.5% over O. This is a substantial speed-up, which is attained inexpensively in software by improving the RC algorithm. Further, this speed-up implies that a large fraction of the RC overhead in O has been eliminated. Indeed, as shown in Figure 4.3, client programs spend on average 42% of their time in RC operations. With B, it can be shown that we eliminate more than half of such RC time.

We see that, on average, the B configuration is within 3.7% of the I configuration. This difference is smaller than the 8.8% difference observed in Table 4.6 between the B (owner) and I increment operations. This is due to Amdahl's law, as programs only spend a fraction of their time performing RC operations.

Figure 4.8 shows the throughput of the server programs under the O and B configurations, normalized to the O configuration. We do not show data for the I configuration because running these programs without atomic operations causes frequent program crashes due to premature object deallocations. The figure shows that B attains a substantial average throughput increase of 7.3% over O. This improvement is smaller than the 22.5% average reduction in the execution time of the client programs. This is expected, given that the overhead of RC in Figure 4.3 is higher in the client programs than in the server ones.

4.6.4 Memory Overhead

In this section, we evaluate BRC's memory overhead by comparing the peak memory usage of the O and B configurations. Figure 4.9 shows the peak memory usage of these configurations normalized to the peak memory usage of O. Recall from Section 4.4.2 that

Figure 4.8: Throughput of the server programs under the O and B configurations.

our BRC design does not increase the size of the per-object RCWord. Hence, the additional memory overhead of B comes from the use of the `QueuedObjects` structure.

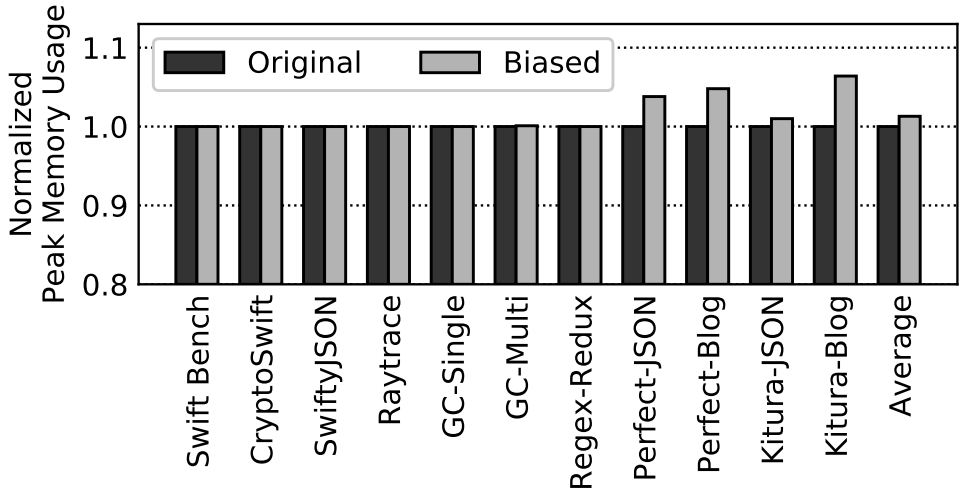


Figure 4.9: Peak memory usage under the O and B configurations.

We observe that, on average, B has only a 1.5% higher memory overhead than O. Single-threaded programs (i.e., the first 5 programs) have no additional memory overhead in B because the shared counter is not used and, consequently, there are no queued objects. While GCBench-single and Regex-Redux are multithreaded, they have negligible additional memory overhead because they have almost no queued objects (Column 11 of Table 4.5). The server programs have only a small fraction of queued objects and, therefore, their additional memory overhead is on average about 4%.

We also measure the memory overhead in the alternative B implementation described

in Section 4.4.2, where we add an additional 64-bit word to the object header to preserve 30-bit counters. In this case, the peak memory usage in B can be shown to be, on average, a modest 6% higher than in O.

4.6.5 Sensitivity Study

To simulate a worst-case scenario for BRC, we create a synthetic benchmark where we can control the number of queued objects. In the benchmark, a main thread creates 1,000,000 objects, creating a reference to each object, then performs a fixed amount of dummy computation, and finally removes any remaining references to the objects. In parallel, a second thread removes the references to $1,000,000 \times R$ objects allocated by the main thread. When a shared counter becomes negative, the second thread adds the corresponding object to the main thread's `QueuedObjects` list. In our experiments, we vary R, which we call Ratio of Queued Objects. Figure 4.10 shows the execution time and peak memory usage under the B configuration as we vary R, normalized to the O configuration.

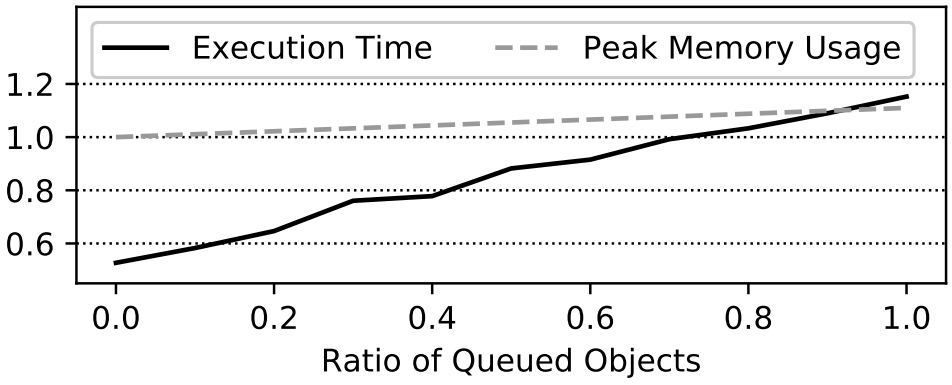


Figure 4.10: Normalized execution time and normalized peak memory usage of the B configuration as we vary the number of queued objects.

Our results show that, for B to perform worse than the O configuration, one needs 75% or more queued objects. In reality, as shown in Table 4.5, the percentage of queued objects in our programs is much lower than this break-even point.

We also measure the additional memory overhead of B for this benchmark. As we see in the figure, the additional memory overhead is kept low, under 12% over the O configuration. This is because the counter merging and object dequeuing happen frequently enough that reclamation of dead queued objects is not delayed too much. Note that, for our programs in Table 4.5, the percentage of queued objects is very small and, therefore, the additional memory overhead of BRC is small.

4.7 DISCUSSION

4.7.1 Putting BRC in Context

In this section, we qualitatively compare BRC to other RC algorithms. Table 4.7 examines the space-time trade-off of various RC implementations. Each row corresponds to a different RC implementation. The table ranks the RC implementations from 1 (lowest) to 4 (highest) in terms of performance overhead and memory overhead.

Algorithm	Performance Overhead	Memory Overhead
Basic non-deferred RC	4	1 (tie)
Non-deferred RC w/ optimization	3	1 (tie)
Deferred RC (DRC)	1	3
BRC	2	2

Table 4.7: Ranking performance and memory overheads of RC implementations from 1 (lowest) to 4 (highest).

The first row corresponds to the basic non-deferred RC described in Section 4.2.1. It suffers from a high execution time overhead due to frequent atomic RC operations. However, it has a minimal memory overhead thanks to immediate reclamation.

The second row corresponds to the non-deferred RC with the optimization described in Section 4.2.2. This is Swift's RC implementation. Compared to basic non-deferred RC, the execution time overhead is dramatically reduced. This is because many unnecessary RC operations are removed at compile time. Specifically, we found that Swift removes up to 97% of RC operations in our programs. This implementation is very effective at removing RC operations for local variables. At the same time, it maintains immediate reclamation, and hence has the same minimal memory overhead as the basic non-deferred RC.

The third row corresponds to deferred RC (DRC) implementations, as described in Section 4.2.2. The performance overhead is lower, as deferral and coalescing avoid atomic RC operations during the mutation phase. However, since DRC does not perform immediate reclamation for all objects, the memory overhead is higher than the basic non-deferred RC.

The last row corresponds to BRC. While Swift's non-deferred RC with optimization is fast, it is still slower than DRC (about 20% [71]). BRC narrows this performance gap by replacing atomic RC operations with non-atomic ones in most cases. It also retains immediate reclamation for most objects in our programs. Hence, it increases the memory overhead *very little* compared to the basic non-deferred RC. We discuss BRC's impact on performance and memory in Section 4.6.3 and 4.6.4 in detail.

Overall, we believe that BRC enables a new space-time trade-off in the RC design space, different from what has been proposed thus far. Further, we believe that BRC aligns well with Swift's philosophy that emphasizes speed and low memory consumption.

4.7.2 Ownership Transfer

In this section, we discuss the ownership transfer especially regarding assigning a new owner after the initial owner gives up the ownership. Note that the current BRC algorithm never transfers ownership after it assigns the initial owner. The initial owner of an object is set to the thread allocating the object. After the initial ownership assignment, the only time the ownership of the object is changed is when the the owner gives up the ownership either by implicit or explicit merges as described in Section 4.4.5. After this point, the object has no owner and all threads access the shared counter.

We extend `SlowIncrement` to enable a new ownership assignment as shown in Algorithm 4.5. Lines 2-11 are added to assign a new owner if the object has no owner. It first checks whether the object has an owner (line 3). If not, it prepares a new biased half-word with the current thread as the owner (lines 6) and tries to set the biased half-word atomically (line 8). If successful, it returns, and otherwise, it continues to the original part of `SlowIncrement` to update the shared half-word (lines 12-18). `SlowDecrement` is extended similarly.

Algorithm 4.5 SlowIncrement with new owner assignment

```
 1: procedure SlowIncrement(obj)
 2:     owner_tid := obj.rcword.biased.tid
 3:     if owner_tid == 0 then
 4:         old := 0                                  ▷ Current biased half-word should be 0
 5:         new := old
 6:         new.tid := GetThreadID()                  ▷ Assign new owner
 7:         new.counter := 1                          ▷ Set biased counter
 8:         if CAS(&obj.rcword.biased, old, new) then ▷ Atomic update of biased half-word
 9:             return
10:         end if
11:     end if
12:     do
13:         old := obj.rcword.shared                  ▷ Read shared half-word
14:         new := old
15:         new.counter += 1
16:     while !CAS(&obj.rcword.shared, old, new)
17:                                                   ▷ Atomic increment of shared counter
18: end procedure
```

The performance impact of such extension is negligible in our programs (less than 1%

slowdown). This is because the sharing patterns in our programs do not require ownership transfer as the initial owner of an object tends to drop the last reference to the object and deallocate the object. In other words, in most cases, an object is deallocated when its biased counter reaches zero in our programs. The minor slowdown comes from extra checks added to `SlowIncrement` and `SlowDecrement` to assign a new owner.

As different kinds of programs, exhibiting different sharing patterns, are developed in Swift in the future, it may be worthwhile to reconsider the potential ownership transfer. As shown in Algorithm 4.5, BRC is flexible to support various kinds of ownership transfer.

4.8 RELATED WORK

There have been many works [61, 84, 85, 86, 87, 88, 89] which try to limit the amount of overhead to acquire uncontested locks. While BRC is inspired by biased locking [61], it is not a straightforward re-application of biased locking. BRC proposes an efficient biasing technique tailored to RC by exploiting the fact that *RC does not require strong exclusivity like locking*. BRC lets multiple threads access the same object concurrently, by dividing an object's reference count into two counters. In biased locking, this is not possible. BRC also makes ownership revocation very cheap. This is because ownership is typically voluntarily revoked in BRC and only requires one CAS. On the other hand, ownership revocation is extremely expensive in biased locking. It is triggered by a non-owner thread, and requires inter-thread communication through OS signals or safepoints. This is the main drawback of biased locking.

Subsequent works on biased locking [84, 85, 86, 87, 88, 89] improve on the original work by making ownership revocation more efficient, enabling ownership transfer, or determining when it is best to bias an object. It is possible to apply such ownership transfer techniques to BRC. In future work, we plan to implement similar techniques to better support various program behaviors.

Many prior works on RC [62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75] focus on reducing the number of RC operations. They are are briefly summarized in Section 4.2.2, and compared to BRC in Section 4.7.1. Another category of works attempt to efficiently detect and remove cyclic references [90, 91, 92, 93, 94]. Swift solves this problem through weak references, an approach popular in previous literature [77, 78]. We believe that BRC can also be integrated into RC implementations with cyclic reference detection and removal algorithms.

Joao et al. [95] propose hardware support for RC. They augment the cache hierarchy to gradually merge RC operations. Due to the delay of merging in hardware, their technique

does not support immediate reclamation. In contrast, BRC supports immediate reclamation in most cases.

Recently, Ungar et al. [96] propose a compiler-assisted dynamic optimization technique for RC in Swift. It is similar to BRC in that it dynamically replaces atomic RC operations with non-atomic ones. It adds checks before stores to conservatively capture escaping objects, and uses atomic RC operations for escaped objects only. Compared to BRC, their technique maintains the immediate reclamation property of non-deferred RC, while BRC relaxes this for queued objects. However, their technique uses more atomic operations than BRC due to its conservative escape detection, and its lack of the notion of biased threads. In addition, it increases the overhead of the store barrier to detect and recursively mark escaping objects. Finally, it does not fully support all of Swift's function argument passing semantics.

4.9 CONCLUSION

This paper proposed Biased Reference Counting (BRC), a novel approach to speed-up non-deferred RC for garbage collection. BRC is based on the observations that most objects are mostly accessed by a single thread, and that atomic operations have significant overheads. BRC biases each object toward a specific thread. Further, BRC adds a second counter to the object header, enabling the owner thread to have its own counter. These changes allow the owner thread of each object to perform RC operations without atomic operations, while the other threads atomically update the other counter. BRC correctly manages the merging of these two counters, handling all corner cases.

We implemented BRC in the Swift programming language runtime and evaluated it with various client and server programs. We found that BRC accelerated non-deferred RC. Specifically, it reduced the average execution time of client programs by 22.5%, and improved the average throughput of server programs by 7.3%.

CHAPTER 5: CONCLUSION

Modern programming languages provide many convenient features, such as dynamic type system and automatic memory management, which increase the productivity of programmers. Such benefits, however, come with performance overheads.

This thesis focused on the overheads of various convenient features provided by modern programming languages. It started by evaluating the overheads of dynamic type system. It analyzed the state-of-the-art implementation of inline caching which is one of the fundamental optimization techniques for dynamically-typed languages. First, the thesis proposed an enhanced IC design called *Reusable Inline Caching (RIC)* which enabled the reuse of IC information across executions to avoid deterministic IC misses during JavaScript initialization. Second, it proposed *ShortCut*, an architectural support to accelerate IC hit operations to improve the overall performance by extending the branch prediction mechanism. Lastly, it evaluated the overheads of reference counting (RC) for automatic memory management and proposed a novel algorithm for reference counting, *Biased Reference Counting (BRC)*, which accelerated RC operations by biasing each object toward a specific thread.

REFERENCES

[1] "The Chromium Projects," https://www.chromium.org/.

[2] "Tizen," https://www.tizen.org/.

[3] "Node.js," https://nodejs.org/.

[4] "Ruby on Rails," http://rubyonrails.org/.

[5] "Perfect: Server-side Swift," http://perfect.org/.

[6] "Kitura: A Swift Web framework and HTTP Server," http://www.kitura.io/.

[7] "V8 JavaScript Engine," https://developers.google.com/v8/.

[8] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The hiphop virtual machine," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 777–790.

[9] J. Oh and S.-M. Moon, "Snapshot-based loading-time acceleration for web applications," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 179–189.

[10] Zona Research, "The economic impacts of unacceptable web-site download speeds," 1999.

[11] Jupiter Research, "Retail web site performance: Consumer reaction to a poor online shopping experience," 2006.

[12] C. Rheem, "Consumer response to travel site performance," 2010.

[13] Akamai Technologies, "Performance matters: 9 key consumer insights," 2014.

[14] B. McQuade, D. Phan, and M. Vajihollahi, "Instant mobile websites: Techniques and best practices," 2013.

[15] T. Verwaest and C. Bruni, "Real-world JavaScript Performance," https://docs.google.com/presentation/d/14WZkWbkvtmZDEIBYP5H1GrbC9H-W3nJSg3nvpHwfG5U/edit?usp=sharing.

[16] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications," in *Proceedings of the 2010 USENIX Conference on Web Application Development*, ser. WebApps'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3.

[17] "New JavaScript techniques for rapid page loads," https://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html.

[18] "Improved code caching," https://v8.dev/blog/improved-code-caching.

[19] C. Chambers, D. Ungar, and E. Lee, "An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '89. New York, NY, USA: ACM, 1989, pp. 49–70.

[20] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '84. New York, NY, USA: ACM, 1984, pp. 297–302.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[22] AngularJS, https://angularjs.org/.

[23] React, https://facebook.github.io/react/.

[24] CamanJS, http://camanjs.com/.

[25] Handlebars, http://handlebarsjs.com/.

[26] Underscore, http://underscorejs.org/.

[27] jQuery, http://jquery.com.

[28] JSFeat, http://inspirit.github.io/jsfeat/.

[29] "Octane Benchmark," https://developers.google.com/octane/.

[30] W. Ahn, J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas, "Improving javascript performance by deconstructing the type system," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 496–507.

[31] "Custom startup snapshots," https://v8.dev/blog/custom-startup-snapshots.

[32] Google, https://blog.mozilla.org/luke/2014/01/14/asm-js-aot-compilation-and-startup-performance/.

[33] M. N. Kedlaya, B. Robatmili, and B. Hardekopf, "Server-side type profiling for optimizing client-side javascript engines," in *Proceedings of the 11th Symposium on Dynamic Languages*, ser. DLS 2015. New York, NY, USA: ACM, 2015, pp. 140–153.

[34] "ECMAScript," http://www.ecmascript.org/.

[35] "Python Programming Language," https://www.python.org/.

[36] "Ruby Programming Language," https://www.ruby-lang.org/.

[37] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *Proceedings of the European Conference on Object-Oriented Programming*, ser. ECOOP '91. London, UK, UK: Springer-Verlag, 1991, pp. 21–38.

[38] "JavaScriptCore," https://trac.webkit.org/wiki/JavaScriptCore.

[39] K. Nygaard and O.-J. Dahl, "The development of the simula languages," *SIGPLAN Not.*, vol. 13, no. 8, pp. 245–272, Aug. 1978.

[40] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[41] C. L. Farooq, Muhammad Umar and L. K. John, "Value based btb indexing for indirect jump prediction," in *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture*, ser. HPCA '10, Jan 2010, pp. 1–11.

[42] C. Kim, S. Kim, H. G. Cho, D. Kim, J. Kim, Y. H. Oh, H. Jang, and J. W. Lee, "Short-circuit dispatch: Accelerating virtual machine interpreters on embedded processors," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, ser. ISCA '16, June 2016, pp. 291–303.

[43] "SunSpider Benchmark," https://webkit.org/perf/sunspider/sunspider.html.

[44] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[45] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, no. 1, pp. 6–22, Jan. 1984.

[46] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 80–90.

[47] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "Vpc prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 424–435.

[48] V. Agrawal, A. Dabral, T. Palit, Y. Shen, and M. Ferdman, "Architectural support for dynamic linking," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 691–702.

[49] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers, "Checked load: Architectural support for javascript type-checking on mobile processors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11, Feb 2011, pp. 419–430.

[50] G. Chadha, S. Mahlke, and S. Narayanasamy, "Accelerating asynchronous programs through event sneak peek," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 642–654.

[51] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side web applications," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 762–774.

[52] M. Mehrara and S. Mahlke, "Dynamically accelerating client-side web applications through decoupled execution," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 74–84.

[53] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 87–98.

[54] G. Dot, A. Martínez, and A. González, "Erico: Effective removal of inline caching overhead in dynamic typed languages," in *Proceedings of the 2016 IEEE 23rd International Conference on High Performance Computing*, ser. HiPC '16, Dec 2016, pp. 372–381.

[55] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf, "Improved type specialization for dynamic scripting languages," in *Proceedings of the 9th Symposium on Dynamic Languages*, ser. DLS '13. New York, NY, USA: ACM, 2013, pp. 37–48.

[56] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf, "Deoptimization for dynamic language jits on typed, stack-based virtual machines," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: ACM, 2014, pp. 103–114.

[57] Benedikt Meurer, "V8: Behind the Scenes (February Edition feat. A tale of TurboFan)," http://benediktmeurer.de/2017/03/01/v8-behind-the-scenes-february-edition/, March 2017.

[58] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[59] G. E. Collins, "A method for overlapping and erasure of lists," *Commun. ACM*, vol. 3, no. 12, pp. 655–657, Dec. 1960.

[60] "Swift Programming Language," https://swift.org/.

[61] K. Kawachiya, A. Koseki, and T. Onodera, "Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations," in *Proc. of the 17th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02, 2002.

[62] L. P. Deutsch and D. G. Bobrow, "An efficient, incremental, automatic garbage collector," *Commun. ACM*, vol. 19, no. 9, pp. 522–526, 1976.

[63] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ser. SDE 1, 1984, pp. 157–167.

[64] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith, "Java without the coffee breaks: A nonintrusive multiprocessor garbage collector," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01, 2001, pp. 92–103.

[65] S. M. Blackburn and K. S. McKinley, "Ulterior reference counting: Fast garbage collection without a long wait," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03, 2003, pp. 344–358.

[66] R. Shahriyar, S. M. Blackburn, and D. Frampton, "Down for the count? getting reference counting back in the ring," in *Proceedings of the 2012 International Symposium on Memory Management*, ser. ISMM '12, 2012, pp. 73–84.

[67] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley, "Taking off the gloves with reference counting immix," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-oriented Programming Systems Languages, and Applications*, ser. OOPSLA '13, 2013, pp. 93–110.

[68] R. Shahriyar, S. M. Blackburn, and K. S. McKinley, "Fast conservative garbage collection," in *Proceedings of the 2014 ACM International Conference on Object-oriented Programming Systems Languages, and Applications*, ser. OOPSLA '14, 2014, pp. 121–139.

[69] Y. Levanoni and E. Petrank, "An on-the-fly reference counting garbage collector for java," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '01, 2001, pp. 367–380.

[70] H. Paz and E. Petrank, "Using prefetching to improve reference-counting garbage collectors," in *Proceedings of the 16th International Conference on Compiler Construction*, ser. CC'07, 2007, pp. 48–63.

[71] P. G. Joisha, "Compiler optimizations for nondeferred reference-counting garbage collection," in *Proceedings of the 5th International Symposium on Memory Management*, ser. ISMM '06, 2006.

[72] P. G. Joisha, "Overlooking roots: A framework for making nondeferred reference-counting garbage collection fast," in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM '07, 2007, pp. 141–158.

[73] P. G. Joisha, "A principled approach to nondeferred reference-counting garbage collection," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08, 2008, pp. 131–140.

[74] Y. G. Park and B. Goldberg, "Reference escape analysis: Optimizing reference counting based on the lifetime of references," in *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '91, 1991, pp. 178–189.

[75] J. M. Barth, "Shifting garbage collection overhead to compile time," *Communications of the ACM*, vol. 20, no. 7, pp. 513–518, 1977.

[76] "Swift has reached 1.0," https://developer.apple.com/swift/blog/?id=14.

[77] D. R. Brownbridge, "Cyclic reference counting for combinator machines," in *Conference on Functional Programming and Computer Architecture*, 1985, pp. 273–288.

[78] T. H. Axford, "Reference counting of cyclic graphs for functional programs," *The Computer Journal*, vol. 33, no. 5, pp. 466–472, 1990.

[79] "Swift Benchmark Suite," https://github.com/apple/swift/tree/master/benchmark.

[80] "CryptoSwift," https://github.com/krzyzanowskim/CryptoSwift.

[81] "SwiftyJSON," https://github.com/SwiftyJSON/SwiftyJSON.

[82] "Swift Version of Ray Tracing," https://github.com/rnapier/raytrace.

[83] "Server-side swift benchmarks," https://github.com/rymcol/Server-Side-Swift-Benchmarks-Summer-2017.

[84] D. Dice, M. Moir, and W. S. III, "Quickly Reacquirable Locks," Sun Microsystem Laboratories, Tech. Rep., 2003.

[85] T. Onodera, K. Kawachiya, and A. Koseki, "Lock Reservation for Java Reconsidered," in *Proceedings of the 18th European Conference on Object-Oriented Programming*, ser. ECOOP '04, 2004.

[86] K. Russell and D. Detlefs, "Eliminating Synchronization-related Atomic Operations with Biased Locking and Bulk Rebiasing," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06, 2006, pp. 263–272.

[87] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards, "Simple and Fast Biased Locks," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 65–74.

[88] F. Pizlo, D. Frampton, and A. L. Hosking, "Fine-grained Adaptive Biased Locking," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '11, 2011, pp. 171–181.

[89] I. Rogers and B. Iyengar, "Reducing Biased Lock Revocation by Learning," in *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2011, pp. 65–73.

[90] R. E. Jones and R. D. Lins, "Cyclic weighted reference counting without delay," in *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, ser. PARLE '93, 1993, pp. 712–715.

[91] A. D. Martínez, R. Wachenchauzer, and R. D. Lins, "Cyclic reference counting with local mark-scan," *Inf. Process. Lett.*, vol. 34, no. 1, pp. 31–35, 1990.

[92] D. F. Bacon and V. T. Rajan, "Concurrent cycle collection in reference counted systems," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01, 2001, pp. 207–235.

[93] H. Paz, E. Petrank, D. F. Bacon, E. K. Kolodner, and V. T. Rajan, "An efficient on-the-fly cycle collection," in *Proceedings of the 14th International Conference on Compiler Construction*, ser. CC'05, 2005, pp. 156–171.

[94] H. Paz, D. F. Bacon, E. K. Kolodner, E. Petrank, and V. T. Rajan, "An efficient on-the-fly cycle collection," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 4, 2007.

[95] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, 2009, pp. 418–428.

[96] D. Ungar, D. Grove, and H. Franke, "Dynamic atomicity: Optimizing swift memory management," in *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS '17, 2017, pp. 15–26.