MuscalietJS: Rethinking Layered Dynamic Web Runtimes

Behnam Robatmili[†] Călin Caşcaval[†] Mehrdad Reshadi[‡1] Madhukar N. Kedlaya[||]
Seth Fowler[§1] Vrajesh Bhavsar[†] Michael Weber[†] Ben Hardekopf[||]

[†]Qualcomm Research Silicon Valley [||]University of California, Santa Barbara [‡]InstartLogic [§]Mozilla

mcjs.devel@qti.qualcomm.com

Abstract

Layered JavaScript engines, in which the JavaScript runtime is built on top another managed runtime, provide better extensibility and portability compared to traditional monolithic engines. In this paper, we revisit the design of layered JavaScript engines and propose a layered architecture, called MuscalietJS[2], that splits the responsibilities of a JavaScript engine between a high-level, JavaScript-specific component and a low-level, language-agnostic .NET VM. To make up for the performance loss due to layering, we propose a two pronged approach: high-level JavaScript optimizations and exploitation of low-level VM features that produce very efficient code for hot functions. We demonstrate the validity of the MuscalietJS design through a comprehensive evaluation using both the Sunspider benchmarks and a set of web workloads. We demonstrate that our approach outperforms other layered engines such as IronJS and Rhino engines while providing extensibility, adaptability and portability.

1. Introduction

JavaScript is a language in which almost everything is dynamically modifiable: it is dynamically typed, object properties (the JavaScript name for object members) can be dynamically inserted and deleted, inheritance hierarchy (via prototype chains) can be changed dynamically, new code can be injected dynamically, and so on. These properties make JavaScript application development flexible. However, they also make compiling JavaScript a significant challenge. In addition, JavaScript is still a relatively new and evolving language, requiring an extensible compilation engine to enable easy exploration of new language features. Adding language-level compiler optimizations should be easy as well, especially in an engine architecture that supports different compilers and optimization levels. JavaScript engines are used in a variety of environments, from web page loading to web applications and server processing [14]. As such, an engine needs to adapt the quality of generated code and the time spent compiling to the particular workload. Finally, portability is critical, especially in the browser case, as JavaScript engines run on a wide variety of platforms.

Given the significant advancements in virtual machines such as .NET and JVM, both in terms of improved efficiency and large number of available features, a fundamental question is whether we can exploit these VMs to efficiently build and run a dynamic language like JavaScript in an adaptable manner. Such an approach can potentially ease new language feature extensions and improve portability. This sounds like a ideal solution for the significant need in JavaScript (and other dynamic languages) for extensibility and portability. However, concentrating almost exclusively on performance, traditional engines [9, 18] implement the entire dynamic managed runtime using an unmanaged language such as C or C++. In these engines, the basic constructs of the dynamic runtime such as object layout, heap, garbage collector, compilers, profilers and call frames, have been designed and implemented from scratch in the low-level language. This approach can achieve high performance but requires significant effort. Since everything is designed together, the monolithic architecture ends up less extensible, making it more challenging to implement an evolving dynamic language.

Recently, there have been several efforts [1, 2, 10, 16] for building a high-level dynamic VM runtime (such as JavaScript) on top of another low-level VM (such as JVM or .NET). We call such a runtime a *layered* runtime. Although a layered runtime can experience a performance hit due to the presence of the low-level VM, it can significantly ease development and improve extensibility. In this paper, we reexamine and reenvision the architecture of layered JavaScript engines with the goal of providing a flexible and performant framework for optimization. The specific contributions of this paper are:

- A new, layered JavaScript engine, MuscalietJS (or MCJS), for decent performance, rapid prototyping and exploration

[1]Work carried out while at Qualcomm Research Silicon Valley.

[2]MuscalietJS can be found at http://www.github.com/mcjs/mcjs.git

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2764-0 /14/03.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

of research ideas implemented on top of the .NET common language runtime (CLR). This architecture is different than any previously proposed layered engine such as SPUR [1], IronJS [10] or Rhino [16] in how it divides responsibilities between high-level and low-level engines and achieves better performance (Section 3). JavaScript-specific optimizations, such as property lookup and type inference, are performed in the high-level engine (Section 4). Code generation and hardware-specific optimizations are performed at the low-level VM. MuscalietJS exploits features provided by the low-level VM runtime, such as *Reflection* and *Runtime Method Attributes*, to emit efficient code for hot functions (Section 5).

- Insights into the low-level VM features that will maximize the performance benefits of the JavaScript runtime without losing generality. Examples include eliminating array bounds checks on data structures that are either used by the JavaScript JIT or generated by it, and which are guaranteed not to overflow, explicit object initialization, and avoiding some JITed code validation checks (Section 5.3).
- We demonstrate the validity of our approach by comparing the performance of MuscalietJS to previously proposed layered engines implemented in managed languages, including Rhino [16] and IronJS [10] and also monolithic engines such as V8 [9] (Section 6). We find that on both traditional JavaScript benchmarks and several record-and-replay benchmarks based on real websites, MuscalietJS significantly outperforms other layered engines such as Rhino and IronJS (by a factor of 2 to 3), while still supporting extensibility and portability.

2. Design Space of JavaScript Engines

Given the pace of change experienced in web development and the growing ubiquity of JavaScript, we identify the following criteria for evaluating JavaScript engines:

- **Performance.** Performance is the key criterion by which production JavaScript engines are evaluated. Given the increase in dynamic execution of web pages [15] and the trend toward web apps, better JavaScript performance in browsers is increasingly important.
- **Adaptability.** A JavaScript engine must be adaptable enough to recognize different workloads, ranging from server-side [14] to latency-sensitive page-load and iterative hot-spot workloads, and react accordingly to provide the best possible quality of service.
- **Extensibility.** The JavaScript language is constantly evolving both formally, (e.g., the new ECMAScript standard [7]), and informally [19–21]. This means that JavaScript engines must evolve constantly and must be designed such that extensions can be added and integrated easily, while taking advantage of the full feature set of optimizations. Changes must be localized and modular, and avoid re-engineering the engine flow.
- **Portability.** Browsers, and hence JavaScript, are widely used on many platforms, from desktops to mobile devices; thus an engine should be easily portable to different hardware architectures. This implies that engines should be decoupled from the underlying hardware, treating the hardware interface as a separate concern. However, it is still critical for performance that an engine take advantage of any available hardware-specific features.

2.1 Traditional Monolithic Architectures

The traditional engines are designed to achieve the best performance and adapt to different Web workloads. For example, the Google V8 JavaScript engine [9] is a native C++ application that has two just-in-time (JIT) compilers and no interpreter. One is a quick, simple compiler that generates very fast generic code using inline caches, and the second is a profile-based optimizing JIT compiler called Crankshaft. The optimizing compiler applies a wide range of low-level optimizations such as SSA (Static Single Assignment) redundancy elimination, register allocation, and static type inference when generating native code. It also uses type feedback from the inline caches from the simple compiler. V8 does not use a bytecode-based IR (intermediate representation); instead, it uses both high-level and low-level graph-based IRs for different levels of optimization. SpiderMonkey [18], Firefox's JavaScript engine, is another monolithic engine written in C/C++ and comprised of a bytecode-based interpreter, a baseline JIT compiler, and an optimizing JIT compiler (IonMonkey). The baseline compiler is similar to the V8 engine's simple compiler and uses inline caches to generate fast code.

While these runtimes are usually implemented and compiled in a low-level unmanaged language such as C++, they need to provide complete JIT compilers with several IRs and optimizations at different levels and the assembly code generators for all targeted machines. The compiler stack and multiple assembly code generators must be updated frequently to support newly added language features. Significant development effort is required to extend and maintain such engines and port them to new platforms. Researchers have been experimenting with various JIT compilers in the Mozilla framework, such as the tracing JIT in Gal et al. [8], and highlight the complexity of implementing such components in monolithic engines. Additionally, in these architectures, there is an impedance mismatch between the runtime call stack (generally C++) and the call stack of the executing generated code in the target language (JavaScript). Consequently, communication between the executing JITed code and the runtime is complex and has high overhead.

2.2 Layered Architectures

In layered JavaScript (also called Repurposed JIT) architectures, the target dynamic runtime is built on top of another runtime engine, rather than building an entire compiler from scratch. These designs are typically more portable

Table 1. Traditional vs. Layered JavaScript Engines

Architecture	Name	Host Language	Compilation methodology	LOC
Monolithic	V8	C++	Adaptive: Inline-cache based fast compiler wit no IR for cold code; an JavaScript optimizing compiler (HIR and LIR) for hot code	1M
Monolithic	SpiderMonkey	C++	Adaptive: A byte-code based interpreter for cold code; a trace JIT and method JIT for hot JavaScript code	300K
Layered	MuscalietJS	CLR (C#)	Adaptive: an IR-based interpreter implemented in CIL for cold code; an JavaScript optimizing compiler (implemented in CIL) and a CIL code generator for hot code	95K
Layered	SPUR	CLR (C#)	Translate JavaScript to CIL; leaves dynamic optimizations to tracing CIL JIT	N/A
Layered	IronJS	DLR (F#)	Translate JavaScript to DLR; leaves dynamic optimizations to DLR	23K
Layered	Rhino	JVM (Java)	Non-adaptive but can be reconfigured ahead of time (AOT); supports various degrees of optimizations (interpreter to optimized code)	115K

than monolithic ones, as hardware-specific optimizations and functionality are provided by the low-level VM. Additionally, the target runtime may take advantage of features and resources provided by the host runtime including object layout, garbage collection and code generators. Therefore, these designs can better fit the quick development cycles needed for evolving dynamic languages. For example, as shown in Table 1, the code size for these architecture is significantly smaller (up to 10x) than the code size of traditional engines (of course it can be argued that LOC may not be the best metric indicating complexity). Also, adding new features to the layered engines is usually easier. For example adding support for parallel execution or parallel compilation to monolithic engines is a significant task as it requires major changes to the memory system, garbage collector and runtime. Whereas, basic parallelism support in a layered engine that runs on top of .NET for example, such as MuscalietJS, can be provided using Task-Parallel Library (TPL) already available in the .NET runtime.

Table 1 compares MuscalietJS against three common layered JavaScript engines and two traditional engines. MuscalietJS is a layered JavaScript engine architecture built on top of the .NET Common Language Runtime (CLR). Most other layered engine provide a thin layer on the top and rely on the low-level engine to provide the optimizations. MuscalietJS, however, implements most of language-level optimizations at the high-level, where the semantics of the language allows for better decisions. And it delegates the traditional compiler optimizations to the low-level engine (like register allocation). MuscalietJS performs adaptive JavaScript-specific optimizations and uses CIL code generation to JIT optimized versions of hot function after applying high-level JavaScript optimizations. These high-level optimizations include hidden classes, property lookup, type analysis, and restricted dataflow analysis. MuscalietJS also exploits special features in CLR runtimes to generate high-performance optimized code. For example, MuscalietJS uses reflection and inlining hints to generate optimized code for dynamic JavaScript operations. MuscalietJS communicates special hints to the CLR engine to avoid array-bounds checks and object initialization for JavaScript property access and JavaScript object creation.

SPUR [1] is a tracing JIT compiler for Common Intermediate Language (CIL) in which JavaScript is directly compiled to CIL and CIL is trace-compiled by SPUR for better optimization. The authors show that tracing CIL generated by compiling JavaScript programs gives similar performance gains to a JavaScript tracing compiler. Due to the modular nature of our engine, we believe that it is possible for MuscalietJS to run on top of any CIL compiler including SPUR. Given the effectiveness of SPUR over Microsoft's .NET runtime, MuscalietJS might benefit from it; unfortunately, SPUR is not available for general use and so we could not test this. IronJS [10] translates JavaScript to Dynamic Language Runtime (DLR) expression trees [5] and leaves dynamic optimizations to the DLR. DLR uses the concept of *dynamic callsites* to generate type specialized versions of each operation. This is similar to polymorphic inline caches. Rhino [16] is a JavaScript engine on top of the JVM which offers various levels of optimization. The default configuration of the runtime invokes a naïve bytecode-based interpreter written in Java. Other optimization levels invoke a code generator that generates Java class files. When set to the highest optimization level, Rhino performs optimizations such as detection of numerical operations, common sub-expression elimination, and function call target caching. Though the current implementation lacks adaptive compilation, optimizations can be enabled or disabled ahead of time. Nashorn [13] is a more recent implementation of a JavaScript runtime on the JVM. Nashorn uses the *invoke-dynamic* bytecode instruction, which was added to recent versions of the JVM to enable efficient implementation of dynamic language runtimes. We were not able to find a performant version of Nashorn to compare against.

Some prior studies [2] have stressed the importance of having matching semantics of the two layers. We believe semantics of the low-level VM might be of some importance but the overall architecture and work breakdown between high-level and low-level VMs are more important. For example, by moving JavaScript optimizations (similar to the one explained in Section 4) to the high-level engine, it is possible to achieve significant speedups regardless of the semantics of the lower-level language. As our results show while semantics of DLR used by IronJS are relatively close to dynamic languages such as JavaScript, with better engineering MCJS outperforms IronJS.

3. MuscalietJS Architectural Overview

MuscalietJS is a layered architecture, shown in Figure 1. It splits responsibilities across two levels: a JavaScript-specific engine and a language-agnostic low-level VM. In principle, any managed language VM can serve as the low-level engine. Our current implementation uses the Common Language Runtime (CLR), as implemented by Mono [12]. The low-level VM provides traditional compiler optimizations: instruction scheduling, register allocation, constant propagation, common subexpression elimination, etc., as well as code generation and machine specific optimizations. In addition it provides managed language services such as garbage collection, allowing us to focus on the JavaScript-specific aspects of the engine.

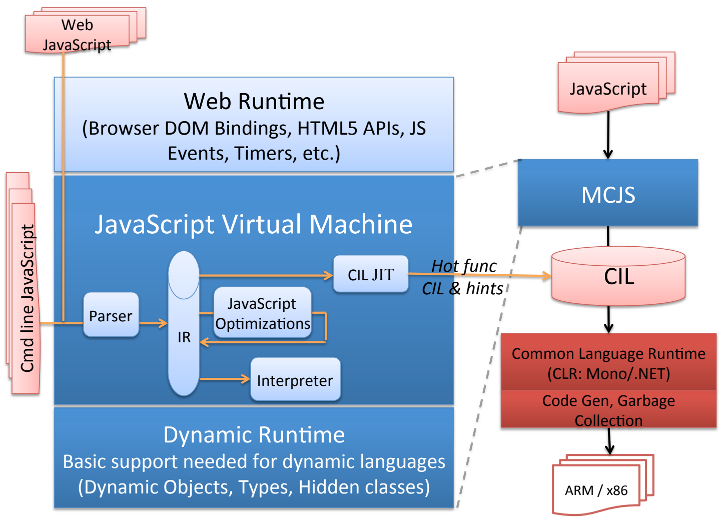
Web JavaScript
Web Runtime
(Browser DOM Bindings, HTML5 APIs, JS Events, Timers, etc.)
JavaScript Virtual Machine
Cmd line JavaScript
Parser
IR
CIL JIT
JavaScript Optimizations
Interpreter
Hot func
CIL & hints
Dynamic Runtime
Basic support needed for dynamic languages
(Dynamic Objects, Types, Hidden classes)
JavaScript
MCJS
CIL
Common Language Runtime
(CLR: Mono/.NET)
Code Gen, Garbage Collection
ARM / x86

Figure 1. MuscalietJS Architecture

The JavaScript specific layer is further decomposed into several components:

- **JavaScript runtime.** The high-level runtime consists of a parser, an interpreter, a parallel JIT compiler, and a profiler. The parser takes in JavaScript code and produces a custom IR (see Sec. 4.1). The interpreter executes the IR directly, while the JIT compiler applies JavaScript-specific transformations and optimizations, and generates CIL bytecodes for hot functions. Some examples of performed optimizations are: type analysis and type inference, array analysis, and signature-based specialization; these are further discussed in Section 4.

  The combination of interpreter, JIT, and profiler provides adaptability in our design by deciding which compilation path is more appropriate given the workload. For latency-sensitive scenarios, like browser page load, we provide an interpreter which can directly execute the output of the parser without any intermediate bytecode generation. As functions are invoked multiple times and become hot, the JIT compiler will optimize them since the compilation time can be amortized. For the hottest, most performance-sensitive code, we apply more expensive optimizations at both the high-level JavaScript engine layer and the low-level VM layer.

- **Dynamic runtime.** The dynamic runtime provides the necessary support to enable compilation for dynamic, prototype-based languages. This includes dynamic values, objects, types, and hidden classes.

- **Web runtime.** The web runtime handles the integration with the browser. MuscalietJS was designed in concert with a browser architecture to optimize the bindings between the browser and the engine. The web runtime understands the semantics of the DOM and implements DOM bindings as well as other browser-related services like events and timers.

Running the JavaScript engine inside another VM has performance implications. Our split design relies on JavaScript specific optimizations at the high-level to help mitigate the overhead of running on the CLR. The JavaScript engine code generator exploits advanced high-level techniques combined with type analysis and special hints to lead the low-level VM to generate high-quality optimized code. There are performance advantages to running on top of the CLR.

4. JavaScript Specific Optimizations

This section discusses JavaScript-specific optimizations applied to the input JavaScript code in the JavaScript runtime (JSR) and dynamic runtime (DR) components.

4.1 Parsing and IR Generation

For JavaScript function, the MuscalietJS engine uses a graph-based intermediate representation (IR) that describes JavaScript code, using simple operations that are easy to analyze – for example, all types of loops at the JavaScript level are represented using a single construct at the IR level. Our IR describes the flow of data through operations by placing edges in the graph between expressions that generate values (called *Writers* in our IR) and expressions that use those values (*Users*). It also represents implicit operations like type conversions explicitly so that they can be taken into account during later phases. This simplifies the implementation of analyses like type inference. The interpreter, optimization passes, and code generator all operate using this IR.

We depart from common practice in our approach when constructing the IR. Rather than building a temporary abstract syntax tree (AST) which is then used to build the IR in a separate pass, in our design the parser generates the IR directly. Because of the structure of the IR, constructing it has the effect of performing some of our analyses up front. Dataflow analysis, for example, is performed completely at parse time by connecting Writers and Users as they are constructed. We also build a symbol table for each function and determine useful metadata like whether the function uses *eval* or closes over its environment. This has significant advantages in the context of a latency-sensitive system like a web browser: it reduces the number of passes that are necessary, improves locality (since multiple passes are fused), and eliminates the overhead of constructing temporary AST nodes that will later be discarded. The disadvantage of this approach is the complexity involved in implementing the semantic actions of the parser. This problem is approached by separating the parser's view of the IR from that of the *IR factory* which constructs the IR nodes. From the parser's perspective, we maintain the illusion, whenever possible, that we are building an AST that follows the structure of the ECMAScript 5 [7] grammar very closely. This is done by tagging the real IR node classes with empty interfaces corresponding to nonterminals in the grammar. These interfaces have no effect at runtime, but they provide static type-level constraints that make it easier to ensure the parser behaves correctly when constructing the IR graph. They also hide the complexity of the underlying graph nodes by presenting the parser with a simple, uniform interface. The IR factory, meanwhile, works with the concrete IR node types, and does not concern itself with the structure of the grammar at all.

Since there is such an impedance mismatch between the grammar and the IR, we cannot always hide the truth from the parser perfectly. However, by obeying two requirements when designing the IR and the factory code that constructs it, we were able to isolate these issues to a few small parts in the parser. One requirement was that the IR could be constructed in a recursive fashion, matching the parser's algorithmic structure. This was fundamentally needed to allow the construction of the IR without using a separate pass. We also needed to allow the parser to backtrack in certain circumstances. To support this, we required that the IR factory act only on local state stored within the IR nodes themselves, so that IR subgraphs could be thrown away without corrupting global data structures. The combination of these constraints also has another advantage: the designs of the parser and the IR factory lend themselves nicely to parallelism. We leave further exploration of that approach for future work. Once the IR for a parsed function is created, it can be used by the interpreter to run the function or by the CIL JIT engine to apply type optimizations and eventually generate optimized code.

4.2 Adaptive Function-level Execution

To achieve adaptability and performance, traditional JavaScript engines support different modes of execution (interpretation, basic JIT and advanced JIT with specialization) at function granularity. For example, V8 first quickly generates an unoptimized version of the code for each function. After a certain number of executions a trampoline code section replaces the code pointer with a runtime function that generates a heavily optimized version of the code. The code pointer is then changed to point to the optimized code. It remains in this state unless the assumptions made during optimization prove false, in which case a *deoptimization* happens which restores the unoptimized code.

Providing this type of dynamic code adjustment (especially across the runtime function and JITed code) requires access to stack frames, control over stack semantics, and sometimes support for self-modifying code. This is hard to achieve in a layered design given the restrictions imposed by host VMs like the CLR. We therefore take a different approach to achieve the same degree of adaptability. Each JavaScript function object in MuscalietJS has a *codePtr*: a function pointer (C# delegate) that takes as its only argument a *CallFrame* object. The CallFrame includes a reference to the function object for that function as well as the actual input arguments and their types. A codePtr can point to different runtime functions that manage the execution and optimization of the function, or to different JITed specializations of the function. Each JavaScript function object also stores *function metadata*, which includes the current compilation state of the function (parse, analyze, JIT, specialized JIT), the IR graph of the function, and a code cache to store the function's JITed CIL code. The *codePtr* of a JavaScript function can point to one of the following functions:

- **FirstExecute (runtime function):** When the JavaScript function is called by the JavaScript code for the first time, this function performs pre-JIT analysis and the initialization of the JIT code cache for that function. At the end, the codePtr is changed to point to another runtime function responsible for normal execution.
- **NormalExecute (runtime):** Depending on the status of the JavaScript function, the normal execution function either calls the interpreter (for "cold" functions) or JITs the function at one of several optimization settings.
- **Interpret:** The interpreter executes the function by traversing its IR graph and calling runtime functions that implement operations, property lookup, and other JavaScript semantics.
- **CIL JITed code:** For hot functions, normal execution starts the JIT to generate a specialized version of the function for the current arguments' types. The generated code will be added to the code cache and *codePtr* will be updated to point to it.

Tight coupling between the target and host runtime allows low overhead switching between interpreter and JIT code similar to traditional monolithic runtimes.

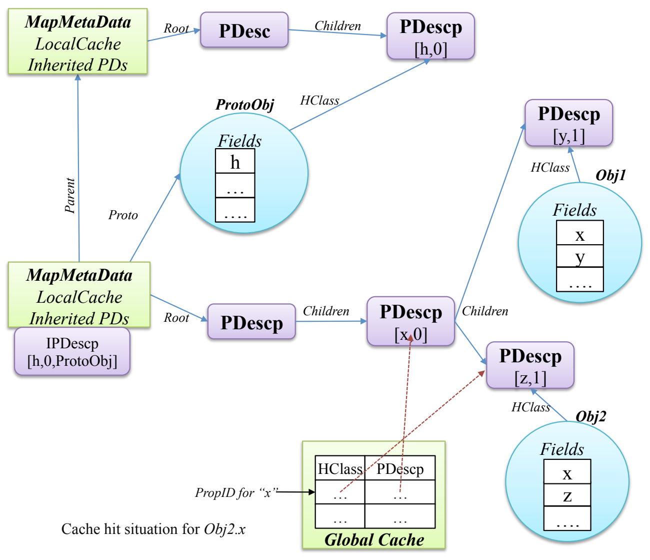

Figure 2. Property description data structure for representing hidden classes (maps) in MuscalietJS DR.

4.3 Type Analysis

Advanced monolithic engines employ type analysis to JIT efficient code for hot functions. MuscalietJS also implements an advanced type analysis by combining type feedback and type inference before JITing optimized code for hot functions. An initial type inference pass is first applied to reduce type feedback overhead by enabling more intelligent placement of profiling hooks. After profiling, when a hot function is detected in the next invocation of that function, a second type inference pass uses the profiling type information to infer the types of most variables or expressions in the function.

MuscalietJS supports type signature-based method dispatch for JavaScript. The basic concept is to dynamically generate different JITed implementations of the same function, each specialized for one observed signature [4], i.e. the types of the function's arguments for a particular invocation. At each invocation of the method the current signature is used to dispatch to an appropriate implementation. The type analysis and signature-based method dispatch algorithm used by MuscalietJS is similar to the type inference and analysis explained in [11].

4.4 Property Lookup

As in any dynamic runtime, the way objects and their fields are stored in memory affects the performance and efficiency of the memory allocation and object and field accesses [3]. In monolithic engines such as V8 [9], object layouts are complex and tightly coupled to the garbage collector, imposing a huge burden on extending the engine with new object types and features. For JavaScript, engines also have an internal data structure representing the implicit class of each object at a given point during execution, known as a *hidden class* or *map*. The hidden class of an object can change at any point given the dynamic nature of the language. This affects the way optimizing compilers generate efficient JITed code, and the JIT must take this layout into account in order to provide efficient property lookups using low-level optimizations such as inline caches. In MuscalietJS, both JavaScript objects and their hidden classes are normal CIL objects. Adding new features is easy, since it requires no translation between layers. This extensibility does not come at the cost of speed; the MuscalietJS representation is capable of fast property lookups using techniques like multi-level caching and property propagation. The same lookup mechanism is used directly by the interpreter and the JIT engine in the JSR.

MuscalietJS employs a data structure for representing hidden classes that is customized for prototype-based languages. In such languages, objects act as multi-level dictionary, where each level adds or replaces properties and inherits the rest from another object, its *prototype*. Since an object may both be a prototype and have a prototype of its own, a *prototype chain* is formed. The prototype chain is highly dynamic in nature because any of the objects forming the chain may be changed at any time. To address the challenge of inferring hidden classes from these chains of objects at runtime, we use a data structure with a tree of *property descriptors* at each level of inheritance or *prototype depth*. Each node in the tree represents an *owned* property – that is, a property which is set directly on the object rather than being inherited. Each property descriptor includes the property name and the offset at which the corresponding field is stored in the objects that share this prototype. Every path from a node to the root of the tree represents the fields of an object in the order they were added. Each node thus corresponds to a particular hidden class, shared by all objects whose fields were constructed in the same manner. This data structure makes it easy to keep the inferred hidden class of an object up to date as properties are added at runtime. The root of the tree has no property info. Therefore, the root node alone represents any object with no properties of its own. Note that such an object may still inherit properties from its prototype chain.

All of the property descriptors in a tree share a *map metadata* structure that includes a reference to the prototype object, a list of properties inherited through the prototype, and a cache for recently accessed property descriptors. Figure 2 shows an example of the two-level map data structure used by MuscalietJS for managing dynamic objects and properties. In this particular example, the prototype has one field called *"h"*. Objects *Obj1* and *Obj2* inherit *"h"* from that prototype and add { *"x", "y"* } and {*"x", "z"*}, respectively as sets of owned fields stored directly on each object.

Slow Path Lookup: The default process for looking up a property name in an object (e.g. *obj1.x* in Figure 2) includes walking the tree of property descriptors, starting at the node corresponding to the hidden class (map) of the object and continuing towards the root of the tree. If a property descriptor with the desired property name is found, the value is retrieved from the field at the corresponding offset in the object. Otherwise, the search is continued on the next object in the prototype chain. For instance, looking up *obj1.h* in Figure 2 will cause the property descriptors to be walked starting from *Obj1* until the root property descriptor is reached. Since the property won't be found, this will be followed recursively by a walk of *ProtoObj*'s property descriptors (since *ProtoObj* is the prototype of *Obj1*), eventually reaching the property descriptor for *"h"*. To reduce the cost of future lookups, when a lookup on an object results in an inherited property being found, an *inherited property descriptor* is added to the map metadata structure at the object's prototype depth. The inherited property descriptor indicates the prototype object that owns the property and the offset of the corresponding field in the prototype object.

Fast Path Lookup: MuscalietJS employs two levels of caching to speed up the property lookup process: a global cache shared between all hidden classes and objects and a cache at the map metadata level. For faster lookups in these caches, the MuscalietJS runtime globally assigns an integer ID (called a *propID*) to each statically known string property name and manages the propIDs during code generation and use these IDs to access these caches for known property names. The global cache is a small array, indexed by the low-order bits of the propID, that contains tuples composed of a hidden class (identified by a node in the tree of property descriptors) and the last property descriptor retrieved for that propID. If the looked up hidden class matches the hidden class of the object on which we are performing the lookup, then the property descriptor is still correct and can be used to access the corresponding value. A global cache hit for *Obj2.x* (or any other object with the same hidden class accessing property *"x"*) is shown in Figure 2. If there is no hit in the global cache, we check a second-level cache associated with the map metadata of that object (for example the cache in *Obj1.HClass.MapMetadata* for Obj1 in the example). This map metadata local cache similarly maps a property name to the last property descriptor matched for that name and the hidden class the match was for. While slower, this cache achieves better locality than the global cache given that it is limited to the hidden classes associated with a particular map metadata.

If the propID cannot be obtained for a property statically, the generated code bypasses the cache and uses the runtime property name as a string through the slow lookup path. Using these compile time assigned propIDs, the caching overhead is significantly reduced. Some engines, such as V8 [9], use transient maps for representing hidden classes and use inline caches for fast property lookup and requires deoptimization in case of inline cache miss. Instead MuscalietJS hidden class model requires a tree walk for cache misses and is amenable to property caching at the runtime level. Also it uses two levels of caching and the results in Section 6 report good hit rates for different types of workloads using this mechanism.

5. CIL Code Generation for Hot Functions

In JavaScript, most operations are dynamically defined depending on the types of their operands. For example, a property load operation such as *object.a* can turn into a direct field access in *object*, a field access in *object*'s prototype chain, a call to a getter function call, or a call back to a browser function. A binary addition can translate to integer or double addition, a string concatenation or a complicated user-defined conversion on operands followed by a numerical addition or a concatenation, depending on the value types of the operands.

Given the heavy use of the dynamic operations in JavaScript (and most other dynamic languages), generating efficient code based on the inferred or profiled types is very important when JITing code for hot functions. With the large number of variants for each of these operations, producing low-level code for all of the variants, as is done in traditional engines, is challenging and hard to debug. MuscalietJS exploits the rich features of the host engine such as reflection and special code-generation hints to JIT efficient code while maintaining scalability and debuggability. This section discusses how MuscalietJS performs code generation for operations in hot functions using these techniques. The section first explains how MuscalietJS implements operations. Then, it explains how efficient code generation is performed using these operations and lookup mechanisms and the special hints MuscalietJS passes to the host runtime (Mono or .NET) for guiding low-level code generation.

5.1 Operation Implementation

Initially, we generated the CIL bytecode implementing each operation manually; there were many variations depending on the operand type information extracted during type inference. However, this approach is very tedious and hard to maintain because the relationships between the input and output types of the operations and their corresponding implementations were separated and implicitly captured in multiple places, including the type inference and JIT algorithms. Debugging was also very difficult.

To address these problems, we created an operation database. This is essentially a series of overloaded functions, each implementing one particular instance of an operation's behavior. During type inference, we use reflection to map operand types (function argument types) to operation result types (function return value types). During JIT we again use reflection to look up the appropriate operation implementation function based on the types of the operand and simply generate a call to that function. Figure 3 (B) shows a simplified code for binary addition operator for three of the possible many cases including *(Int, Int)*, *(Undefined, Bool)* and *(Bool, Float)* in the MuscalietJS operation code database.

```
/**** (A) CodeGen for binary operations ***********/
Visit(BinaryExp node, OpCache operation)
{
    var ltype = Visit(node.Left);
    var rypet = Visit(node.Right);
    /* Using reflection to extract the expected type and call*/
    var methodInfo = operation.Get(ltype, rtype);
    var resultType = operation.ReturnType(ltype, rtype);
    GenerateCall(methodInfo, resultType);
}

/**** (B) Addition operation ***********/
public static class Add
{
    ...
    [MethodImplAttribute(MethodImplOptions.AggressiveInlining)]
    public static float Run(bool i0, float i1)
    { return Run((float)Convert.ToNumber.Run(i0), i1); }
    ...
    [MethodImplAttribute(MethodImplOptions.AggressiveInlining)]
    public static double Run(mdr.DUndefined i0, bool i1)
    { return double.NaN; }
    ...
    [MethodImplAttribute(MethodImplOptions.AggressiveInlining)]
    public static int Run(int i0, int i1) { return i0 + i1; }
    ...
}
```

Figure 3. CodeGeneration (JIT) code for binary operation IR nodes (A) and *Addition* operation implementation (B).

This approach produces very robust, maintainable, and easy to debug code. Since we use reflection to take advantage of the overloading functionality of the host language, we do not even need to generate lookup code. By assigning proper CIL method attributes to these operator implementation functions, the runtime is forced to inline them, and the compiled code achieves the same performance as the previous manually-generated CIL bytecode implementation. We developed a template-based code generator to generate most of the C# code for the various possible implementations of each operation, and used the "partial classes" feature of the C# language to manually implement the corner cases and integrate them with the rest of the auto-generated code.

5.2 CIL Code Generation

The MuscalietJS code generator (JIT engine) traverses the IR of the function being JITed and uses information added to the IR during type inference or other pre-JIT phases to generate efficient CIL code. This section discusses some of the techniques used by the engine and how it interacts with the low-level VM to achieve high performance.

The engine generates CIL for JavaScript code using the CLR's reflection API. The various JavaScript-level expressions are implemented by generating calls to appropriate operation functions specialized for the inferred types of their operands as shown in Figure 3. To show how our code generator uses operand types and hints to the low-level VM to help the underlying platform generate efficient code, Figure 4 illustrates: (A) a sample JavaScript function, (B) its corresponding CIL code generated by MuscalietJS and, (C)

```
/********* (A) JavaScript Source ***********/
function doublePlusOne(a)
{
    return 2 * a + 1;
}
doublePlusOne(20)
............
/*** (B) Generated CIL code for doublePlusOne ***/
............
0.0 ldarg.0 ;loading call frame
1.1 ldflda ;loading arg0 in callframe (a)
2.1 call Int32 DValue:AsInt32 () ;unboxing arg0
2.2 stloc.1
3.0 ldarg.0
4.0 ldflda
5.1 ldc.i4 2
5.2 ldloc.1
5.3 call Int32 Binary.Mul:Run (Int32, Int32)
6.1 ldc.i4 1
6.2 call Int32 Binary.Add:Run (Int32, Int32)
7.0 call Void DValue:Set (Int32) ;boxing for return
............
/*** (C) x86 assembly code generated by mono ***/
............
addl $0x10,%esp
leal 0x34(%edi),%eax ;load arg0 (a)
movl 0x04(%eax),%ecx
movl %ecx,0xf4(%ebp)
leal 0x0c(%edi),%eax
shll %ecx ;inlined/optimized integer multiply
incl %ecx ;inlined/optimized integer addition
movl %ecx,0x04(%eax) ; boxing for return
movl $0x00000009,(%eax)
leal 0xfc(%ebp),%esp
```

Figure 4. JITed CIL and assembly for a sample code.

the x86 code generated by the low-level engine. The call to the `doublePlusOne` function shown in the figure passes an *Int* value (a) as argument, and the literals in the function are all integer constants. Therefore, the type inference pass infers the type of the operands to both the multiplication and addition operations to be *Int*. During JIT compilation, as shown in Figure 3, the JIT engine uses reflection to look up multiplication and addition operation functions in the runtime appropriate for *(Int, Int)* operands. The resulting functions, which are called in the JITed code (CIL lines 5.3 and 6.2), are the basic integer addition and multiply functions `Binary.Mul:Run` and `Binary.Add:Run`, which are implemented in terms of CIL-level primitive operations. Since these functions return *Int* as well, the intermediate value passing the multiplication result to the addition operation is inferred to be an *Int* as well, and so it does not generate any boxing or unboxing operations. All the operations in the function are pure integer operations except the unboxing of the function argument read from the call frame (CIL line 2.1) and the boxing for the return (CIL line 7.0).

When the low-level VM sees this specialized version of `doublePlusOne`, it is guaranteed to inline the calls to `Binary.Mul:Run`, `Binary.Add:Run`, and the runtime functions that handle boxing, because we annotate them with CIL attributes that require this behavior (*AggressiveInlining*). After inlining, the calls are replaced with simple integer + and `*` operations. The low-level VM then applies other optimizations such as dead code elimination, register allocation, and constant folding on the resulting code. Finally, optimized code is generated for the target hardware architecture. For example, as shown in Figure 4, the x86 generated code implements the multiplication and addition operations in the original JavaScript function with a `shift left` instruction and an `increment` instruction.

We implement property access expressions (the `.` operator) at the JavaScript level with calls to the property lookup runtime operation discussed in Subsection 4.4. These calls are also inlined by the low-level VM. For indexing operations (the `[]` operator), MuscalietJS performs special optimizations: if the inferred type of the expression that is being indexed is *Array*, and the indexing expression's inferred type is *Int*, then we can skip the property lookup code and cache accesses. Instead, the engine generates a simple array access, using the integer to directly index the internal CIL-level array used to implement the array object.

In traditional JavaScript engines, calls to runtime functions from the JITed code is usually very costly, given that the runtime is unmanaged while the JITed code exists in a managed environment. The primary source of overhead is the difference in stack frame models between the two environments. In the MuscalietJS runtime, however, JavaScript objects, the runtime, calls to operations, boxing and unboxing, and JITed code all exist in the same managed world and so communication between them has essentially no overhead. As shown, MuscalietJS effectively exploits this low-overhead during CIL code generation for hot functions.

5.3 Special Hints

Through significant profiling and evaluation of the Mono and .NET VMs, we designed a set of hints that can be passed to the low-level VM to significantly improve the generated code in a layered architecture like MuscalietJS. Most of these hints can be provided as per-value or function attributes passed to the runtime once. We were able to implement some of these features in the Mono runtime:

- **Avoiding array bound check for property access operations:** MuscalietJS can guarantee the validity of the length of the property array in the objects through hidden classes, so the low-level code that would be generated by the low-level .NET VM for those checks is redundant.
- **Avoiding object zero initialization for JavaScript objects created by MuscalietJS:** MuscalietJS explicitly sets the fields of JavaScript objects after creating them in JITed CIL codes.
- **Avoiding IL validation security checks for CIL code emitted by MuscalietJS:** We found that the Microsoft .NET runtime [6] spends a huge amount of time on security validation of our JITed CIL code. Our profiling shows that with .NET, Sunspider benchmarks are slowed on average by 2× (and sometimes as much as 5×) because of the JIT method access check done on every invocation of MuscalietJS generated code for hot methods. This is observed in the .NET runtime, but we did not observe this issue with the Mono runtime [12]. This security check is redundant because MuscalietJS performs code validation for type safety during code generation. We were not able to fix this issue and so could not fully evaluate MuscalietJS on .NET. Having a selective IL validation check can improve the code quality of layered designs such as MuscalietJS.
- **Allowing on-stack replacement (OSR) between two CIL emitted methods:** As MuscalietJS employs type profiles as well as type inference, it sometimes generates speculative code protected by guards. As a result, it needs support for deoptimization and jumps back to the interpreter if the guard condition is violated. Switching contexts between the optimized method and the interpreter and continuing execution is a challenging task given the current limitations of the CIL stack machine. Since both Mono and .NET internally support OSR, having high-level access to that feature would have simplified the implementation of such a deoptimizer.

6. Evaluation

6.1 Evaluation Methodology

We evaluated MuscalietJS against three existing JavaScript engines: Rhino [16] and IronJS [10], which both have layered architectures (one running on top of JVM and the other one on top of .NET DLR) similar in some respects to MuscalietJS (MuscalietJS performs many more high-level JavaScript-specific type analysis and optimizations), and V8 [9], which is a state-of-the-art native engine. We were not able to evaluate against SPUR [1] as SPUR is not publicly available. We used the latest release of each engine as of this writing: Rhino v1.7, and IronJS v0.2.0.1. Our engine, MuscalietJS v0.9, and IronJS run on top of Mono 2.10. All experiments were carried out on a 2.80GHz Intel Core i7 machine with 8 GB of RAM, running Ubuntu 11.04.

Table 2. JavaScript Engines Configurations

Name	Description
MCJS_I	MuscalietJS interpreter only
MCJS_J	MuscalietJS JIT
MCJS_J+	MuscalietJS JIT + type inference/specialization and array optimization
MCJS_IJ	MuscalietJS interpreter/JIT for cold/hot functions
Rhino_I	Rhino interpreter only
Rhino_C	Rhino basic compiler (-O 0)
Rhino_C+	Rhino compiler with maximum optimizations (-O 9)
IronJS	IronJS translates to DLR expression trees, DLR performs dynamic optimizations

Our test configurations are described in Table 2. Note that Rhino does not JIT code on a per-function basis as MuscalietJS does; instead, depending on its configuration, it is either a pure interpreter or a pure AOT compiler. To

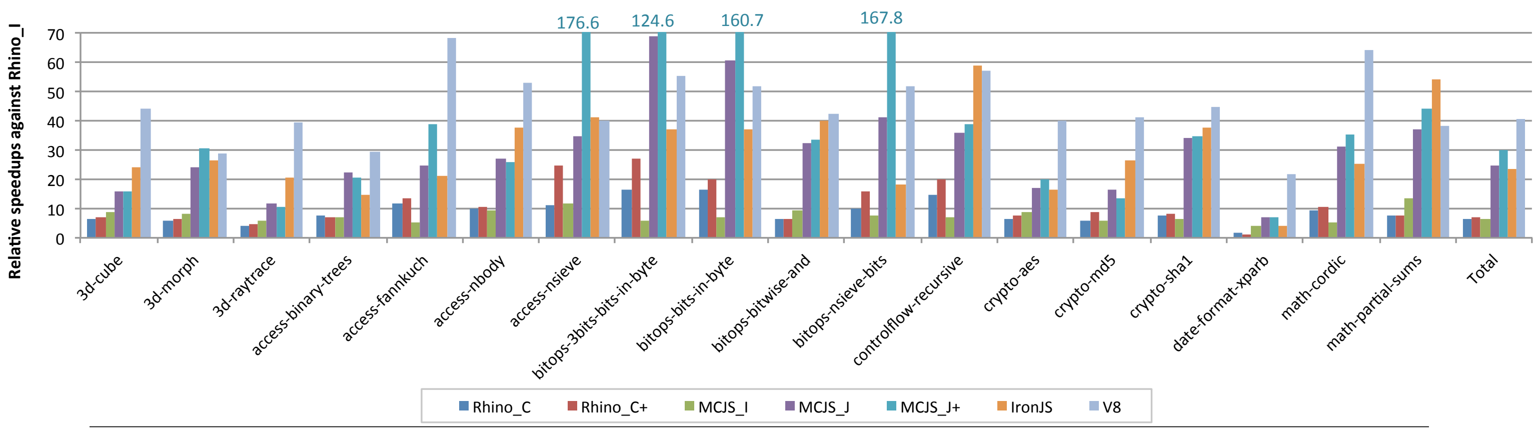

Figure 5. Relative speedups against Rhino_I for Sunspider benchmarks

reduce variation, we run each benchmark four times on each engine and average the execution times. For the results, we measured confidence intervals of 1%, 2%, 5%, and 2% for V8, Rhino, IronJS, and MuscalietJS, respectively.

Benchmark Selection: Selecting the right set of benchmarks across all platforms and configuration was challenging. Rhino and IronJS fail to execute JSBench [17] benchmarks and several Octane benchmarks require typed arrays not yet supported by MCJS. We present results for the Sunspider and V8 benchmark suites. We run each test in loop 4 times to increase the running time and improve the confidence of our results.

In addition we want to capture workloads that are characteristic of web page loading, in which large amounts of JavaScript code are downloaded, but only small portions are executed most of the time, and these portions are usually only executed once [15]. To make evaluating such workloads possible, we generate a set of benchmarks using a record-and-replay mechanism. To generate such a benchmark for a given web page, we record the process of loading the web page in the browser. Later, our benchmark generation tool can reconstruct all of the JavaScript code locally. To model the DOM-based interaction between the browser and the JavaScript engine, the tool prefixes each script with additional code that initializes a simulated DOM tree represented by a tree of JavaScript objects, each of which has the same properties that were read from their real counterparts during execution. The performance overhead introduced by the simulated DOM tree is similar to the browser-integration, as the JavaScript side of the DOM bindings also wraps native objects and accesses them in the managed JavaScript heap.

6.2 Traditional Benchmarks

Figure 5 presents average execution time for the Sunspider benchmark suite on the configurations of MuscalietJS, Rhino, and IronJS listed in Table 2. The last column (*total*) shows the total Sunspider score speedup, also relative to the Rhino interpreter. A few benchmarks crash on IronJS running with Mono; therefore we present 18 benchmarks running correctly on all platforms. The IJ and IJ+ results for MuscalietJS are very close to I and J+ so the table does not include them. On Sunspider, the MuscalietJS interpreter (I) performs close to the Rhino AOT compiler (C). The MuscalietJS JIT (J) improves performance over the MuscalietJS interpreter by a factor of 2 to 3× for these benchmarks, and delivers almost 3× the performance of the Rhino AOT compiler at its maximum optimization level (C+) despite the overhead of JITing code at runtime. This is slightly higher than the speedups achieved by IronJS using .NET DLR. This is not surprising, since we implemented hidden classes and property lookup directly in the managed runtime using its internal data structures. In addition, when we enable JavaScript-specific type optimizations, such as array optimization and type analysis (J+), MuscalietJS outperforms IronJS (1.3× on average up to 4× on some benchmarks such as access-nsieve). The speed ups over Rhino_C+ is about 4×. These significant speedups emphasize the effectiveness of our JavaScript-specific optimizations such as property lookup, array optimizations and type analysis. On average MCJS_J+ achieves about 75% of the speed of the V8 engine on Sunspider benchmarks, which is by far the highest reported performance for a layered engine.

Table 3 presents average execution time for a subset of the V8 benchmark suite that we were able to get to run on all of MuscalietJS, Rhino, and IronJS configurations. Rhino compiler performs very well achieving an average speed up of 3.8 compared to Rhino interpreter. Given that for these long-running benchmarks, the compilation time is negligible, relying on several static optimizations, Rhino AOT compiler is able achieve these speedups. Relying on JavaScript-specific optimizations, MuscalietJS optimizing CIL compiler, on the other hand, is able to outperform both IronJS and Rhino best configurations. On average, the best MuscalietJS configuration (MCJS_J+) outperforms the IronJS and Rhino best configurations by about 2× and 1.2×, respectively. The two slow benchmarks for MuscalietJS are *Splay* and *Regex*. *Splay* stresses garbage collector and the slowdown can be related to the Mono garbage collection mechanism and changing the garbage collection mechanism

```
function nsieve(m, isPrime) {
   var i, k, count;
   for (i=2; i<=m; i++) { isPrime[i] = true; }
   count = 0;
   for (i=2; i<=m; i++){
      if (isPrime[i]) {
         for (k=i+i; k<=m; k+=i) isPrime[k] = false;
         count++;
      }
   }
   return count;
}
function sieve() {
    for (var i = 1; i <= 3; i++ ) {
        var m = (1<<i)*10000;
        var flags = Array(m+1);
        nsieve(m, flags);
    }
}
sieve();
```

Figure 6. *access-nsieve* sunspider benchmark.

can potentially improve the results. *Regex* on the other hand, runs in .NET regex operations most of the time, which could also be the source of slowdown for that benchmark.

Table 3. Relative speedups against Rhino_I for V8 benchmarks

Benchmark	Rhino		MCJS			IronJS
	C	C+	I	J	J+	
deltablue	3.05	2.69	1.37	2.50	3.78	1.53
navier-stokes	5.43	7.93	1.42	5.84	10.62	4.20
regex	1.22	1.19	0.56	0.59	0.58	0.58
richards	2.17	2.20	1.55	3.05	3.09	1.90
splay	2.50	2.40	1.25	1.49	1.48	2.10
Average	2.88	3.28	1.23	2.69	3.91	2.06

Speedup contributions breakdown: There are two factors contributing to speedups: JavaScript optimizations and hints to low-level VM. Figure 5 shows MCJS_J+ speeding up Sunspider benchmarks about 4.7× compared to the interpreted execution (29.71/6.37). 75% of this speedup is due to MuscalietJS CIL code generation boosted by array optimization, type analysis and function specialization combined with our operation implementation and property lookup. The additional 25% of the speedup is due to our special hints passed to the Mono runtime (Section 5.3).

Individual Benchmark Analysis: Figure 5 also provides details on the execution time of each benchmark in the Sunspider suite, normalized against the execution time of the same benchmarks on the Rhino interpreter. The benefit for type inference and array optimization (J+) for benchmarks like *access-nsieve*, *bitops-bit-in-bytes*, and *bitops-nsieve-bits* is very significant – 3 to 4× compared to the standard MuscalietJS JIT, which is already optimized. These benchmarks make heavy use of arrays, and the combination of type analysis and signature-based specialization makes very effective array optimizations possible.

Figure 6 shows part of the *access-nsieve* benchmark source code. In function *nsieve*, the argument *isPrime* is indexed several times in multiple loops. The standard MuscalietJS JIT converts *isPrime* accesses such as *isPrime[k]* into direct calls to property lookup functions; these require accesses to the global property cache and indirection through property descriptors (explained in Section 4.4) before finally loading the value in the internal array of the *isPrime Array* object. Applying static type analysis on this function in isolation is not enough to infer the type of *isPrime*, *k*, and *i* correctly because *isPrime* is an argument, and the values of *k* and *i* depend on another argument *m*. Instead, with signature-based function dispatch enabled, the first hot execution of the function *nsieve* triggers type inference based on the runtime types of the actual arguments, *m* and *flags*, which are passed in from the *sieve* function. The algorithm determines that *m* and *isPrime* are of type *Int* and *Array*, respectively. As a result, the type of *i* and *k* are inferred to be *Int*. This allows all accesses to *isPrime* to be converted to direct array accesses, which speeds up the algorithm significantly. For this simple example, inlining the internal function and then applying a range-based flow-sensitive type analysis may achieve similar results. However, such an analysis cannot always infer the types given the dynamism in the language, and even when it can it may need to generate extra guards in the JITed code. This type of analysis can also be expensive for long scripts. MuscalietJS is able to exploit dynamic types of arguments to provide effective optimizations using a simple, fast type inference algorithm.

For a few of the benchmarks, such as *date-format-xparb*, the speedups are not significant. Our investigation shows that these benchmarks spend a high portion of their time in JavaScript builtin functions, some of which are not implemented very efficiently in MuscalietJS. We are redesigning those builtins to improve performance and to allow them to be optimized along with the rest of the JITed code.

6.3 Web Replay Benchmarks

Table 4. Speedup vs. Rhino_I for WebReplay benchmarks

Benchmark	Rhino		MCJS			
	C	C+	I	IJ	J	J+
BBC	0.79	0.74	1.62	0.60	0.60	0.60
Yahoo	0.99	0.99	3.36	1.67	1.29	1.24
Google	0.98	0.95	3.98	2.18	0.62	0.60
Wikipedia	0.70	0.70	2.03	1.81	0.69	0.67
Mozilla	1.01	0.95	2.08	1.61	0.64	0.62
Amazon	0.99	0.96	2.12	1.83	0.60	0.60

This section presents the results for 6 web replay benchmarks, which are based on the page load behavior of six popular websites: *BBC*, *Yahoo*, *Google*, *Wikipedia*, *Mozilla*, and *Amazon*. IronJS does not support this benchmark as it only supports ECMA3, but the other evaluated engines support ECMA5 [7]. These page load benchmarks involve large JavaScript codes, but most of the code is not executed, and most code that is executed is only run once [15]. This means that JITing can account for a significant portion of overall execution time and result in major slowdown, and our results bear this out. Table 4 presents the speedups we measured for each configuration, normalized against the performance of the Rhino interpreter. For these benchmarks, Rhino performs better with just the interpreter than when using the AOT compiler at any optimization level. Similarly, MuscalietJS's interpreter (I) outperforms its JIT compiler (J and J+). The MuscalietJS interpreter also outperforms the Rhino interpreter by a factor of up to 3.98 (2.53× on average).

6.4 Effectiveness of Property Lookup Operations

To study the difference between these workloads and measure the effectiveness of the property lookup algorithm used by MuscalietJS, Table 5 reports the hit rate of our global property lookup cache (Section 4.4) and also percentage of inherited properties. For the traditional benchmarks, nearly 90% of property lookups hit in the global cache. Although not shown here, the hit rate for the second-level local property lookup caches is also very high for these benchmarks. However, for the web replay benchmarks less than half of the lookups could be found in the global cache. This reflects the fact that during page load, not only is execution time distributed across many more functions than in the data processing benchmarks, but data access patterns are also distributed across a much larger set of properties. Another interesting point here is that even for web non-repetitive Web replay benchmarks, the property cache hit rate is more than 40% which mean for over 40% of property accesses in the MuscalietJS interpreter the slow path lookup is avoided; speeding up execution as shown in Table 4.

Table 5. MuscalietJS benchmark optimization statistics

Benchmark	Global property cache hit	Percent of Inherited properties
Traditional	88.0%	33.6%
Web Replay	41.2%	33.4%

7. Conclusions

In this paper, we argue that JavaScript engines can be architected to be performant without sacrificing other desirable properties. We showed a layered JavaScript engine, with a high-level runtime handling language-specific optimizations, relying on a low-level host virtual machine to provide runtime services and machine-specific optimizations. This layered architecture combined with general-purpose code-generation hints passed from the top-level VM to the .NET VM achieves significantly performance when compared to prior layered architectures. The layered approach lets us focus on language-level optimizations, while relying on the low-level VM for efficient code generation, and using host VM features like reflection to improve performance. We see significant value in this approach, and we believe this trend will continue for other languages. As existing low-level managed VMs like CLR continue to mature, there will be an incentive to expose APIs to some of their low-level features to ease the development of layered engines.

References

[1] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 708–725, Reno/Tahoe, Nevada, USA, 2010.

[2] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. In *International Conference on Object Oriented Programming Systems Languages and Applications*, pages 195–212, Tucson, 2012.

[3] C. Chambers, J. Hennessy, and M. Linton. The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages. Technical report, Stanford University, Department of Computer Science, 1992.

[4] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *ACM Conference on Programming Language Design and Implementation*, pages 93–102, La Jolla, 1995.

[5] Dynamic Language Runtime. `http://msdn.microsoft.com/en-us/library/dd233052.aspx`.

[6] .NET Framework. `http://msdn.microsoft.com/en-us/vstudio/aa496123.aspx`.

[7] ECMAScript. `http://www.ecmascript.org/`.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM conference on Programming language design and implementation*, pages 465–478, Dublin, Ireland, 2009.

[9] Google Inc. V8 JavaScript virtual machine. `http://code.google.com/p/v8/`.

[10] IronJS. `https://github.com/fholm/IronJS`.

[11] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf. Improved type specialization for dynamic scripting languages. In *Dynamic Languages Symposium*, pages 37–48, October 2013.

[12] Mono Project. `http://www.mono-project.com`.

[13] Nashorn JavaScript engine. `http://openjdk.java.net/projects/nashorn`.

[14] Node.js. `http://nodejs.org`.

[15] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *USENIX Conference on Web Application Development*, WebApps'10, pages 3–3, 2010.

[16] Rhino JavaScript engine. `https://developer.mozilla.org/en-US/docs/Rhino`.

[17] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 677–694, Portland, Oregon, USA, 2011.

[18] SpiderMonkey: Mozilla's JavaScript engine. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[19] ECMAScript typed arrays. `http://www.khronos.org/registry/typedarray/specs/latest/`.

[20] Typescript. `http://www.typescriptlang.org/`.

[21] ECMAScript web workers. `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html#workers`.