ShortCut: Architectural Support for Fast Object Access in Scripting Languages

Jiho Choi Thomas Shull Maria J. Garzaran Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ABSTRACT

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 paper 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.

CCS CONCEPTS

• Computer systems organization → High-level language architectures; **• Software and its engineering → Just-in-time compilers**; **Scripting languages**;

KEYWORDS

Microarchitecture, Inline Caching, Scripting Language, JavaScript

ACM Reference format:
Jiho Choi Thomas Shull Maria J. Garzaran Josep Torrellas University of Illinois at Urbana-Champaign. 2017. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080237

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06. . . $15.00
https://doi.org/10.1145/3079856.3080237

1 INTRODUCTION

Dynamic scripting languages such as JavaScript [1], Python [5], and Ruby [7] are widely used in many application domains, both for clients [9, 10] and servers [3, 6]. 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 [11] 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.

```
function get_x(obj) {
 return obj.x;
}

a = {x:0};
b = {y:1, x:2};

get_x(a); // IC miss
get_x(a); // IC hit

get_x(b); // IC miss
get_x(b); // IC hit
```

(a) JavaScript code example with a property load

```
// get_x
...
if obj_type == Type_a
  call Handler_a
elif obj_type == Type_b
  call Handler_b
else
  call IC_Miss_Handler
...
```

(b) Inlined dispatcher

```
// get_x
...
call Custom_Dispatcher
...
```

```
// Custom_Dispatcher
if obj_type == Type_a
  jmp Handler_a
elif obj_type == Type_b
  jmp Handler_b
else
  jmp IC_Miss_Handler
```

(c) Custom dispatcher

```
// get_x
...
call Shared_Read_Dispatcher
...
```

```
// Shared_Read_Dispatcher
foreach Entry in IC_Vector (of site S)
  if (obj_type == Entry.Obj_type)
    jmp Entry.Handler
jmp IC_Miss_Handler
```

(d) Shared read dispatcher

Obj_type	Handler
$\texttt{Type}_a$	$\texttt{Handler}_a$
$\texttt{Type}_b$	$\texttt{Handler}_b$
...	...

(e) IC Vector of site *S*

```
// Handler_a
ld R_dst,[R_obj+offset_a]
ret
```

```
// Handler_b
ld R_dst,[R_obj+offset_b]
ret
```

(f) Handlers

Figure 1: Code generation example with different inline cache implementations.

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.

2 BACKGROUND AND MOTIVATION

2.1 Inline Caching in Scripting Languages

Many scripting languages, including JavaScript [1], Python [5], and Ruby [7], 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.

As an example, consider Figure 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.

2.2 Approaches to Inline Caching

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

2.2.1 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 1(b) shows an example. This code performs the actual reading of `obj.x` in function `get_x`. In Figure 1(b), the code checks for $\texttt{Type}_a$ and $\texttt{Type}_b$ in sequence. The handlers access the property and then return, as shown in Figure 1(f). If no match occurs, the code in Figure 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 [11] uses this design, and even inlines the handlers at the access site.

2.2.2 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 [23]. In the dispatcher, the code has the usual set of *if* statements with comparisons and jumps to handlers.

Figure 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).

2.2.3 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 1(a) — i.e., `obj.x`. Figure 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 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.

2.2.4 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.

3 SHORTCUT ARCHITECTURE

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 2(a). The access site (e.g., `obj.x` in Figure 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 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 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 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 1(f). Calling the handler and returning has substantial overhead. Hence, Aggressive ShortCut takes a more aggressive approach, as shown in Figure 2(c). The idea is to perform the load or the store as part of the original instruction that used to call the dispatcher.

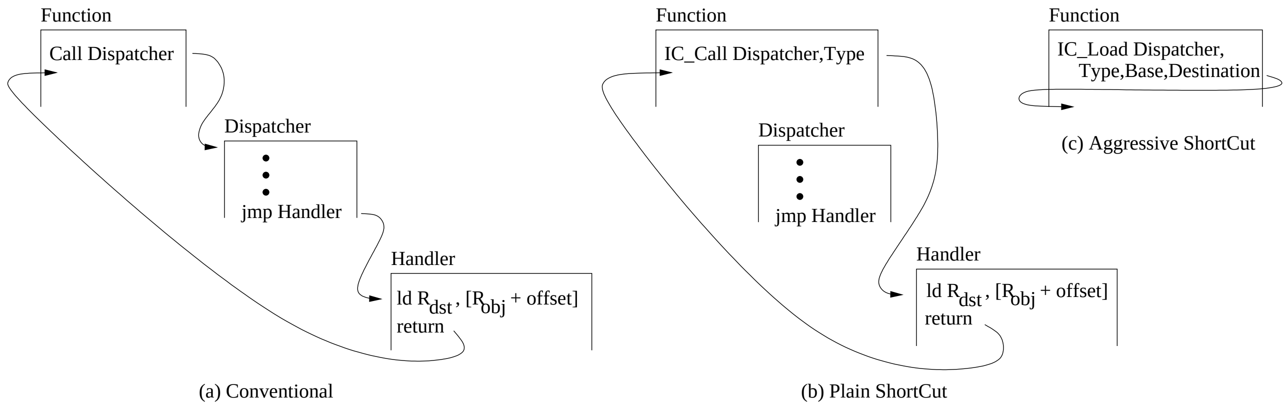

Figure 2: Operations of a conventional IC (a), Plain ShortCut (b), and Aggressive ShortCut (c).

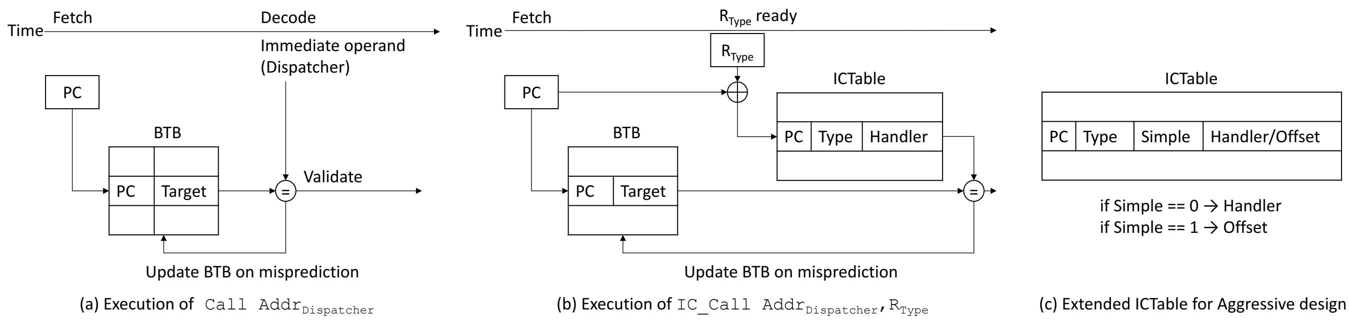

Figure 3: Structures used in a conventional IC (a), Plain ShortCut (b), and Aggressive ShortCut (c).

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 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 2(a).

To see how we support these ideas, consider first how a conventional `Call Dispatcher` instruction works in Figure 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(b). The `Call Dispatcher` instruction is now replaced with our *IC_Call* instruction. The *IC_Call* instruction takes an additional register (R_{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 R_{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(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.

3.2 Detailed Design of Plain ShortCut

The Plain ShortCut architecture is shown in Figure 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 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(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.

3.2.1 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 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 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 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 1(f), these simple handlers execute an instruction of the form `ld` R_{dst}`,[`R_{base}`+offset]` (or `st` R_{src}`,[`R_{base}`+offset]`

#	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 1: Interaction between the BTB and the ICTable.

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 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(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 (Section 2.2.3), 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 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*.

4 ADDITIONAL DESIGN ASPECTS

4.1 ISA Extensions

ShortCut extends the ISA to expose the ICTable to the software. It adds the five instructions shown in Table 2. The first three instructions can replace the call to the dispatcher at IC access sites (`call` $Addr_D$).

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 2: Instructions added by ShortCut.

IC_Call $Addr_D$**,** R_{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 R_{Type}. If it hits, control is transferred to the handler provided by the ICTable; otherwise, control is transferred to the dispatcher.

IC_Load $Addr_D$**,** R_{Type}**.** It takes as operands the relative address of the dispatcher, one explicit register with the object type, and two implicit registers. One of the implicit registers contains the base of the object (R_{base}) and the other will receive the value of the object property (R_{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 R_{Type}. If it hits and the ICTable provides an offset, the instruction performs `ld` R_{dst}, [R_{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.

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 (Section 2.2.3), 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.

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 R_{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.

5 DISCUSSION

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 [2]. 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 [31], 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.

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 [16] 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.

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 [2] uses bytecode data, and the upcoming V8 Ignition [11] 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 [22, 25].

6 EXPERIMENTAL SETUP

To support ShortCut, we implement the compiler changes discussed in Section 4.2 to the state-of-the-art Google V8 JavaScript JIT compiler [11]. 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 [4] and SunSpider 1.0.2 [8] application suites.

To model and evaluate the ShortCut hardware, we extend the Sniper simulator [17], which is a widely-used Pin-based [28] architecture simulator. The parameters of the processor architecture are shown in Table 3. The baseline processor uses a 4K-entry BTB to predict indirect branches [27]. 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.

We evaluate the four pairs of configurations shown in Table 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 [22, 24, 26]. Finally, we model Plain ShortCut with its modified V8 compiler (*PSO, PS*), and Aggressive ShortCut with its modified V8 compiler (*ASO, AS*).

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: 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 4: Architecture and compiler configurations evaluated.

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 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*.

7 EVALUATION

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 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.

Figures 4 and 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.

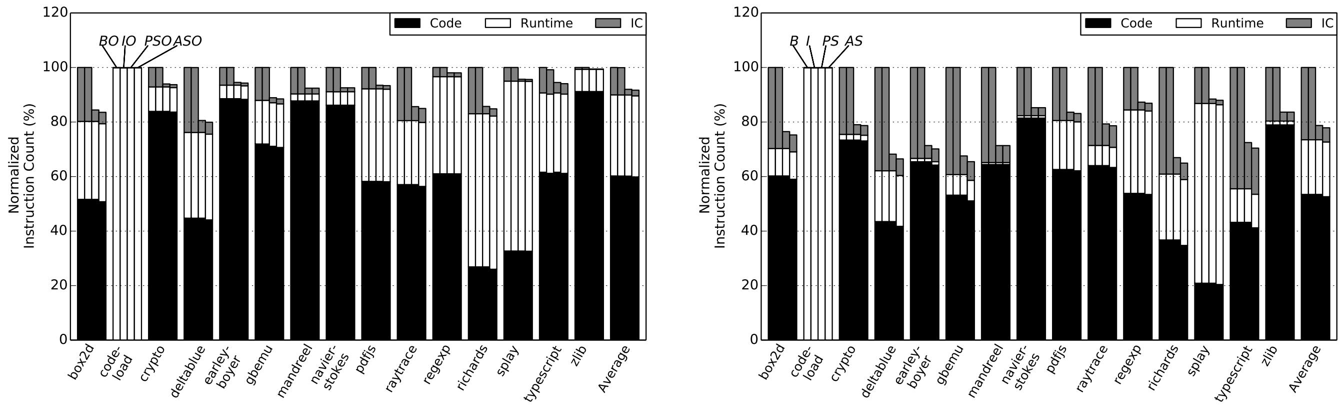

Figure 4: Breakdown of dynamic instruction count in Octane with (left) and without (right) the optimizing tier.

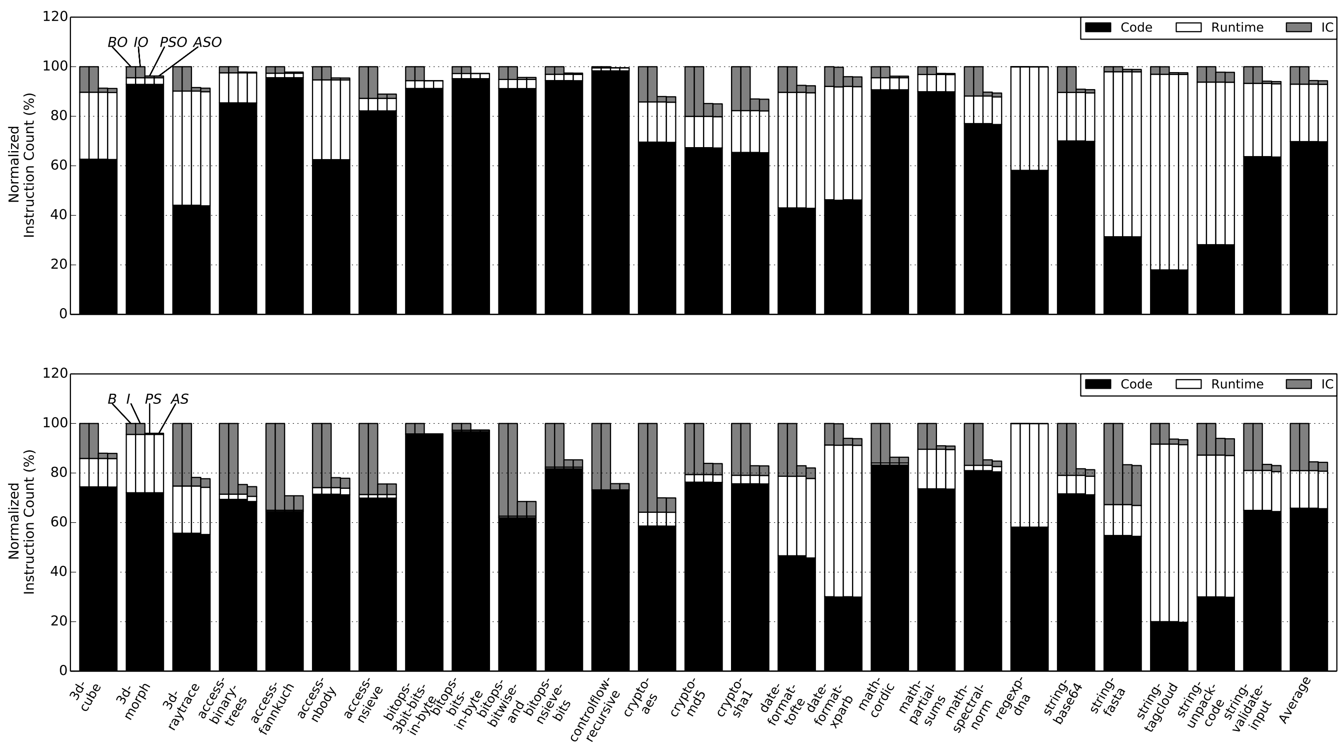

Figure 5: Breakdown of dynamic instruction count in SunSpider with (top) and without (bottom) the optimizing tier.

If we have the optimizing tier enabled (Figures 4-left and 5-top), the unmodified V8 system (*BO*) executes, on average, 10% and 7% of the instructions in the IC dispatcher for Octane and SunSpider, respectively. Moreover, if we do not have the optimizing tier enabled, (Figures 4-right and 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 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.

7.2 Impact of ShortCut

7.2.1 Dynamic Instruction Count. We now compare the different bars in Figures 4 and 5. First, the *IO* and *I* configurations have the same instruction count as *BO* and *B*, respectively.

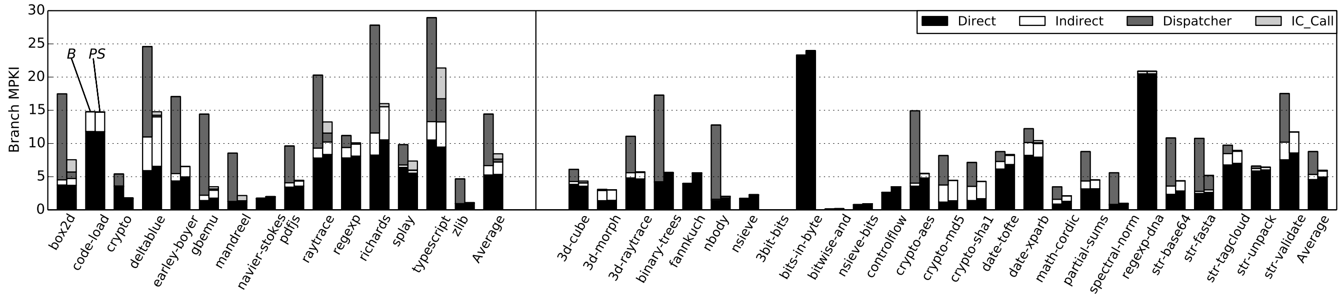

Figure 6: Branch MPKI in Octane (left) and SunSpider (right) for *B* and *PS*.

We now consider Plain ShortCut. Without the optimizing tier, (Figures 4-right and 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 4-left and 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 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 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*).

7.2.2 Branch Prediction. Figure 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 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.

7.2.3 Execution Time. Figures 7 and 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 4 and 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 7-right and 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 Sections 7.2.1 and 7.2.2. *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 7-left and 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.

7.3 Sensitivity Study

We perform a sensitivity study by varying the ICTable size at a fixed associativity of 4, for Plain ShortCut. Figure 9 shows the average execution time of Octane and SunSpider, respectively, for *PS* with different ICTable sizes relative to *B*.

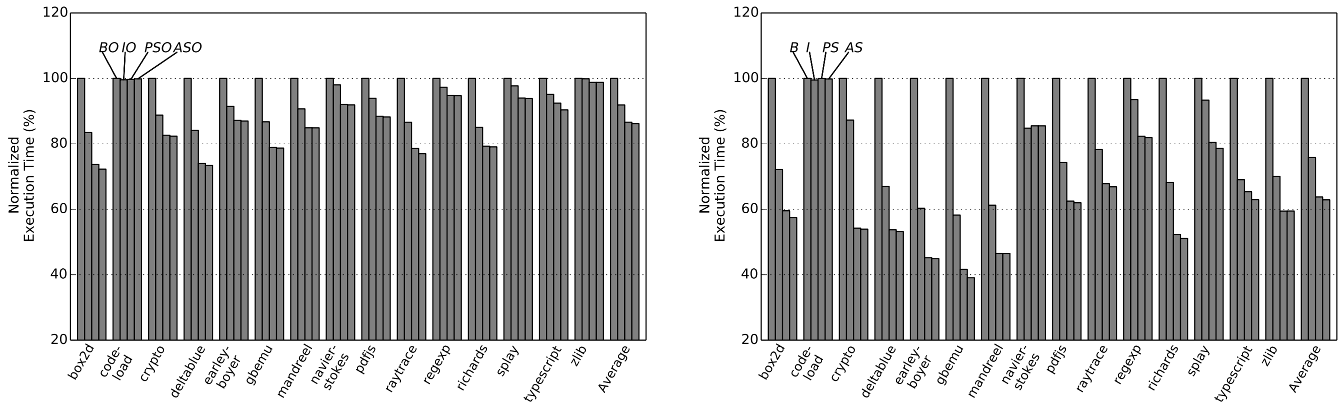

Figure 7: Normalized execution time of Octane with (left) and without (right) the optimizing tier.

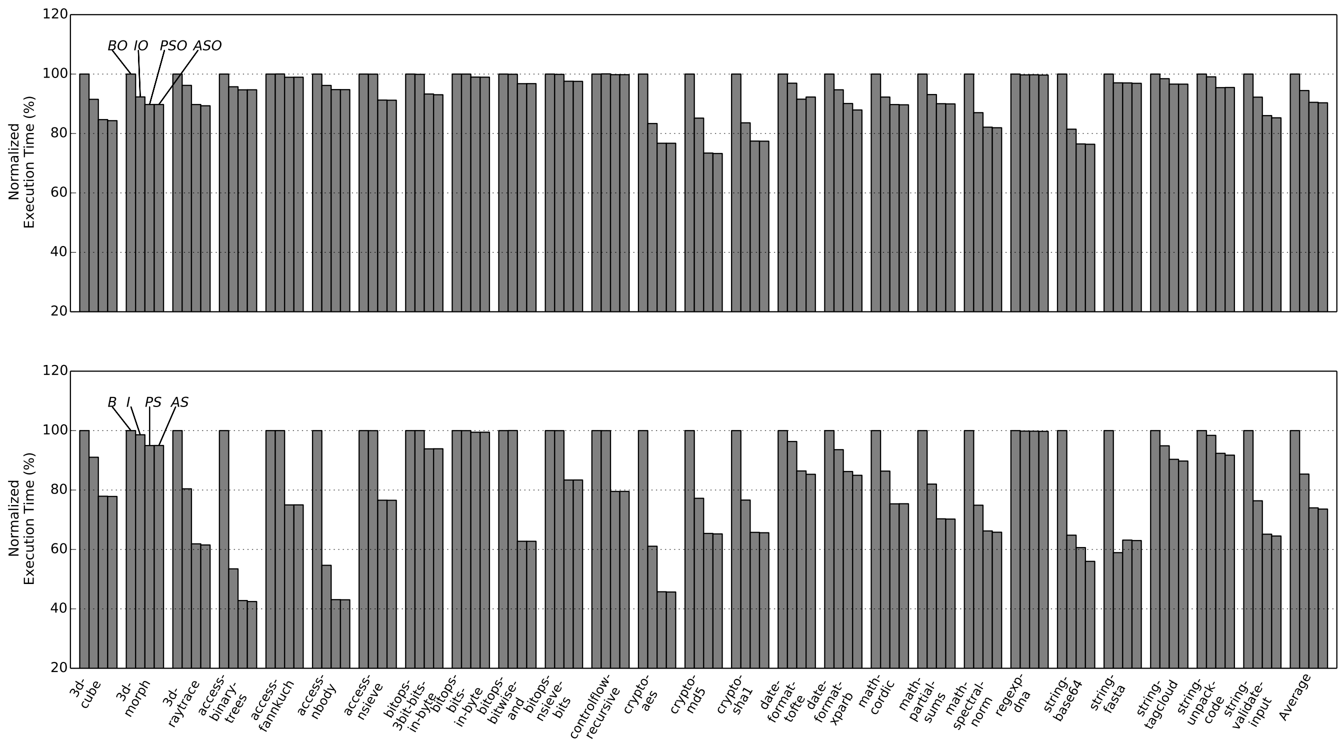

Figure 8: Normalized execution time of SunSpider with (top) and without (bottom) the optimizing tier.

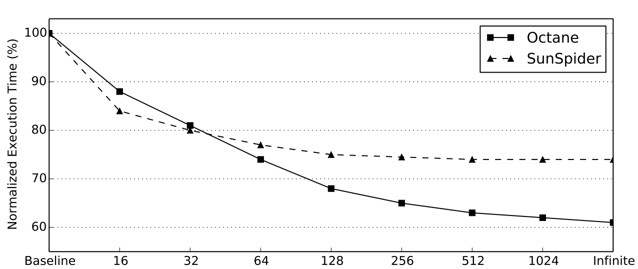

Figure 9: Sensitivity of the execution time of *PS* to varying ICTable size. The execution time is normalized to *B*.

Figure 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 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.

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, 23].

Ahn et al. [13] 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 [22, 24, 26]. In particular, our approach of using both the access site address and the object type to index the ICTable resembles the VBBI BTB design [22]. 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. [12] 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. [25] 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 [14] extends the ISA to reduce the overhead of checking primitive types. Some proposals improve instruction cache performance on client-side [18] and server-side [32] JavaScript programs. ParaGuard [30] and ParaScript [29] enable parallel execution of JavaScript programs. These proposals are orthogonal to ShortCut, and they could be used together with it.

Dot et al. [21] 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.

Since the submission of this paper, the V8 team has announced their intention to change the compilation tiers within V8 [15]. 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.

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.

ACKNOWLEDGMENT

This work was supported in part by NSF under grants CCF 1527223 and CNS 1319657.

REFERENCES

[1] ECMAScript. http://www.ecmascript.org/.
[2] JavaScriptCore. https://trac.webkit.org/wiki/JavaScriptCore.
[3] Node.js. https://nodejs.org/.
[4] Octane Benchmark. https://developers.google.com/octane/.
[5] Python Programming Language. https://www.python.org/.
[6] Ruby on Rails. http://rubyonrails.org/.
[7] Ruby Programming Language. https://www.ruby-lang.org/.
[8] SunSpider Benchmark. https://webkit.org/perf/sunspider/sunspider.html.
[9] The Chromium Projects. https://www.chromium.org/.
[10] Tizen. https://www.tizen.org/.
[11] V8 JavaScript Engine. https://developers.google.com/v8/.
[12] Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, and Michael Ferdman. 2015. Architectural Support for Dynamic Linking. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 691–702. https://doi.org/10.1145/2694344.2694392
[13] Wonsun Ahn, Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 496–507. https://doi.org/10.1145/2594291.2594332
[14] Owen Anderson, Emily Fortuna, Luis Ceze, and Susan Eggers. 2011. 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 (HPCA '11)*. 419–430. https://doi.org/10.1109/HPCA.2011.5749748
[15] Benedikt Meurer. 2017. 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).
[16] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692
[17] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models.

ACM Trans. Archit. Code Optim. 11, 3, Article 28 (Aug. 2014), 25 pages. https://doi.org/10.1145/2629677

[18] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2015. Accelerating Asynchronous Programs Through Event Sneak Peek. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 642–654. https://doi.org/10.1145/2749469.2750373

[19] C. Chambers, D. Ungar, and E. Lee. 1989. 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 (OOPSLA '89)*. ACM, New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

[20] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[21] Gem Dot, Alejandro Martínez, and Antonio González. 2016. ERICO: Effective Removal of Inline Caching Overhead in Dynamic Typed Languages. In *Proceedings of the 2016 IEEE 23rd International Conference on High Performance Computing (HiPC '16)*. 372–381. https://doi.org/10.1109/HiPC.2016.050

[22] Chen Lei Farooq, Muhammad Umar and Lizy K. John. 2010. Value Based BTB Indexing for Indirect Jump Prediction. In *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA '10)*. 1–11. https://doi.org/10.1109/HPCA.2010.5416659

[23] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK, 21–38. http://dl.acm.org/citation.cfm?id=646149.679193

[24] Jose A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N. Patt. 2008. 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 (ASPLOS XIII)*. ACM, New York, NY, USA, 80–90. https://doi.org/10.1145/1346281.1346293

[25] Channoh Kim, Sungmin Kim, Hyeon Gyu Cho, Dooyoung Kim, Jaehyeok Kim, Young H. Oh, Hakbeom Jang, and Jae W. Lee. 2016. Short-Circuit Dispatch: Accelerating Virtual Machine Interpreters on Embedded Processors. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA '16)*. 291–303. https://doi.org/10.1109/ISCA.2016.34

[26] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. 2007. VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-based Dynamic Devirtualization. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 424–435. https://doi.org/10.1145/1250662.1250715

[27] J. K. F. Lee and A. J. Smith. 1984. Branch Prediction Strategies and Branch Target Buffer Design. *Computer* 17, 1 (Jan. 1984), 6–22. https://doi.org/10.1109/MC.1984.1658927

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[29] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. 2011. 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 (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 87–98. http://dl.acm.org/citation.cfm?id=2014698.2014898

[30] Mojtaba Mehrara and Scott Mahlke. 2011. Dynamically Accelerating Client-side Web Applications Through Decoupled Execution. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 74–84. http://dl.acm.org/citation.cfm?id=2190025.2190055

[31] Kristen Nygaard and Ole-Johan Dahl. 1978. The Development of the SIMULA Languages. *SIGPLAN Not.* 13, 8 (Aug. 1978), 245–272. https://doi.org/10.1145/960118.808391

[32] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. 2015. Microarchitectural Implications of Event-driven Server-side Web Applications. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/2830772.2830792