CacheIR: The Benefits of a Structured Representation for Inline Caches

Jan de Mooij
jdemooij@mozilla.com
Mozilla
Utrecht, the Netherlands

Matthew Gaudet
mgaudet@mozilla.com
Mozilla
Toronto, Canada

Iain Ireland
iireland@mozilla.com
Mozilla
Toronto, Canada

Nathan Henderson
nthender@ualberta.ca
University of Alberta
Edmonton, Canada

J. Nelson Amaral
jamaral@ualberta.ca
University of Alberta
Edmonton, Canada

Abstract

Inline Caching is an important technique used to accelerate operations in dynamically typed language implementations by creating fast paths based on observed program behaviour. Most software stacks that support inline caching use low-level, often *ad-hoc*, Inline-Cache (ICs) data structures for code generation. This work presents *CacheIR*, a design for inline caching built entirely around an intermediate representation (IR) which: (i) simplifies the development of ICs by raising the abstraction level; and (ii) enables reusing compiled native code through IR matching techniques. Moreover, this work describes *WarpBuilder*, a novel design for a Just-In-Time (JIT) compiler front-end that directly generates type-specialized code by lowering the CacheIR contained in ICs; and *Trial Inlining*, an extension to the inline-caching system that allows for context-sensitive inlining of context-sensitive ICs. The combination of CacheIR and WarpBuilder have been powerful performance tools for the SpiderMonkey team, and have been key in providing improved performance with less security risk.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Interpreters**; **Just-in-time compilers**; **Dynamic compilers**.

Keywords: inline caching, dynamically-typed programming languages, just-in-time compilation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MPLR '23, October 22, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0380-5/23/10…$15.00
https://doi.org/10.1145/3617651.3622979

ACM Reference Format:
Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J. Nelson Amaral. 2023. CacheIR: The Benefits of a Structured Representation for Inline Caches. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3617651.3622979

1 Introduction

Throughout the extensive history of dynamically typed languages (DTLs) and their pursuit of efficient software execution, two fundamental techniques have proven their resilience: JIT compilation and inline caching.

JIT compilation can make computation more efficient by leveraging dynamic information to compile language methods, or traces, into native code at runtime. Inline caching can reduce the cost of polymorphic operations (e.g. method dispatch and operators) by creating a cache *directly associated with a particular call-site or operator instance*. The original design of inline caching relied on the observation that operation sites may be polymorphic in principle, but were typically monomorphic in practice [12]. Later, inline caches were extended to sites that are polymorphic but have a limited number of frequent targets [16]. In most software stacks for dynamically typed languages, inline caching is an integral part of the JIT code generation process.

To illustrate the inline-cache mechanism, consider a binary + operator. A particular `x+y` expression can represent many operations such as string concatenation, list appending, floating-point addition, or other operations, depending on the language semantics. The computation that must be performed when this expression is evaluated depends on the type of the operands. A naive system would invoke a virtual-machine routine each time that the expression is evaluated to ensure that the correct operation is performed, but this could lead to poor performance if that particular expression always resolves to the same computation. Alternatively, a system that supports inline caching for + resolves a `x+y` expression by (i) identifying the types of `x` and `y`, (ii) constructing a fast

path conditioned on those types, and (iii) saving the conditioned fast path in an inline cache. On subsequent visits to the same x+y expression, the system consults the cache and takes the fast path if the incoming operand types agree with the cached operand types. This system avoids the slow path subsequently so long as the types are the same.

There are various possible representations for inline caches. Early representations tended to include branches in the emitted JIT compiled code that could be patched by the runtime system [12, 16, 17]. In V8, polymorphic inline caches are represented as a table of tuples containing a mapping from runtime type to native handler code [2, 11]. In SpiderMonkey, and other projects such as GraalVM [20], ICs are represented as chains of type-specialized fast paths [1]. Each fast path in SpiderMonkey is a *stub*, which is a data structure associated with a small sequence of native code referred to as the *stub code*. Each stub code sequence contains a fast implementation that is specialized to an observed operand type for the associated operation.

SpiderMonkey is unique in its adoption of CacheIR, a bytecode intermediate representation (IR) for inline cache stubs. In SpiderMonkey, using CacheIR makes creating IC stubs akin to compiling small methods in a JIT. When attaching a stub, SpiderMonkey creates a representation in this specialized IR and compiles the IR representation to match the appropriate calling convention to deploy the IC.

This paper reports on the advantages created through the adoption of an intermediate representation for inline caching, including: (i) the simplification of adding new ICs, which increases the productivity of compiler developers, and (ii) the creation of a novel JIT compiler design that can consume the inline cache IR directly as part of the JIT compiler IR for a method that naturally creates opportunities for type specialization in the JIT compiler. This strategy is adopted in the front-end of SpiderMonkey's top-tier JIT compiler where bytecodes that have a single IC stub have their CacheIR directly lowered into a JIT compiler IR sub-graph. This architecture has been very successful at Mozilla and, with an extension called *Trial Inlining*, has outperformed SpiderMonkey's previous JIT compiler system when evaluated on workloads representative of deployed Web applications.

2 Making JavaScript Fast

Implementing JavaScript in a performant manner is a challenge because it is a dynamic language. To address this challenge, numerous techniques have been adapted and innovated to create fast and efficient JavaScript implementations.

2.1 Inline Caching: Dynamically Creating Fast Paths

Inline caching was initially proposed by L. Peter Deustch and Allan M. Schiffman to accelerate method invocation on dynamically typed objects in the Smalltalk-80 system [12]. For instance, the code o.x sends a message x to a receiver object o. Many different methods named x could be invoked depending on the runtime types of the object o. To avoid executing an expensive lookup routine each time the program evaluates o.x, the address for the previously resolved method is cached *inline* to accelerate subsequent invocations of x when o is of the same type. Initially, the method call is bound to the address of the default method-lookup routine; the *fallback* case. When the lookup routine resolves an address, it overwrites the fallback address to provide a fast path for o's type. On later evaluations of the same o.x expression, o's type is checked against the type corresponding to the cache. If the type is the same, the fast path is taken, otherwise, the fallback path is taken and the call site is bound to the newly resolved method address. This IC scheme improved the Smalltalk-80 system greatly but was limited because it provided a fast path for only a single type for each static expression o.x.

Recognizing that languages such as SELF [10] often have more frequent polymorphic method invocations, Urz Hölze, Craig Chambers, and David Ungar introduced Polymorphic Inline Caches (PICs) to accelerate the operation sites that observe up to ten types [16]. For polymorphic sites, instead of embedding the resolved method address directly into the native code, the system embeds the address of a created stub that guards on previously observed types in a switch-statement-like manner and executes the associated method. If no guards pass, a PIC invokes the fallback method and appends a new guarded fast path within the stub.

These seminal approaches established the concepts of guards and fallback paths for ICs. Guards are conditional branches that ensure the correctness of fast-path execution based on observed types. A fallback path is a routine that is executed to ensure that types previously unseen are handled appropriately. Modern dynamically-typed language implementations, such as SpiderMonkey, V8, and JavaScriptCore (JSC) JS engines, use a variation of this PIC technique to accelerate many operations.

2.2 JS Object Models: Shapes and Slots

One of the main features shared between SELF and JS implementations that enable the use of PICs is the *shape*[1] representation of objects. Every object has (a) a shape that represents the layout of that object in memory, and (b) slots that hold actual values. Each shape represents a mapping from property-name strings to the slots where the properties can be found within the object, and shapes are shared amongst objects that have the same mappings. Shapes are immutable; if a property is added to an object, then a different shape needs to be associated with the object. If an existing shape matches the new mapping, that shape is associated with the object. If no existing shape is found, then a new shape with the correct set of properties is created and associated with

[1] Called Hidden Classes in V8 and Structures in JSC.

the object. Because shapes are immutable, objects with the same shape can be handled similarly and operate under the same assumptions. For PICs, this immutability allows for shapes to be used in guard conditions – comparing shapes – to select the appropriate fast path through cached code or execution of the fallback path. The JS engines also exploit an insight from the work by Hölzle et al.: not only do PICs accelerate polymorphic operation sites, but they also provide a rich collection of per-site type information. This type information can be leveraged by the runtime system to enable optimizations such as speculative compilation of hot functions that exhibit function-wide type stability.

2.3 Multiple Execution Engines Within an Implementation

To maintain quick start-up times while supporting robust optimizations for hot code, JS engines employ tiered compilation strategies, changing the execution-engine running code dynamically. This Section provides context about JS engine design through a discussion of the optimization and code-generation strategy in SpiderMonkey, the JS VM that powers Firefox and is the JS embedding of MongoDB, CouchDB, and GJS among other projects. Other high-performance JavaScript engines have similar designs.

2.3.1 Tiered Interpretation and Compilation. Optimization pipelines for modern JS engines typically operate on an executable program segment — such as a function, module, or `eval`. In SpiderMonkey, JS source text is compiled, at run time, into a set of internally used *scripts*, each corresponding to an executable body. Initially, scripts are compiled to a stack-based VM bytecode, but, throughout execution, a script may gain additional representations when transitioning between four different execution engines (Figure 1):

- **Interpreter**: For the first few invocations of a script, SpiderMonkey uses a threaded, single-bytecode interpreter implemented in C++.
- **Baseline Interpreter**: The next tier is a JIT-compiled interpreter that is created at process startup. This Baseline Interpreter starts collecting bytecode operand-type information by compiling and using ICs for supported bytecode operations. Allowing the Baseline Interpreter to use ICs results in larger performance gains over the Interpreter.
- **Baseline JIT**: After sufficient execution in Baseline Interpreter, a script may be compiled by the Baseline compiler, a template JIT compiler that stitches together native code sequences for each bytecode. For IC-supported operations, Baseline directly emits a call to the compiled IC code. Most of the ICs that were pre-populated while the code already ran in the baseline interpreter continue to be used in the baseline version of the function.

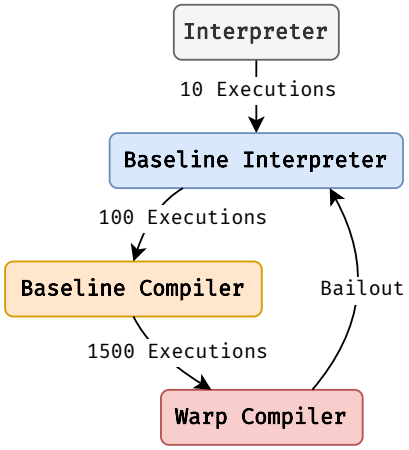

Figure 1. SpiderMonkey's execution tiers and transition conditions. These counts can also be incremented on loop back-edges, and thus state transition can also happen earlier than an execution count suggest. SpiderMonkey supports tiering up in the middle of a long-running loop through On-Stack Replacement.

In the Baseline JIT, ICs are the only form of optimization, other than the dispatch reduction from the template-JIT style compilation including simple register allocation. The use of ICs is sufficient to provide a substantial performance boost compared to the interpreter.

- **Warp and Ion**: For frequently executed scripts, a version is compiled by Warp and Ion, SpiderMonkey's optimizing compiler pipeline. Warp parses bytecode to produce a high-level intermediate representation, *MIR*, which is optimized and lowered into a low-level IR, *LIR*. Using MIR and LIR, Ion performs a collection of standard compiler optimizations such as Global Value Numbering [7], Dead Code Elimination, Scalar Replacement[9], and efficient register allocation and code generation. Ion is a type-specializing, speculative compiler that produces executable bodies optimized to observed types whose information is stored within ICs and communicated through the execution tiers. If the speculation fails, Ion-compiled script bodies *bailout* to the Baseline Interpreter tier, restoring sufficient state to resume execution. If there are too many bailouts from Ion then the script is invalidated, setting the stage for recompilation under a new set of assumptions.

2.3.2 Inline Caches: A List of Stubs. SpiderMonkey's ICs are associated directly with a specific bytecode in a script. All ICs are polymorphic. Structurally, each IC has a linked list of stubs[2], a State structure — which contains the number

[2]The code for an inline cache is not actually inlined in memory. The name is preserved because they are in-line in a logical sense, in spite of their out-of-line code organization.

```
class InlineCacheStub {
  NativeCode* stubCode;  // Native Code
  uint32_t enteredCount; // Jumps to stub
  InlineCacheStub* next; // Next stub
  StubInfo* stubInfo;    // Metadata
}
```

Figure 2. SpiderMonkey's in-code representation of an IC stub.

```
GuardToObject InputId0 --> ObjId0
GuardShape ObjId0, Field0
LoadFixedSlotResult, ObjId0, 8
ReturnFromIC
--Stub Fields---
Field0: Shape 0xabcdef0123
```

Figure 3. An example of CacheIR for an object property read: `obj.prop`

of stubs, the mode (see Section 3.1.2), and Trial Inlining state (see Section 4.2). Stubs are heap-allocated data structures consisting of a pointer to native code, a counter incremented when the stub is *entered*, a pointer to the next stub in the chain, and an associated metadata structure containing information such as a pointer to the stub's IR (see Section 3) and the data on which the stub operates.[3] Figure 2 shows an in-code representation of an IC stub.

Initially, an IC consists of a *fallback* stub that invokes a fallback handler — sometimes called the *lookup* method in the literature. The fallback handler computes the return value and tries to attach a new specialized IC stub for the observed inputs. The temporal-locality property states that recently used data are likely to be used again soon. Thus, the newly attached stubs are prepended to the IC chain so that the most recently added stubs are tried first.

Correctness requires that, regardless of the structure used to implement an inline cache, ICs must preserve the semantics in the original bytecode: all results — and side effects — of the interpretation of the original bytecode, must be the same as if the bytecode were interpreted. There is freedom in how this result is accomplished. Since ICs are type-specialized fast paths, in some implementations, such as SpiderMonkey for example, more complex ICs can even replace function calls with the effects of that call – effectively inlining the call, even in the Baseline interpreter execution tier. In SpiderMonkey, inlining is limited to calls to native (C++ implemented) code, but this covers many important use cases such as pushing elements into, or popping from, an array.

3 CacheIR: A Linear Bytecode for Inline Caches

CacheIR is a simple typed bytecode specialized for compiling inline caches. CacheIR bytecode is 'linear' in that it has only two control-flow primitives:

- **Guards**: Instructions that verify a stub invariant, preventing the execution of the stub if the guard does not hold.
- **Return**: A single bytecode that returns from the IC stub code.

Containing no other control flow instructions, a CacheIR sequence is akin to an extended basic block [18]. Each IC has a single entry point, has multiple exit points through guard-failure paths and the return operation. Once the execution passes all the guards, every instruction in the IC executes once in order.

CacheIR bytecodes operate on typed *Operands*, which are either input values or the return value of a CacheIR bytecode operation. The number of implicit input operands in the CacheIR for an IC is determined by the arity of the bytecode to which the IC is attached. For an IC attached to a bytecode that produces a value, the CacheIR has an output operand for that value as well. In addition to operands, CacheIR stubs have *stub fields*, which are values associated with and used within the stub. For Baseline ICs, stub fields facilitate the sharing of native code for stubs that are identical except for offsets and pointer values, and simplify the process of integrating stubs into the garbage collector.

Figure 3 shows an example of a CacheIR illustrating guards, operations, and stub fields. The `GuardToObject` operation is an example of a guarded cast; either the input is of Object type, producing a new Object typed operand, or the guard condition fails and control is given to the next stub. The `GuardShape` operation tests if the object-operand's shape matches the shape stored in the stub field. `LoadFixedSlotResult` loads a value out of the fixed slot at offset 8 in the object[4], placing the result into the implicit result register (hence the `Result` suffix).

Currently, SpiderMonkey's CacheIR has more than 300 CacheIR instructions, including 64 guard conditions, covering a large number of behaviours inside the engine. Moreover, the IR's simplicity enables a straightforward implementation path to add support for further operations.

3.1 Generation of CacheIR

CacheIR is generated in the fallback path for an IC miss. While details vary slightly, the fallback handler logic generally follows these steps: (i) compute the result for the current operation; (ii) invoke the appropriate CacheIR IR Generator — this generator analyses the input values, opcode, and resulting value, and uses a simple hand-crafted pattern matching

[3] A detailed description of the setup appears in the SpiderMonkey source code https://searchfox.org/.

[4] This slot was determined during the generation of this cache and is correct because of the previous shape guard.

code to instantiate a matching sequence of CacheIR operations that handle the input values; (iii) generate native code from the sequence of CacheIR operations; (iv) attach the resulting stub to the front of the IC chain.

3.1.1 Raised Abstraction == Higher Productivity. Language runtime development must focus on performance improvement. Sometimes a performance cliff can be eliminated by designing the engine to generate specialized code for specific cases. SpiderMonkey, for example, often generates specialized code by adding support for a specific case in the inline cache generator. For instance, consider an illustrative example: Assume that a performance analysis determines that an important site often performs arithmetic with `null`. Such operations are unexpected and thus are not handled by the default IC generation policy. CacheIR simplifies the process of inserting an IC to generate specialized code for the execution of operations such as `a + b` where either `a` or `b` is `null`, leading to performance improvement in this example.

Adding such support often only requires a modification to the existing Binary Arithmetic CacheIR generator[5] that adds a case to emit the correct CacheIR, as shown in Figure 4. The general-shape CacheIR generation pattern matches the actual results that the fallback stub observed when running the operation (lines 2-9), and then generates CacheIR that provides a fast path for the specific observed case (lines 17-21). In some cases, correct handling may require adding a CacheIR operation. In SpiderMonkey, adding a CacheIR operation requires a modification of an operator description file, and code generation for that operation through a platform-independent MacroAssembler.

By adding support to the CacheIR generator, we've also added support for specializing this operation to Warp, assuming the CacheIR operations are all successfully handled by Warp.

Working with CacheIR raises productivity for writing ICs in other ways as well. CacheIR has a simple register allocator to allow managing values inside of caches. Furthermore, CacheIR also automatically creates the failure paths required to restore the input registers to their original values on failure allowing every stub in the IC chain to start from the same state.

Using an Intermediate Representation for inline caches also eases the development of tooling to analyze the behaviour of inline caches because a structured machine-independent representation is appreciably easier to investigate than generated native machine code.

3.1.2 Stub Generation Policies: Avoiding Pathological Outcomes. Each IC chain has some associated state: the number of stubs attached, and a *mode*. There are three modes, coarsely characterizing observed type history for an IC chain:

[5]Used for infix binary operations.

```
1  AttachDecision BinaryArithIRGenerator::
       tryAttachNullInt() {
2    // Only Handle Add
3    if (op_ != JSOp::Add) {
4      return AttachDecision::NoAction;
5    }
6    // Only handle LHS null RHS int32.
7    if (!lhs_.isNull() || !rhs_.isInt32()) {
8      return AttachDecision::NoAction;
9    }
10
11   ValOperandId lhsId(
12     writer.setInputOperandId(0));
13   ValOperandId rhsId(
14      writer.setInputOperandId(1));
15
16   // null + int32rhs = int32rhs
17   writer.guardIsNull(lhsId);
18   Int32OperandId rhsIntId =
19     writer.guardToInt32(rhsId);
20   writer.Int32Result(rhsIntId);
21   writer.returnFromIC();
22 }
```

Figure 4. A simplified fictional example of the CacheIR generation process for a `null` + int opportunity

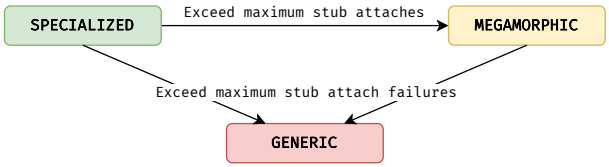

Figure 5. SpiderMonkey's IC modes and transition criteria.

1. Specialized: In this mode, the CacheIR generators attempt to create stubs that are tightly specialized to the observed values, but can only handle specific, tightly guarded cases. For example, a specialized stub for a property read has a guard on the receiver being an object and a guard on the object's shape (as in Figure 3).
2. Megamorphic: In this mode, the CacheIR generators create stubs that are not as type specialized as Specialized stubs. Often this involves creating a stub that calls a routine in the VM runtime, which is slower than a Specialized stub, but still faster than interpretation. For a property read, Megamorphic stubs guard on the receiver being an object before calling a VM routine to produce the result via a lookup in a global property cache.
3. Generic: In this mode, stub attachment is disallowed, and a call to a VM runtime routine will simply provide the result value without attempting to attach a stub.

Each IC chain is initially Specialized, from which point it may either (i) remain Specialized, (ii) transition to Megamorphic, or (iii) transition directly to Generic. When the IC chain transitions from one mode to another, it discards all the currently attached stubs other than the fallback stub. As shown in Figure 5, an IC chain transitions from Specialized to Megamorphic when it exceeds the maximum number of attached stubs that are allowed for a specialized IC chain. The goal of this limit is to prevent the creation of long chains of stubs whose traversal may cause high overhead. Each IC chain also keeps track of the number of times a fallback stub has been reached and fails to attach a stub. Failure to attach any stub can happen because JavaScript is an extremely dynamic language with a large number of possible behaviours, many of which are effectively never seen in practice. As a result, optimization effort is typically focused on observed patterns, leading to rare combinations of operations lacking IC support. In addition, there is a maximum number of failures-to-attach that the stub chain will tolerate; when that is hit, the stub chain transitions to Generic mode to avoid wasting further resources on attempting to attach new stubs.

3.2 Compilation of CacheIR into Native Code

SpiderMonkey has two CacheIR compilers: one compiler shared by Baseline Interpreter and Baseline, called the *Baseline CacheIR Compiler*, and one specialized to Ion — SpiderMonkey's top-tier compiler used only for the hottest scripts — called the *Ion CacheIR Compiler*. The two compilers share a considerable amount of engine code but are specialized to the required calling convention of the target IC system. Each compiler also adopts a different policy for the handling of Stub Fields.

The Baseline CacheIR compiler handles stub fields by generating native code that loads the values out of the stub metadata. The Ion CacheIR compiler adopts a policy of directly embedding stub field values in the IC stub code. Field-value embedding makes Ion IC stub code ineligible for sharing but requires less indirection, and thus Ion ICs ultimately execute faster than Baseline ICs. The Baseline CacheIR compiler's approach is slightly less performant than Ion's approach but has the benefit that caches with the same CacheIR that differ only in the values of their stub fields can share native code. Stub code sharing dramatically reduces the amount of native code required by Baseline ICs, saving memory and the time required to generate native code when stubs can be shared. Furthermore, stub code sharing also avoids manipulating page tables to mark pages as executable — a major cost of dynamic code generation.

Both CacheIR compilers use a simple register allocator to track where each operand is during execution — in a register, in a stack slot, on the language stack, etc. This register allocator can provide extra registers to caches where required. If a guard fails, the IC needs to restore the input registers to their original values, allowing the next stub in the chain to start in a known state. The register allocator has the information that it needs to generate these failure paths automatically. Failure paths are shared between guard instructions if the register state has not changed between the guards.

4 WarpBuilder: Consuming CacheIR Directly as Type Feedback

WarpBuilder, a front-end for the Ion Compiler, converts bytecodes into MIR and was built around the insight that CacheIR provides an excellent source of type information for an optimizing compiler.

It replaces a component called IonBuilder, which used a hybrid Type Inference (TI) system, proposed by Hackett et al., that uses both static-analysis and run-time-collected type information to infer facts about object types and shapes [15]. The TI system allows for complicated reasoning about objects and nested property accesses. However, its power came at a cost: Type Inference consumed memory to power optimizations which could only occur once functions made it to top-tier compilation, but an even larger cost was the engineering costs associated with TI.

To be sound, the TI system needed information to be correctly propagated throughout the engine, which meant that TI code was necessary for many parts of the engine. Moreover, any failure to properly maintain type data could lead to security problems because the Ion optimizing compiler would consult the TI system and use the provided invariants to elide checks that would otherwise be necessary. This lookup mechanism was particularly pernicious because erroneous handling of a value in one place could be exploitable by code very far away, as a result of the poisoning of the global analysis.

In WarpBuilder, type data is exclusively sourced from the inline caches generated as part of lower-tier executions, and the consumption of that type data is local to the MIR for a particular bytecode. WarpBuilder can precisely build specialized code by directly compiling the CacheIR for an eligible stub to MIR, converting guard failures to bailouts. With this compilation strategy, the MIR code is immediately specialized to the observed types in the program, and those type checks are made visible to the Ion optimizer. The optimizer re-orders them and eliminates redundancy to further optimize the code. Furthermore, complex ICs with complicated guard conditions do not require more complex analysis glue code in WarpBuilder to take advantage of stored types – Warp builds guards correctly and automatically for any CacheIR instructions that it compiles.

Compared to IonBuilder it is an appreciably simpler approach, eliminating the requirement for globally correct reasoning, and limiting the scope of impact for erroneous code. Moreover, using CacheIR for ICs in lower tiers is a performance optimization that provides direct value, whereas tracking TI information is pure overhead before tiering up to the

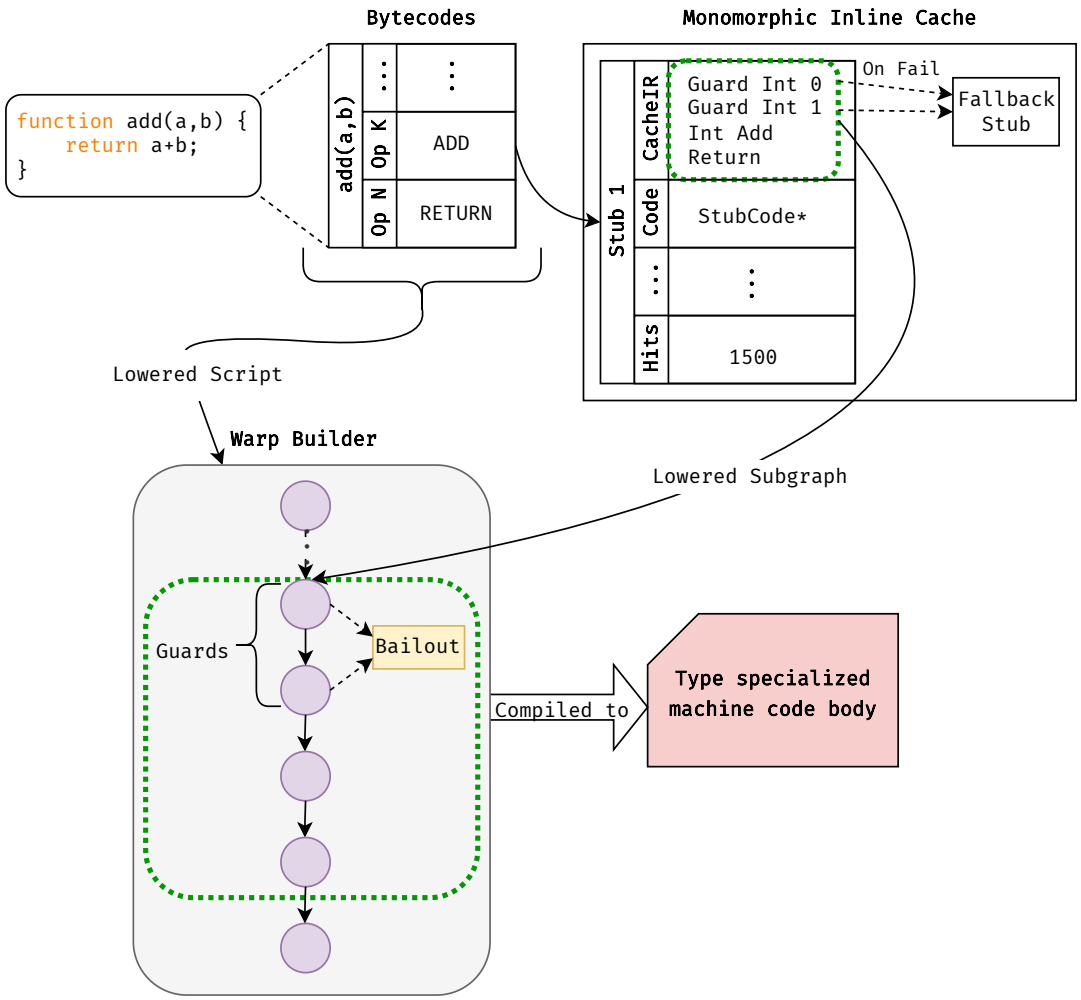

Figure 6. A diagram of the CacheIR powered transformations in SpiderMonkey

Ion compiler. Therefore, WarpBuilder also provides improvement over IonBuilder on memory consumption, startup time, and flat code profiles.

4.1 The Compilation Pipeline

The WarpBuilder compilation process is split into three phases:

Phase 1: Snapshot Building. At the start of top-tier compilation a *WarpOracle* creates a snapshot of information, including the CacheIR stub information for the script. The WarpBuilder uses this snapshot to generate MIR.

Adopting this snapshot-generation process allows the remainder of the Warp compilation to be executed in a different thread from the main thread. This latency optimization is important because the main thread performance is extremely performance-sensitive in a production JS engine.

Phase 2: MIR Generation. In a separate thread, the WarpBuilder uses the bytecode and the WarpSnapshot, to create MIR. During this phase, when a bytecode that supports ICs is encountered, Warp can do one of the following:

1. If the IC has multiple active stubs, Warp emits code that constructs and uses an Ion IC chain.
2. If the IC has only a single stub attached, or has only a single active stub at the front of the IC chain, then Warp lowers[6] that stub's CacheIR to MIR and inline.
3. If no IC stubs are attached, Warp generates an unconditional bailout, thus enabling speculative dead code elimination while maintaining appropriate handling if that speculation is false.

Stub lowering is the relatively straightforward process of converting the sequence of CacheIR ops in the stub into equivalent MIR nodes. Rather than jumping to another implementation or stub, **Guard** instructions are lowered such that a guard failure results in a bailout with the execution continuing in the Baseline Interpreter.

[6]This step is referred to as *transpilation* in the source code.

Previous work used inline caches to drive optimization by taking advantage of the insight that inline caches capture exactly the set of run-time observed types at a particular bytecode [17]. However, unlike previous work that simply refined types using the data contained in ICs, SpiderMonkey lowers CacheIR and is able to generate specialized code in Warp without the engineering cost of writing, yet again, type-specialization code for the Ion Compiler, thus allowing for direct fast-path code generation. Warp also does not need to analyze the semantics at the level of source language because lowering CacheIR to MIR is a mechanical translation — each CacheIR operation is lowered to a predictable MIR node. A lowering pipeline for a CacheIR operation is reused between different kinds of ICs. For example, for a GuardShape CacheIR operation, the pipeline is defined once and used in any operation backed by an IC that needs to guard on an object's shape.

Phase 3: Optimization and Code Generation. The MIR is then optimized by the Ion optimizer, lowered into LIR, and compiled into native code.

4.2 Trial Inlining

True monomorphism is highly desirable because it allows for tightly specialized code generation and provides a reasonable basis for making type-based assumptions in the compilation pipeline. ICs must reflect, as accurately as possible, the expected set of types during Warp compilation because type specialization is handled by IC analysis. An obvious challenge is how to handle the inlining of functions that may be polymorphic across the program, but monomorphic in a particular calling context (see Figure 7).

To handle this case, the SpiderMonkey team developed *Trial Inlining* – a kind of context-sensitve profiling driven by CacheIR. Trial Inlining allows the Baseline compiled code to associate distinct sets of ICs to distinct call sites that call the same function. After Trial Inlining, each set of ICs uses the CacheIR system to collect type information local to the call site that the set is associated with. Therefore, if Warp inlines that call, it can create type-specialized code for that call site, handily exploiting local monomorphism. Trial Inlining can nest within another Trial Inlining, further specializing code of calls within the Trial Inlined calls[7].

5 CacheIR and WarpBuilder Evaluation

The evaluation of CacheIR and Warp in this Section supports that: (i) CacheIR is a useful abstraction for developing inline caching systems; and (ii) CacheIR enables the development of a JIT compiler that has a simple high-level design that delivers excellent performance. .

[7]To avoid unbounded memory consumption, however, nesting is limited to a maximum inlining depth of 4.

```
1  function adder(a,b) {
2      return a+b;
3  }
4
5  function strings() {
6      var str = "";
7      for (var o of ["a","b","c"]) {
8          str += adder(o,",");
9      }
10     return str;
11 }
12
13 function numbers() {
14     var n = 0;
15     for (var o of [1,2,3]) {
16         n += adder(2,o);
17     }
18     return n;
19 }
```

Figure 7. An example of local monomorphism: the Add op within `adder` may operate either on strings or on integers; however, within each call context the types that reach the Add are monomorphic.

5.1 Benchmarks, Hardware, and Software

Data presented in this section was collected from a machine that runs Fedora 36 (Kernel 5.18.11-200) and is equipped with an Intel i5 12400 16 GiB of DDR4 memory. Moreover, this section uses Speedometer 2.1 – Speedometer – and JetStream 2.1 – JetStream – benchmark suites to evaluate performance, Speedometer as a proxy for real-world workloads, and the AreWeSlimYet (AWSY) [4] *tp6* benchmark to evaluate memory consumption.

Speedometer contains 16 subtests, all of which are `To-Do list` applications written in different popular JavaScript frameworks like React, React with Redux, Ember.js, Backbone.js, AngularJS, Vue.js, jQuery, Preact, Inferno, and Flight. These frameworks are ubiquitously used in Web development and Speedometer is meant to reflect real-world WebApp workloads by conducting operations on a `To-Do list` both synchronously and asynchronously. To characterize performance, Speedometer computes a final score:

$$\frac{6000}{geomean(medians) * correctionFactor} \quad (1)$$

where *medians* is the set containing a value for each subtest, computed by the median run time in milliseconds across all iterations; and *correctionFactor* is a scalar value used to scale down the final score. Faster subtest run times — smaller geometric mean values — increase the final score.

JetStream contains 64 subtests evaluating computationally intensive workloads.[8] On each invocation of JetStream,

[8]A full list is found here: https://browserbench.org/JetStream2.1/in-depth.html

each subtest runs for 120 iterations and computes a score value:

$$subScore_i = \frac{5000}{time_i} \tag{2}$$

where $time_i$ is the time in milliseconds to complete the i^{th} iteration. JetStream computes a single score across all iterations for each subtest B:

$$score_B = geomean(firstIteration_B, worstFour_B, average_B) \tag{3}$$

where $firstIteration_B$ is $subScore_0$; $worstFour_B$ is the arithmetic mean of the lowest four $subScore$ values; and $average_B$ is the arithmetic mean across all iterations. To characterize overall performance, JetStream computes a final score:

$$geomean(scores) \tag{4}$$

where $scores$ is the set containing the $score_B$ values for each subtest.

AreWeSlimYet is a project maintained by Mozilla to track memory usage across Firefox builds. The tp6 test automates opening browser tabs and loading popular Web pages to simulate common user behavior. AWSY collects memory usage statistics that discriminate between sources of memory consumption. This evaluation focuses on the memory usage attributed to the SpiderMonkey JavaScript engine. Each memory usage value is the geometric mean across all iterations – in this evaluation 15 – of an AWSY run.

5.2 Experimental Methodology

The results presented throughout this section use Mozilla's `mach` tool with the `raptor` subcommand to collect benchmark metrics for Speedometer, JetStream, and the `awsy-test` subcommand to collect metrics for AWSY. Each data point in the figures represents the metrics described above computed from 15 `mach` invocations for both Speedometer and JetStream.

5.3 Execution Engines, Inline Caches and Their Impact on Performance

Figure 8 examines the performance contribution of various execution engines on Speedometer and JetStream. The speedup factor of each tier is calculated over the lowest tier in the engine, the C++ Interpreter, which is a simple interpreter loop and has no support for inline caching. Enabling higher optimizing tiers leads to better performance for both benchmark suites. At the Baseline Interpreter tier, Speedometer and JetStream observe a 1.63× and 1.95× improvement respectively, directly as a result of the CacheIR system. By exploiting and refining CacheIR information collected in the Baseline Interpreter, the Baseline Compiler further increases performance by 1.5× and 1.75×. Finally, when Warp specializes and optimizes native code by lowering the CacheIR, it increases performance by another 13% and 2×.

To show the value of CacheIR to each tier, Figure 8 provides three synthetic tiers where CacheIR's contribution to each tier was removed. The * in the Baseline Interpreter* and Baseline Compiler* tiers indicates that CacheIR is disabled engine-wide. The † represents that only lowering CacheIR in Warp is disabled – in this synthetic tier, Warp compiled function bodies generate inline caches powered by CacheIR, but are devoid of type information provided by CacheIR lowering.

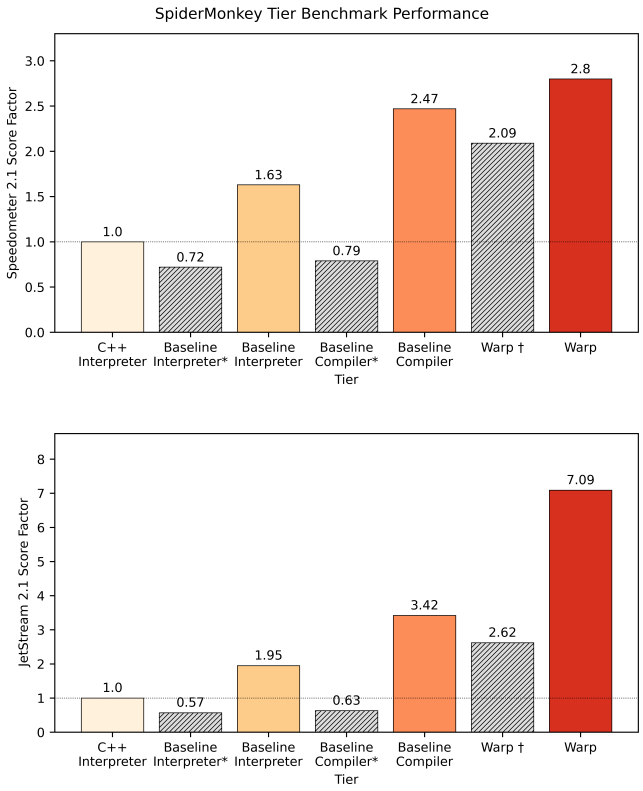

Figure 8. Per-tier benchmark score improvement over the C++ Interpreter tier. Tiers marked with a * signify that CacheIR is disabled. Warp† represents disabled CacheIR lowering.

The results highlight how much the design of SpiderMonkey leans on the CacheIR system: CacheIR is the most important feature to improve performance within the engine. Without inline caching the Baseline Interpreter has high overhead because it spends enough time jumping between JIT and C++ code that it is slower than the C++ interpreter in both benchmarks. The Baseline Compiler, which should derive some benefit from the reduction of dispatch overhead, shows relatively little benefit over the Baseline Interpreter for very similar reasons.

The disparity between Warp's ability to improve benchmark performance on JetStream and Speedometer reflects that a compiler's ability to improve performance is workload-dependent. JetStream has computationally intense kernels that benefit from fine-grained, top-tier, optimizations and

Table 1. Top ten CacheIR sequences for Baseline ICs with respect to the number of occurrences when running Speedometer. As examples, Listings 1 and 2 display the two most common sequences. A scripted function is backed by bytecode — in contrast with a Native function that is a JavaScript function backed by C++.

Operation	Occurrences
ToBool(Boolean)	57554
Scripted function dispatch	46264
Load from an object's fixed slot	42629
Load from an object's dynamic slot	28899
Load from a prototype's dynamic slot	26911
Store to an object's fixed slot	15487
Null value check	14933
Integer comparison	12820
ToBool(Object)	12464
Get global name	11139

it delivers a 2× improvement over the Baseline Compiler. Anh et al. observe that top-tier compilers improve the performance of real-world code much less than computational kernels would suggest [5]. Given that Speedometer is designed to better reflect the flatter profiles of real-world code, Warp's results on Speedometer concur with Anh et al.'s observation, providing a more modest speedup of around 13% over the Baseline Compiler.

5.4 CacheIR: Simplifying IC Development and Enabling Stub Code Sharing

CacheIR makes it easier to develop inline caches for Firefox and can handle a diverse number of cases with relative ease. As a result, running Speedometer and JetStream generates 437 and 531 distinct CacheIR strings respectively, each covering a different case observed in the benchmark run. These are 437 and 531 specialization cases whose CacheIR strings can be forwarded to Warp and lowered throughout the compilation pipeline, massively reducing the complexity of the compiler design required to achieve excellent JavaScript performance.

The diversity of cases covered also helps explain the magnitude of increase provided by CacheIR for Baseline Interpreter performance in Figure 8. CacheIR improves performance even without traditional JIT compilation because it optimizes a large number of diverse cases with native code. A major strength of the CacheIR design is that it makes covering a new important case with an inline cache both performant and easy. Thus, the SpiderMonkey team has managed to cover a broad range of cases, improving the performance of every tier above the C++ interpreter with one action.

In addition to providing a modular IR to share with higher-tier compilers, CacheIR also enables native stub code sharing

Listing 1. First most common CacheIR sequence: Converting a Boolean stored variable to a boolean result.

```
GuardNonDoubleType  inputId 0,
                    type Bool
LoadOperandResult   inputId 0
ReturnFromIC
```

Listing 2. Second most common CacheIR sequence: Calling a specific Scripted function.

```
LoadArgumentDynamicSlot resultId 1,
                        argcId 0,
                        slotIndex 1
GuardToObject           inputId 1
GuardSpecificFunction   funId 1,
                        expectedOffset 0,
                        nargs&FlagsOffset 8
CallScriptedFunction    calleeId 1,
                        argcId 0,
                        flags,
                        argcFixed N
ReturnFromIC
```

by mapping CacheIR sequences to native stub code. In the Baseline tiers, each CacheIR sequence is compiled to native code once, then each IC stub with the same CacheIR sequence holds a pointer to the compiled code. Table 1 illustrates the occurrence count for the top 10 most common CacheIR sequences for Baseline IC stubs in Speedometer. The first and second most common sequences are shown in Listing 1 and Listing 2. Native code corresponding to the CacheIR sequences in these listings is shared between 57,554 and 42,692 Baseline IC stubs, respectively. By eliminating redundant code compilation for different ICs, CacheIR's design significantly reduces the memory footprint required to support the inline caching system (Section 5.5.1).

5.5 WarpBuilder: Exceeding IonBuilder Performance

This section presents a performance evaluation from Firefox 83, the release where WarpBuilder was enabled by default. Firefox 83 is used as a baseline to show what a relatively unoptimized, proof-of-concept version of Warp is capable of delivering. Furthermore, it was the last release where a runtime switch was available to toggle between the old IonBuilder compiler frontend (and associated Type Inference system) and the WarpBuilder frontend with the Type Inference system disabled.

5.5.1 Memory Consumption. As part of the design of WarpBuilder, there are two major sources of potential memory savings: (i) removing the need to track and store global type inference to support the TI system in IonBuilder, as discussed in Section 4; and (ii) stub code sharing, discussed in Section 3.2, meaning that Baseline stubs with identical

CacheIR can share native code. Enabling WarpBuilder reduces memory consumption to 0.91× that of IonBuilder for the AWSY tp6 test suite running on Firefox 83.

5.5.2 Benchmark Performance Evolution. Since the release of Firefox 83, further engineering and tuning has greatly improved the WarpBuilder system. To characterize these improvements, this evaluation contrasts the performance results of Firefox 83 and Firefox 111 in Figure 9 on Speedometer and JetStream.

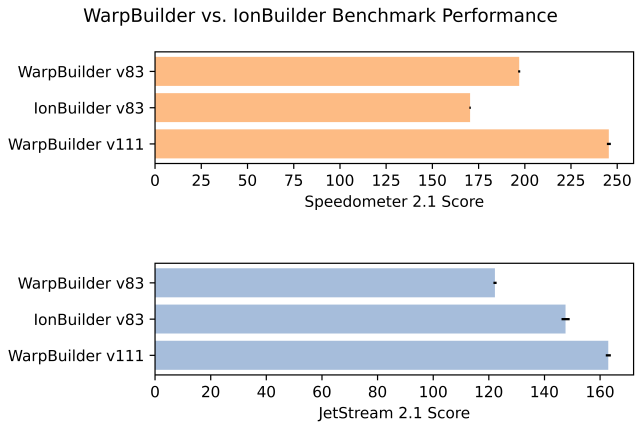

Figure 9. Results from running Firefox with WarpBuilder and with IonBuilder for the Speedometer and JetStream benchmark suites. Each horizontal bar is labeled with the Firefox version number and characterizes the mean score of 15 runs containing the 95% interval as an error. A higher score is better.

For Speedometer running on Firefox 83, enabling WarpBuilder improves the score to 1.15× that of IonBuilder. The improvements on Speedometer stem from the reduction of source code and computation throughout the engine enabled by building Warp around the CacheIR. In contrast, IonBuilder has to track the global type inference data and requires that all functions track type information even if it is only useful in very hot functions. With WarpBuilder, profiling information via CacheIR is used both to (i) optimize Warp; and (ii) speedup the Baseline Interpreter and Baseline JIT. Warp does more work off-thread through the Snapshot-based design, allowing the main thread to make progress in program execution while compilation occurs. Moreover, Warp's design is less prone to overspecialization, leading to fewer recompilations than in the IonBuilder system. Building upon WarpBuilder, Firefox 111, which has WarpBuilder enabled by default, achieves a score 1.44× higher than that of IonBuilder.

For JetStream running on Firefox 83, enabling WarpBuilder reduced the score to 0.83× that of IonBuilder. A substantial fraction of JetStream consists of computational kernel benchmarks which were excellent targets for the combination of Type Inference system and IonBuilder; these types of workloads were intentionally de-prioritized during the development of Warp in lieu of code that more reflected the state of real websites. Nevertheless, WarpBuilder's state as of Firefox 111 improves the JetStream score by 1.10× over IonBuilder. WarpBuilder's performance is regained through developing and applying new inline caches via the CacheIR system and expanding the support for compiler optimizations that had yet to be implemented using the CacheIR-based system instead of the Type Inference-based system.

6 Related Work

According to Deutsch and Schiffman, the key result behind their Smalltalk-80 implementation is *Dynamic Change of Representation* [12]. Under this rubric, JIT compilation (dynamic translation in their terminology) and Inline Caching are different representations of method dispatch. They introduce a one-entry IC, generated by their JIT compiler, that starts out *unlinked* and is then patched with generated native code containing an updated target. They also discuss inlining small methods for commonly used selectors such as +. In a sense, the generated CacheIR for a particular bytecode is an inlined and type-specialized version of a selector, a *dynamic change of representation* similar to Deutsch and Schiffman's system. However, Deutsch and Schiffman do not describe a unifying mechanism underlying their inline caching. Thus, in their design, optimizations are implemented by hand.

Polymorphic Inline Caches (PICs) were first described in the context of the SELF language by Hölzle et al. [16]. The main idea of their PIC scheme is to provide a mechanism wherein, for polymorphic call sites, instead of overwriting the call site directly with the address of a single resolved method, a machine-code stub is created, and the address of the stub is used inline. Their work is foundational for PIC support in modern languages systems; the underlying system outlined in this paper is a clear reflection of the ideas presented by Hölzle et al. However, unlike CacheIR, Hölzle et al. scheme does not rely on a specialized IR to create IC stubs. The presence and architecture of CacheIR in SpiderMonkey enables high levels of native stub code sharing that is contingent on an equality check of the IR itself. This code sharing leads to less frequent stub compilation and greater memory efficiency to drive inline caches.

In later work, within the context of the SELF language, Hölzle et al. describe the use of ICs and PICs to drive compilation decisions in the runtime system [17]. The insight is that the SELF PIC scheme could be used as a type-information collector in addition to accelerating operation sites. By leveraging this type information they are able to speculatively compile hot functions with potentially stabilized types. CacheIR builds greatly upon this insight and enables IC stub code structure to be reached throughout the entire compilation

pipeline. CacheIR is used as a shared source of truth in the engine, having prevalent use from the Baseline Interpreter through to the optimizing Ion compiler providing an efficient means for stub-code lowering.

LIL [14] is an architecture-independent language for writing Virtual Machine stubs. Compared to CacheIR, LIL has a much broader set of responsibilities — it covers many assembly-code stub responsibilities, such as object allocation and garbage-collector barriers — and thus supports arithmetic operations, conditional control flow, multiple calling conventions and more. LIL is a mechanism for replacing assembly-written stubs, but it does not appear that LIL is tightly connected to their IC infrastructure.

The linear intermediate representation CacheIR uses is akin to a *trace*, from a tracing JIT compiler, such as TraceMonkey or PyPy [6, 13]. Traces have very similar side exit mechanisms, also called *guards* where a precondition is required to proceed further into the trace. However, unlike in a tracing JIT the failure of a guard in CacheIR simply starts execution again at the beginning of the next stub. Unlike a fully developed tracing JIT, CacheIR is constructed incrementally by program developers rather than derived directly from execution. This is because CacheIR operates at a sub-bytecode granularity, and as such there is no infrastructure to collect traces automatically.

Wuthinger et al. [20] introduce Truffle, a framework that provides a modular, unified infrastructure and architecture for building dynamic languages VMs. Their architecture enables node rewriting on an AST to specialize execution at runtime. To enable specialization, Truffle supports polymorphic inline caching by expanding a node representing an IC-supported operation to multiple nodes each specialized to an observed type. Similar to IC stubs in SpiderMonkey, Truffle's IC nodes are chained, connected by an edge representing the failure path for the preceeding node. However, the IC scheme in Truffle is more akin to Quickening [8]; rather than backing an IC with dynamically compiled stub code, a generic VM call is replaced with a specialized VM call. Thus, unlike WarpBuilder and CacheIR, there is no abstraction within Truffle and Graal that enables native code sharing between ICs.

An independently developed idea similar to Trial Inlining, called *Polyvariance*, exists in the JSC JavaScript engine powering WebKit [3]. The goal of both Trial Inlining and Polyvariance is to use ICs to specialize polymorphic operations on a per-call-site basis. Polyvariance works by inlining small functions when tiering up from the JCS baseline compiler to the next-tier DFG compiler. These inlined functions have uninitialized ICs associated with DFG. If DFG detects that an IC fast path is taken through the inlined function, it fills in the uninitialized ICs, thus enabling more specialized code generation in the next, highest tier compiler. Polyvariance differs from Trial Inlining in a few notable ways. First, Trial Inlining is an intermediate step between Baseline and Warp whereas Polyvariance is applied later in JSC's JIT pipeline. Thus, SpiderMonkey's Baseline compiler can detect some of the benefits of Trial Inlining and preemptively collect more precise profiling data before tiering up. Second, due to SpiderMonkey's IC architecture, Trial Inlining is able to specialize moderately deep nestings of call sites whereas Polyvariance is a one-level-deep inlining strategy. Third, Trial Inlining does not inline, in the sense of function inlining, until tiering up to Warp. Instead, Trial Inlining creates a specialized set of ICs for a call site, leaving the actual function inlining decision to Warp.

CacheIR has further utility not discussed in this paper. *Cachet* is a domain-specific language intended to help build a formally verified JIT compiler [19]. In their prototype implementation, they build a CacheIR compiler for Cachet to help on the road to formal verification for the SpiderMonkey JIT compilers.

7 Conclusion

This work introduces CacheIR, a novel IC design centered around an intermediate representation that simplifies IC development and enables the reuse of compiled native code through IR matching. By leveraging CacheIR's design, this paper also describes WarpBuilder, a JIT compiler front-end that generates specialized code by lowering CacheIR. Moreover, Trial Inlining extends the inline caching system for context-sensitive IC inlining through the power of CacheIR. In SpiderMonkey, the combination of CacheIR and WarpBuilder has proven to be a beneficial design by significantly improving performance and reducing security risks.

Acknowledgments

The authors would like to thank the entire SpiderMonkey team.

References

[1] 1996. *SpiderMonkey JavaScript Engine.* Retrieved 2023-04-26 from https://spidermonkey.dev/

[2] 2008. *V8 JavaScript Engine.* Retrieved 2023-04-26 from https://v8.dev/

[3] 2020. Speculation in JavaScriptCore. Retrieved 2023-04-26 from https://webkit.org/blog/10308/speculation-in-javascriptcore/

[4] 2023. *AreWeSlimYet.* Retrieved 2023-04-26 from https://firefox-source-docs.mozilla.org/performance/memory/awsy.html

[5] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 496–507. https://doi.org/10.1145/2594291.2594332

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Lancaster, United Kingdom) *(ICOOOLPS '11)*. Association for Computing Machinery, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2069172.2069181

[7] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1997. Value Numbering. *Software: Practice and Experience* 27, 6 (1997), 701–724. https://doi.org/10.1002/(SICI)1097-024X(199706)27:6<701::AID-SPE104>3.0.CO;2-0

[8] Stefan Brunthaler. 2010. Inline Caching Meets Quickening. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 429–451.

[9] David Callahan, Steve Carr, and Ken Kennedy. 1990. Improving Register Allocation for Subscripted Variables. *SIGPLAN Not.* 25, 6 (jun 1990), 53–65. https://doi.org/10.1145/93548.93553

[10] 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* (New Orleans, Louisiana, USA) *(OOPSLA '89)*. Association for Computing Machinery, New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

[11] Jiho Choi, Thomas Shull, and Josep Torrellas. 2019. Reusable Inline Caching for JavaScript Performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 889–901. https://doi.org/10.1145/3314221.3314587

[12] 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* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[13] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. *SIGPLAN Not.* 44, 6 (jun 2009), 465–478. https://doi.org/10.1145/1543135.1542528

[14] Neal Glew, Spyridon Triantafyllis, Michal Cierniak, Marsha Eng, Brian Lewis, and James Stichnoth. 2004. LIL: An Architecture-Neutral Language for Virtual-Machine Stubs. 111–125.

[15] Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. *SIGPLAN Not.* 47, 6 (jun 2012), 239–250. https://doi.org/10.1145/2345156.2254094

[16] 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, Berlin, Heidelberg.

[17] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *SIGPLAN Not.* 29, 6 (jun 1994), 326–336. https://doi.org/10.1145/773473.178478

[18] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] Michael Smith, Abhishek Sharma, John Renner, David Thien, Sorin Lerner, Fraser Brown, Hovav Shacham, and Deian Stefan. 2023. Cachet: A Domain-Specific Language for Trustworthy Just-In-Time Compilers. In *Workshop on Principles of Secure Compilation.*

[20] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. one vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

Received 2023-06-29; accepted 2023-07-31