Bootstrapping a Self-Hosted Research Virtual Machine for JavaScript

An Experience Report

Maxime Chevalier-Boisvert
Université de Montréal
chevalma@iro.umontreal.ca

Erick Lavoie
Université de Montréal
lavoeric@iro.umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Bruno Dufour
Université de Montréal
dufour@iro.umontreal.ca

ABSTRACT

JavaScript is one of the most widely used dynamic languages. The performance of existing JavaScript VMs, however, is lower than that of VMs for static languages. There is a need for a research VM to easily explore new implementation approaches. This paper presents the Tachyon JavaScript VM which was designed to be flexible and to allow experimenting with new approaches for the execution of JavaScript. The Tachyon VM is itself implemented in JavaScript and currently supports a subset of the full language that is sufficient to bootstrap itself. The paper discusses the architecture of the system and in particular the bootstrapping of a self-hosted VM. Preliminary performance results indicate that our VM, with few optimizations, can already execute code faster than a commercial JavaScript interpreter on some benchmarks.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, code generation, run-time environments*

General Terms

Algorithms, Performance, Design, Languages

Keywords

JavaScript, virtual machine, compiler, self-hosted, optimization, implementation, framework

1. INTRODUCTION

JavaScript (JS) [8] was designed by Netscape in 1995 for client-side scripting of web pages. Since then, all mainstream web browsers have integrated a JS Virtual Machine (VM) which can access the Document Object Model, placing JS in a unique position for implementing web applications.

Due to this, JS is currently one of the most widely used dynamic programming languages. Each of the main browser vendors has built their own VM, some being open-source (Apple WebKit's SquirrelFish[1], Google Chrome's V8[2], Mozilla Firefox's SpiderMonkey [6]) and some closed-source (Microsoft Internet Explorer 9's Chakra[3]), all of which use Just-In-Time (JIT) compilation.

With the increased use of JS in web applications, the performance of a browser's JS VM has gained in importance in the past few years, even appearing prominently in the browser's marketing literature. However, the performance of the best JS VMs is still lower than the performance of VMs for static languages. Although there is clearly a need to design more efficient VMs, there has been relatively little academic research on the implementation of JS. We believe this is mainly due to the lack of easily modifiable JS tools, and in particular a research VM, which would allow easy experimentation with a wide variety of new approaches. The mainstream VMs, even if they are open-source, are not appropriate for this purpose because they are large systems in which it is tedious to change even conceptually simple things (such as the function calling convention, the object representation, the memory manager, etc.) without breaking obscure parts of the system. This has motivated us to begin the design of a family of JS VMs and tools suitable for research and experimentation.

This paper is an experience report on the design of *Tachyon*, our first JS VM. Tachyon is a work in progress, and this paper discusses the current state of the system. Specifically, Tachyon has now reached the point where it bootstraps itself. The discussion of the bootstrap process is thus a central aspect of this paper.

1.1 Self-Hosting

The most remarkable feature of Tachyon is that it is written almost entirely in JS. We expect this self-hosting to yield some important benefits compared to a VM written in another language. It is typical to write JIT-based VMs in a

DLS'11, October 24, 2011, Portland, Oregon, USA.

[1] http://trac.webkit.org/wiki/SquirrelFish
[2] http://code.google.com/p/v8/
[3] http://en.wikipedia.org/wiki/Chakra_(JScript_engine)

low-level host language (e.g. C/C++) in order to have fast JIT compilation. But this is at odds with the productivity advantage of a high-level language, such as JS, for writing the complex algorithms that are found in compilers. This is an important issue for a research compiler where rapid prototyping of new compilation approaches is desirable. Since we expect Tachyon's code generation to eventually be competitive with compilers for static languages, we also believe the self-hosting will not cause the JIT compiler to be slow. Self-hosting also has the advantage of a single runtime system (memory manager, I/O, etc), which eliminates code duplication as well as conflictual interactions of independent client and host runtime systems.

1.2 JavaScript

Tachyon aims to implement JS as specified in the ECMAScript 5 (ES5) [8] specification. Although the infix syntax of JS superficially resembles that of Java, the JS semantics have much more in common with Python and Smalltalk. It is a dynamic language, imperative but with a strong functional component, and a prototype-based object system similar to that of SELF [3].

A JS object contains a set of *properties* (a.k.a. *fields* in other OO languages), and a link to a parent object, known as the object's *prototype*. Properties are accessed with the notation *obj*.*prop*, or equivalently *obj*`["`*prop*`"]`. This allows objects to be treated as dictionaries whose keys are strings, or as one dimensional arrays (a numeric index is automatically converted to a string). When fetching a property that is not contained in the object, the property is searched in the object's prototype recursively. When storing a property that is not found in the object, the property is added to the object, even if it exists in the object's prototype chain. Properties can also be removed from an object using the `delete` operator. JS treats global variables, including the top-level functions, as properties of the *global* object, which is a normal JS object.

Anonymous functions and nested functions are supported by JS. Function objects are closures which capture the variables of the enclosing functions. Common higher-order functions are predefined. For example, the `map`, `forEach` and `filter` functions allow processing the elements of an array of data using closures, similarly to other functional languages. All functions accept any number of actual parameters. The actual parameters are passed in an array, which is accessible as the `arguments` local variable. The formal parameters in the function declaration are nothing more than aliases to the corresponding elements at the beginning of the array. Formal parameters with no corresponding element in the array are bound to a specific `undefined` value.

JS also has reflective capabilities (enumerating the properties of an object, testing the existence of a property, etc) and dynamic code execution (`eval` of a string of JS code).

The next version of the standard is expected to add proper tail calls, rest parameters, block-scoped variables, modules, and many other features. Even though we are currently targeting ES5, we have been careful to keep Tachyon's design amenable to implementing the expected additions without extensive refactoring.

1.3 Contributions

This paper presents Tachyon, a meta-circular JS VM. It aims to serve as a case study for the design of a meta-circular VM for dynamic languages. The main contributions of this paper are:

- A presentation of the design of our compilation pipeline (Section 3).
- The design of low-level extensions to JS for manipulation of memory, compatible with the existing syntax (Section 3.5).
- An execution model for the VM (Section 4.1).
- A description of the bootstrap process required to initialize the VM given the execution model (Section 5).

Note that for the remainder of the paper, we use the term *compiler* to designate subsystem responsible for translating JS code to native code, and the term *VM* to refer to the combination of the runtime system and the compiler.

2. RELATED WORK

The literature on the design of meta-circular VMs for dynamic languages is rather sparse. To the best of our knowledge, no comprehensive synthesis of issues and opportunities has been done. There are, however, documented examples for some languages.

Squeak is a recent implementation of Smalltalk, written in Smalltalk. The Squeak VM is a bytecode interpreter written in a restricted, non-object-oriented subset of Smalltalk [7] that can be easily compiled to C code. This approach prevents usage of more expressive features of the language for the implementation of the VM. In contrast, Tachyon can use the entire JS subset it supports in its own implementation.

JikesRVM was the first meta-circular VM to show that high-performance was compatible with meta-circularity. Tachyon uses a similar mechanism to the JikesRVM *magic* class to expose primitive operations (Section 3.5).

Klein, a meta-circular implementation for Self [18], showed that mirror-based reflection [2] could foster much code reuse. Tachyon provides the reflection mechanisms specified in ES5 but those do not comprise mirror-equivalent mechanisms, preventing usage of the Klein implementation techniques for serializing objects in a binary image and remote debugging. Tachyon creates the objects needed at run-time during the initialization phase instead of relying on mirrors to access already existing objects (Section 4.2). Tachyon uses an x86 assembler that we implemented in JS but unlike Klein's assembler, it is not self-checking.

PyPy [16] uses a rather different approach from other meta-circular VM projects. It does not directly compile Python code into executable code. Rather, their approach involves describing the semantics of a bytecode interpreter for a programming language (e.g. Python), and generating a virtual machine (e.g. generating C code) that supports the described language. It is able to improve on the performance of the raw bytecode interpreter by applying . The system now supports most of Python and significantly improves upon the performance of the stock CPython distribution, partly due to a tracing JIT compiler [1]. Tachyon is hand-coded, and does not use automatic generation techniques.

In contrast to the aforementioned systems, Tachyon does not use a bytecode representation for compiled code. It compiles JS directly to native code. A low-level Intermediate

Representation (LIR) based on a Static Single Assignment (SSA) form is used as a platform-neutral representation for compiled code (Section 3.3) instead of bytecode.

Other researchers have studied the compilation of JS. Loitsch has proposed an approach to leverage the similarity between JS and Scheme by using it as a target language for a JS compiler [11]. Gal *et. al.* have described TraceMonkey [6], a trace-based JS compiler that achieves significant speedup by exploiting type stability in (possibly nested) hot loops to perform type specialization and other optimizations. Experiments on a popular benchmark suite reveal that the technique can achieve speedups up to 25x.

The TraceMonkey implementation was later extended by Sol *et. al.* [17] to further remove provably unnecessary overflow tests at run-time using a flow-sensitive range analysis. The authors show that their technique is able to remove more than half of the overflow checks in a sample of real JS code from top-ranked web sites. Logozzo and Venter [10] have also proposed an analysis that determines ranges as well as types (e.g. 64-bit floating-point or 32-bit integer) of numerical variables. The authors report up to 7.7x speedup for some numerical benchmarks. Such techniques could be implemented as part of Tachyon.

Some empirical studies can also shed some light on the behavior of real-world JS applications, for example by comparing properties of standard benchmarks with real applications [15, 13], or by studying the use of particular language features such as *eval* [14]. These studies will help direct future efforts and implementation choices in our project.

3. COMPILER

The compiler operates on a succession of intermediate representations getting progressively closer to native assembly code. Four representations are used with the bulk of the work being done on the third one:

- Source code: string representation of the JS code
- Abstract Syntax Tree (AST): tree representation of declarations, statements and expressions
- Intermediate Representation (IR): Control Flow Graph (CFG) in SSA form. Used to represent both the High-Level IR (HIR) and Low-level IR (LIR)
- Assembly code (ASM): array of bytes and associated meta-information representing the encoded assembly instructions

A number of phases either produce those representations or transform them. The front-end comprises the platform independent phases:

- Parsing and free variable analysis: the source code is translated into an AST and free variables are tagged (Section 3.2).
- AST->IR: the AST is transformed into the HIR (Section 3.3).
- Lowering: the HIR is transformed into the LIR and optimizations are performed (Section 3.4).

The back-end comprises the platform dependent phases:

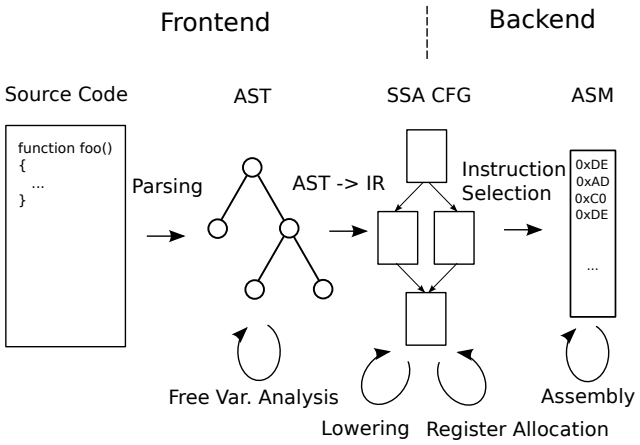

Figure 1: Overview of the compiler phases and representations.

- Register allocation: operands and destination temporaries are associated to physical registers or memory locations in the LIR instructions (Section 3.7).
- Instruction selection: the LIR after register allocation is transformed directly to a partial encoding of ASM (Section 3.10).
- Assembly: the final encoding for ASM is generated (Section 3.11).

Both representations and phases are illustrated in Figure 1.

3.1 Supported JS Subset

Tachyon currently supports enough of ES5 to bootstrap itself. The VM supports strings, arrays, closures, constructors, objects, prototypes and variable argument count. The Boolean, Error, Function, Math, Number, Object, String objects and a subset of their methods from the standard library are also supported. The internal representation for numbers uses fixed precision integers. Bitwise, logical and arithmetic operations are supported.

These functionalities were sufficient to implement the data structures and algorithms needed in pure JS. For example, we required associative tables, sets, infinite precision integers, graphs and linked lists. The VM does not yet support regular expressions, floating-point numbers[4], exceptions, object property attributes (read-only, enumerable, etc.), getters and setters.

While Tachyon does not currently support floating-point numbers, this does not confer it an unfair performance advantage over other JavaScript VMs that do. Type and overflow tests are already inserted in the generated code so as to handle floating-point operations once they are implemented. Hence, the performance is the same as if these operations were supported, but Tachyon cannot yet run code that would invoke floating-point operations.

[4]At the time of publication, regular expressions and floating-point numbers are implemented but not fully tested, and are therefore excluded from the supported JS subset.

3.2 Parsing and AST

The parser is automatically generated from the WebKit *Yacc* JS grammar. A script transforms the grammar into S-expressions which are fed to a Scheme LALR parser generator, and the resulting parsing tables are pretty printed as JS arrays. A hand written scanner and a LALR parser driver complete the parser.

The AST is then traversed to compute some properties (scope of variables, set of free variables, set of variables declared in a function, use of `eval` and `arguments` variables within a function, etc). The AST is also transformed to

- add debugging code when this is requested (for example tracing function entry and return),
- rewrite some constructions into a canonical form, such as transforming *obj*`.`*prop* into *obj*`["`*prop*`"]`, and transforming variable declarations into assignments,
- rewrite formal parameters accesses to indexing of the `arguments` object, when the `arguments` variable is accessed in the function.

3.3 Intermediate Representation

After the AST transformations, the AST is translated into the IR. This IR is in SSA form [5]. It comprises low-level type annotations which were inspired by those of LLVM [9]. We have chosen an SSA-based IR because such an IR is closer to machine code and we expect it will be efficient for recompilations to be done directly on the IR without having to restart the compilation at the AST level. The Tachyon IR is loosely divided into HIR and LIR layers, which are both refinements of a generic SSA-based IR. ASTs are initially translated into the HIR, and the HIR code is later translated into the LIR.

Each function compiled by Tachyon has an associated CFG divided into basic blocks. At the beginning of each basic block is a possibly empty list of phi nodes implementing parallel conditional assignments. These phi nodes are followed by a series of instructions, each of which produces zero or one output value. Branch instructions are only allowed at the end of basic blocks. These can either be return instructions, direct jumps, if-test instructions with two possible targets, throw instructions, or call instructions. Call instructions can have a regular continuation target and an exception target, making exception control-flow explicit.

The HIR comprises high-level operations meant to represent dynamically-typed JS operators directly. It comprises add and subtract primitives, for example, which represent the JS `+` and `-` operators, and can thus operate on any JS type. The HIR is easy to analyze in terms of JS semantics, but remains abstract. It is eventually transformed into the LIR, which translates fairly directly to machine code.

The LIR is meant to closely represent the kinds of operations that most modern computer processors implement. It is more verbose but also more expressive. By controlling how specific instances of HIR constructs are translated into LIR based on type information, the code can be specialized and optimized. The LIR exposes some low-level types such as pointers, garbage-collected references and machine integers. It is close to the expressiveness level of C code, and comprises instructions for native integer, floating-point and pointer arithmetic.

We have chosen to express these low-level operations in the IR so as to allow further optimizations on the LIR to be implemented in a portable way in the front-end. Despite being closer to machine code, the LIR remains relatively machine-agnostic. It is in SSA form and does not expose highly machine-specific details such as machine registers, stack frame formats and calling conventions. These are to be specified by a back-end tailored to the target architecture.

3.4 Optimizations

The front-end currently implements several commonplace optimizations which can operate on both the HIR and LIR. These include function inlining, Sparse Conditional Constant Propagation (SCCP) [19], Common Subexpression Elimination (CSE) as well as dead code elimination, strength reduction and peephole optimization patterns. The optimization patterns simplify control flow graphs by eliminating known patterns of redundant phi nodes, branches and instructions.

The optimizations we have implemented are fairly basic. They help improve the quality of the code generated by Tachyon, but do not yet attack the more critical performance issues involved in optimizing JS code. In particular, they are unable to optimize for more likely code paths, optimize based on type information, or reduce the cost of property accesses. We aim to implement more advanced analyses and optimizations in the future, such as optimistic optimization techniques and inlining (see Section 7).

3.5 JS Extensions

The JS language lacks some of the low-level functionality required to implement a JIT compiler. More specifically, it does not allow accessing raw memory directly, nor does it allow the execution of arbitrary code, or even file system access. As such, Tachyon is not written in pure JS, but instead in an extended dialect of the language.

One obvious way to extend JS was to implement a Foreign Function Interface (FFI) to allow Tachyon to call C functions. Once such an interface is in place, JS code can be made to call C code to implement the functionality that JS itself does not provide. One possible design choice would have been to make no further extensions to JavaScript and implement all the missing functionality required to write a JIT compiler in C. However, this would imply the implementation of significant portions of the compiler in C directly, which could be both difficult to maintain and problematic for performance, as FFI function calls are costly, opaque and non-inlinable.

Since Tachyon compiles JS code to a low-level IR that has an expressive power similar to that of C, we have been able to further extend JS with typed variables directly representing pointers and machine integers. By default, all variables in Tachyon have a *boxed* dynamic type, which can store any JS value, but functions can be annotated to say that they take typed variables as arguments (e.g.: 32 bit integer, raw pointer). External C primitives that take typed variables as input and return typed values can also be called.

The C primitives exposed to Tachyon only implement low-level operations (e.g.: `malloc/free`, file and console I/O, timing functions). To implement Tachyon primitives needing direct access to memory or lower-level hardware capabilities, we directly expose LIR instructions to JS code, in

Inline IR (IIR), a concept similar to inline assembly. Compared to inline assembly, however, Inline IR code is more machine-agnostic and more readable. It makes it possible, for example, to call the LIR load and store instructions as if they were JS functions when one needs to read or write to memory.

The snippet of code below shows the `readConsole` C function being registered with Tachyon:

```
regFFI(new CFunction(
   'readConsole',
   [new CStringAsBox()],
   new CStringAsBox(),
   params
));
```

The function `readConsole` takes a prompt string as argument and reads a user-input string from the console, which is returned. The snippet of code shown creates a proxy function that will convert a boxed string argument (JS string) into a `char*` string pointer for the C function, and perform the reverse conversion on its return value. The `readConsole` function is then used in Tachyon's read-eval-print loop as if it were an ordinary JS function.

For the inline IR, we have designed a syntax that can be parsed by an unmodified JS parser. The example in Figure 2 shows a function used to convert C ASCII strings into JS strings usable by Tachyon. This function uses the `iir.load` instruction to read character values directly from the C string, and casts them into 16-bit UTF-16 characters using the `iir.icast` instruction. The `iir.load` instruction is annotated to say that the value it is loading is a single byte (`i8` type). The output value of the instruction is thus not a boxed value, but a typed LIR value.

```
function cStringToBox(strPtr)
{
    "tachyon:static";          // Statically-linked func.
    "tachyon:noglobal";        // No global object access
    "tachyon:arg strPtr rptr"; // strPtr is a raw pointer

    // If the string pointer is NULL, return the JS null
    if (strPtr === NULL_PTR)
        return null;

    // Compute the string length
    for (var strLen = pint(0); ; strLen++)
    {
        var ch = iir.load(IRType.i8, strPtr, strLen);
        if (ch === i8(0))
            break;
    }

    // Allocate a string object
    var strObj = alloc_str(strLen);

    // For each character
    for (var i = pint(0); i < strLen; i++)
    {
        var cCh = iir.load(IRType.i8, strPtr, i);
        var ch = iir.icast(IRType.u16, cCh);
        set_str_data(strObj, i, ch);
    }

    // Compute the hash code for the new string
    compStrHash(strObj);

    // Attempt to find the string in the string table
    return getTableStr(strObj);
}
```

Figure 2: Function using Inline IR (IIR).

3.6 Implementation of the Primitives

Tachyon implements the JS semantics by mapping the primitive operations of the JS language to HIR instructions which can be translated into one or more LIR instructions. This translation is currently done in a very straightforward way: each HIR instruction maps to a call to a primitive function, which may or may not be inlined as the HIR is translated into LIR. The primitive functions may or may not use inline IR functionality to implement their semantics. The `add` primitive, for example, makes use of an LIR instruction implementing a machine addition with an overflow check so that integer additions can be optimized.

The primitive functions `getProp`, `putProp` and `hasProp` implement object property access. Other primitives implement basic operations on string and array objects. Because it would be tedious to constantly have to perform pointer arithmetic and invoke LIR instructions to access the memory layouts of these objects, we have chosen to take a helpful shortcut. Memory layouts are described in Tachyon using layout objects, which store mappings of field types and names to memory offsets. These are similar to C structs. We then use metaprogramming to auto-generate extended JS code (inlinable setter and getter methods) to access each field of a given layout. This greatly improves the readability and maintainability of our runtime implementation.

It is possible to use extended JS anywhere inside the Tachyon code, however, we have made a conscious effort to try and limit its use to the implementation of low-level primitives. Tachyon's implementation of the JS standard library, as well as most of Tachyon's implementation remains mostly written in pure JS code. This gives us the added benefit that the Tachyon LIR can be changed without an enormous refactoring effort being required.

3.7 Register Allocation

The most important compromise to be made when designing a register allocation algorithm is between compilation speed and quality of the resulting code, namely maximizing usage of registers for the most frequently executed instructions and minimizing the number of moves between registers and memory. Although earlier work was biased toward the latter, recent algorithms targeted at JIT compilers now give more weight to the former.

We initially chose to implement the Linear Scan (LS) register allocator [12], specialized for the SSA representation [20] as it seemed well-suited to our choice of IR and was reported to produce code competitive with a graph-coloring algorithm. However, in improving the implementation, we remarked that operating on live intervals instead of directly on the CFG makes the task of modelling the target architecture constraints more difficult. This motivated the implementation of a second, On-The-Fly (OTF) allocator, also operating on the SSA representation. Both algorithms use the Most Distantly Used heuristic for spilling.

Although a detailed analysis has yet to be performed, initial tests on simple benchmarks suggest that our OTF implementation is simpler, faster and produces code of similar or better quality than our LS algorithm. The current implementation of Tachyon can use either interchangeably.

3.8 Calling Convention and Register Usage

To our knowledge, no systematic exploration of the design space has been done concerning calling conventions and

register usage for compiled dynamic languages. From the very beginning, we anticipated using Tachyon to identify the most promising ideas. To accommodate such flexibility, the number and the nature of registers available for register allocation as well as for passing arguments to functions can be varied by modifying a single configuration object. We chose to reserve for a future time the factorization necessary for other possibilities such as using callee-save registers and using a register to pass the return address.

Exploration of the different possibilities is planned for the near future. Currently, Tachyon uses a single configuration corresponding to our educated guess of what would be faster. Moreover, the same register usage and calling convention are used on both x86 and x86-64. Those choices are motivated by the desire to obtain a working compiler faster with the intention of revisiting the choices made in the future if they become a bottleneck for the execution speed.

The current calling convention uses four registers to pass arguments to a function, respectively for passing the closure pointer, the `this` object and the first two parameters of the function. The stack is used to pass the return address and the remaining parameters. A location on the context object (see Section 4.1) is used for passing the argument count. For performance reasons, direct support for C calls is implemented both for x86 and x86-64. Stack alignment and proper argument passing is inlined in the generated code.

The current register usage on x86 reserves five registers for register allocation, and three registers respectively for keeping a reference to the context object, the stack pointer and a scratch register. The latter is used as a temporary measure to lessen the constraints on register allocation but we intend to eliminate it eventually.

3.9 Function arguments handling

The flexibility gained by having variadic functions by default incurs a run-time overhead when the caller does not know the expected number of parameters for a variadic function. The current implementation uses an assembly language handler prepended at the entry point of each function. This handler manipulates the call stack to match the number of expected parameters. Extra arguments are removed and missing arguments are initialized to the *undefined* value.

As a special case, when the compiler generates calls to primitive functions, since the number of arguments and the non-usage of the `arguments` object are known at compile-time, a fast entry point is used which avoids passing and checking the argument count at run-time.

The `arguments` object also incurs a run-time cost because the object in question needs to be constructed. For functions using it, all arguments passed on the stack are copied in an array, before the check for the number of arguments is performed and the stack frame is manipulated. A handler for creating the `arguments` object is prepended to the variadic function handler for functions making use of the feature.

Note that the `arguments` object contains function arguments, but also a reference to the callee function object. This information is available to functions because the function object is a hidden argument in our default calling convention. Some JavaScript implementations also include a reference to the caller function in the `arguments` object. Tachyon does not provide this because it is not part of the ES5 standard. We do not believe it would be difficult or costly to implement, however.

3.10 Instruction Selection

Instruction selection is done after register allocation. It tries to exploit faster and/or smaller instructions when possible. For example, when performing an addition with an immediate value of 1, the `inc` instruction will be used instead of the regular `add` instruction. Most of the work concerns ensuring the proper location of operands with regard to x86 operand conventions, such as not having two operands in memory and having one of the operands being the destination of the instruction.

Instruction selection notation uses regular JS mixed with a selected subset designed to mimic the GNU assembler syntax. Inside regular code, an assembly language context is initiated by referring to the assembler object, `asm` by convention. Then, using `$` as an immediate value constructor, `mem` as a memory address constructor, variable names to refer to register objects, as well as cascading function calls [4], namely methods returning the `this` object, allows to write assembly code in the style shown below:

```
asm.
mov(eax, ebx).        // 000000 89 c3         movl %eax,%ebx
add($(1), ebx).       // 000002 83 c3 01      addl $1,%ebx
add(mem(0,esp), ebx). // 000005 67 03 1c 24   addl (%esp),%ebx
ret();                // 000009 c3            ret
```

This has proved helpful in working at different levels of abstraction, all within the same language. For example, JS can be used as a metaprogramming language for writing assembly code by defining cascading functions implementing common idioms. The following example illustrates copying values from the stack to an array, using a for loop generator pattern:

```
var i = eax;

asm.
forLoop(i, ">=", $(0), function () // for (;i >= 0; --i)
{
    this.
    mov(mem(0, sp, i), temp).      // temp = sp[i]
    mov(temp, mem(0, arr, i));     // arr[i] = temp
}).
ret();
```

Interleaving assembly instructions with generator patterns makes assembly writing more convenient and arguably easier to read than with a regular assembler.

3.11 Assembly

Once a partial encoding for instructions has been generated by the instruction selection phase, a fixpoint on the assembly code generated is performed to find the minimal length encoding for the given assembly language instructions. This is necessary because the encoding length for some instructions varies as a function of the value of some operands. For example, encoding a relative jump with an 8-bit displacement uses only two bytes instead of five for a 32-bit displacement.

3.12 Compilation Example

```
function add1(n)
{
   return n + 1;
}
```

Above is the source code for a simple JS function which computes `n+1`. Due to the generic nature of the `+` operator,

```
Program                          ("ex.js"@1.1-1.36:)
|-var= add1 [global]             ("ex.js"@1.1-1.36:)
|-func= add1 [global]            ("ex.js"@1.1-1.36:)
|-block=
| BlockStatement                 ("ex.js"@1.1-1.36:)
| |-statements=
| | FunctionDeclaration          ("ex.js"@1.1-1.35:)
| | |-id= add1 [global]          ("ex.js"@1.1-1.36:)
| | |-funct=
| | | FunctionExpr               ("ex.js"@1.1-1.35:)
| | | |-param= n                 ("ex.js"@1.15-1.16:)
| | | |-var= n [local]           ("ex.js"@1.1-1.35:)
| | | |-body=
| | | | ReturnStatement          ("ex.js"@1.20-1.33:)
| | | | |-expr=
| | | | | OpExpr                 ("ex.js"@1.27-1.32:)
| | | | | |-op= "x + y"
| | | | | |-exprs=
| | | | | | Ref                  ("ex.js"@1.27-1.28:)
| | | | | | |-id= n [local]      ("ex.js"@1.1-1.35:)
| | | | | | Literal              ("ex.js"@1.31-1.32:)
| | | | | | |-value= 1
```

Figure 3: AST for the `add1` function.

this either adds one if `n` is a number, or converts `n` to a string, if it isn't already one, and concatenates the string `"1"` to it.

Figure 3 illustrates the AST produced by our parser for the `add1` function. This AST includes the function declaration, the block of statements inside the function body, the variables and their scope, as well as the operator expression adding `n` to `1`, wrapped inside a return statement. The HIR produced for this function is shown below:

```
entry:
box n = arg 2;
box $t_4 = call <fn "add">, undef, undef, n, box:1;
ret $t_4;
```

As can be seen, this representation is rather concise and abstract. The value of the argument `n` is assigned to an SSA temporary. The primitive `add` function is then called to implement the behavior of the `+` operator applied to `n` and `1`. The result of this call is then returned. The two `undef` arguments to the call represent the closure and `this` object references, which are undefined in the case of primitive calls.

```
entry:
box n = arg 2;
pint $t_4 = and_box_pint n, pint:3;
if $t_4 === pint:0 then cmp_true else if_false;

cmp_true:
box $t_14 = add_ovf n, box:1 normal call_res overflow ovf;

call_res:
box phires = phi [$t_11 ovf], [$t_19 if_false], [$t_14 cmp_true];
ret phires;

ovf:
ref $t_9 = get_ctx;
box global_2 = load_box $t_9, pint:36;
box $t_11 = call <fn "addOverflow">, undef, global_2, n, box:1;
jump call_res;

if_false:
ref $t_17 = get_ctx;
box global_3 = load_box $t_17, pint:36;
box $t_19 = call <fn "addGeneral">, undef, global_3, n, box:1;
jump call_res;
```

Figure 4: LIR for the `add1` function.

The LIR for `add1` is produced by inlining the call to the primitive `add` function (see Figure 4). This results in many basic blocks being added to `add1`. This code implements the multiple semantics of the `+` operator. The tag bits of the operand values are first tested to see if both operands are integers. The test of the tag bits of the constant are eliminated by constant propagation. If `n` is an integer, we use the `add_ovf` instruction to perform an integer add with an overflow check directly on the bits of the values.

If the result overflows, the `add_ovf` instruction will branch to the `ovf` basic block, in which the `addOverflow` function is called to handle this case. If `n` was not an integer to begin with, the `addGeneral` function is called to implement the generic addition semantics, which may result in a string concatenation, for example. In all cases, the control-flow eventually reaches the `call_res` block and the final result of the addition is passed to the `phires` phi node, whose value is then returned.

```
0000                        <fn:add1>
/* stack adjustment prelude removed */
0051                        entry:
0051 89 c7                  movl %eax,%edi
0053 83 e7 03               andl $3,%edi
0056 85 ff                  testl %edi,%edi
0058 75 3b                  jne if_false
005a eb 00                  jmp cmp_true
005c
005c                        cmp_true:
005c 89 c7                  movl %eax,%edi
005e 83 c7 04               addl $4,%edi
0061 71 02                  jno call_res
0063 eb 08                  jmp ovf
0065
0065                        call_res:
0065 83 c4 04               addl $4,%esp
0068 89 f8                  movl %edi,%eax
006a c2 00 00               ret $0
006d
006d                        ovf:
006d 89 ce                  movl %ecx,%esi
006f 8b 76 24               movl 36(%esi),%esi
0072 89 fb                  movl %edi,%ebx
0074 89 1c 24               movl %ebx,(%esp)
0077 bf 00 00 00 00         movl <addOverflow_fast>,%edi
007c 89 f5                  movl %esi,%ebp
007e be 04 00 00 00         movl $4,%esi
0083 ba 19 00 00 00         movl $25,%edx
0088 c7 41 04 04 00 00 00   movl $4,4(%ecx)
008f ff d7                  call *%edi
0091 89 c7                  movl %eax,%edi
0093 eb d0                  jmp call_res
0095
0095                        if_false:
0095 89 cf                  movl %ecx,%edi
0097 8b 7f 24               movl 36(%edi),%edi
009a 83 ec 04               subl $4,%esp
009d 89 3c 24               movl %edi,(%esp)
00a0 bf 00 00 00 00         movl <addGeneral_fast>,%edi
00a5 8b 2c 24               movl (%esp),%ebp
00a8 ba 19 00 00 00         movl $25,%edx
00ad be 04 00 00 00         movl $4,%esi
00b2 c7 41 04 04 00 00 00   movl $4,4(%ecx)
00b9 ff d7                  call *%edi
00bb 83 c4 04               addl $4,%esp
00be 89 c7                  movl %eax,%edi
00c0 eb a3                  jmp call_res
```

Figure 5: x86 assembler for the `add1` function.

Finally, the x86 assembler code (32-bit) produced for the `add1` function is shown in Figure 5. For brevity, this snippet is missing the prelude that adjusts the stack frame if the number of arguments is different than expected. The ma-

chine code is generated based on the LIR. The basic blocks are ordered so as to try to linearize the most likely code paths. Basic LIR instructions typically require very few machine instructions. For example, `add_ovf` is implemented as a machine add followed by a jump that tests the overflow condition flag. The x86 code for the `add1` function fits within 194 bytes.

4. RUNTIME

4.1 VM and Program Execution Model

The evolution of dynamic languages shows a trend toward late-binding more and more elements of the language. Accordingly, the implementation of those languages also shows an increasing complexity in their run-time behavior. The availability of the compiler at run-time allows it to manipulate runtime structures required by the program being compiled. This makes for an execution model for the VM that blends compile-time and run-time behaviors. This section explains the particular choices made for the execution model of Tachyon. We introduce the following definitions to simplify reasoning about the execution model in the context of meta-circularity: we refer to the environment in which the compiler executes as the *host* environment, and the environment in which the compiled code executes as the *client* environment.

During execution, a Tachyon program executing in the client environment needs access to a number of data structures. Those data structures are accessed through a context structure. It holds references to the JS global object, a string table and heap allocation pointers. The context structure is an implementation artefact, not accessible as a JS object.

During compilation, the compiler accesses the client environment to initialize resources needed by the compiled code. We chose to create strings in the client environment to avoid maintaining a compile-time string table that would duplicate the runtime string table used by the executing program and avoid the run-time cost of internalizing strings. To be able to inline primitive implementations in the generated code, the compiler needs access to the IR of those primitives. These are maintained in the host environment. Also, the compiler needs access to the OS API functionalities not exposed in JS such as allocating executable memory. This is done through proxies.

The result of a successful compilation is an executable machine code block (MCB), which will be referred after initialization by a JS function object in the client environment. Tachyon also maintains the IR of the function in the host environment to allow recompilation, but this feature has not been used yet. The execution model is illustrated in Figure 6.

Tachyon creates runtime strings during compilation, this technique could also be applied more generally for the creation of runtime function objects maintaining compilation information and manipulation of the global object by the compiler.

When executing on an existing JS implementation, the host environment is necessarily different from the client environment. Once bootstrapped, it would be beneficial to have the host and client environment be the same as this allows sharing of resources between the compiler and compiled code, such as the string table. To keep the implementation simple, however, the compiler currently still executes

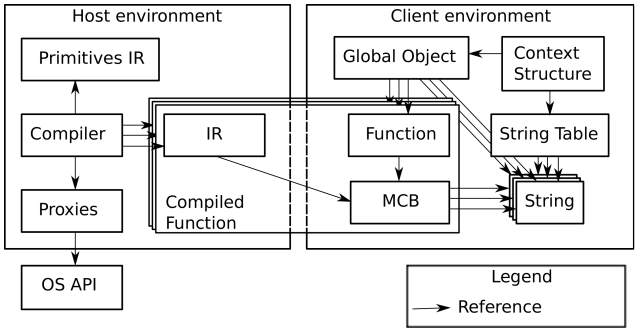

Figure 6: Execution model.

in a different environment. Sections 4.2 and 5 respectively explain the initialization of the client environment and the bootstrap process.

4.2 Initialization Process

We have designed Tachyon to self-initialize. The host environment compiles code to be run in the client environment. The client code is then able to initialize its own objects in its own heap. The host environment never has direct access to the objects in the client heap. This method was chosen because it avoids the need for an interface layer between the host and client object representations. Only the client code needs to know how to manipulate objects inside its heap.

Initialization of the client environment requires a heap in which allocation of objects can occur, a context structure to maintain bookkeeping information and runtime services such as a string table.

The very first operation consists in requesting a contiguous memory space for allocation (heap) from the operating system. This is done through a C `malloc` call.

Next, the compiler needs access to the IR of primitives. To obtain them, the primitives are recompiled from source code and their IR representation are stored on the configuration object in the host environment. This is in turn sufficient to compile bridges between the host environment and the client environment. Bridges reuse the FFI implementation and use C as a common interface language between the host environment and the client environment.

Once the heap, the IR of primitives and bridges are available, the client environment is initialized by allocating the context structure through a FFI call. Since allocation of the context structure requires a context structure, the recursion is avoided by partially initializing the context to allow it to be allocated through the regular allocation mechanism.

Now that object allocation is possible in the client environment, the string table is initialized. Note that all the primitives needed for the previous phases cannot rely on strings for correct behavior since those are not available until this point. However, they might still reference them as long as the code is not executed. At this point, strings used by primitives are allocated in the string table and references are linked in the executable code.

The client global object is then allocated and initialized in the client environment. This allows compilation and initialization of the standard library. Once the standard library is initialized and the system is ready to compile client code, the initialization process is finished.

4.3 Bridges

Tachyon needs to be able to make calls to the client code it compiles in order to initialize it and run it. This process is non-trivial because when Tachyon runs under its host platform, the code it needs to call into uses a calling convention that is not supported by the host platform. Furthermore, even if Tachyon was running independently of a host platform, we may want to change the Tachyon calling convention for a new bootstrap, which would result in a similar scenario. Tachyon may use a different calling convention from the code it is compiling.

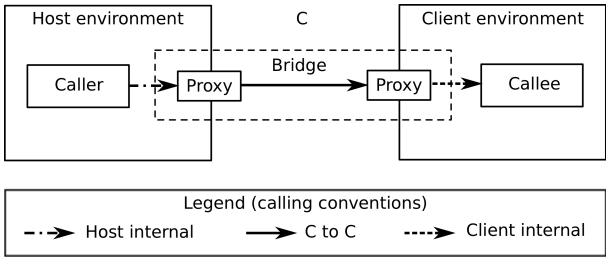

Figure 7: Calling convention bridge.

To resolve this issue, we have designed a mechanism we call a bridge. The Tachyon back-end provides support for calling C functions. It is also able to generate functions that are callable using the C convention. The current Tachyon host platform (Google V8) supports calling into C functions as well. This allows us to implement a system that can call client functions from the host environment by using the C calling convention. We do this by creating a proxy supporting incoming C calls on the client side, and another proxy that exposes the client proxy using the host calling convention on the host side. The host function can then call into its proxy using the host calling convention, which calls the client proxy using the C calling convention, which finally calls into the client function using the client calling convention. This is illustrated in Figure 7.

Bridges are not like the mirrors of Klein [18], which allow reflective access to objects residing in a remote VM. Rather, they are a lower-level mechanism that allows us to call functions residing in another VM while properly handling the discrepancies in calling conventions and argument type conversions. Tachyon uses them to initialize a new VM during the bootstrap process. We may eventually use bridges as a tool in implementing mirror-like facilities.

4.4 Object Representation

Heap-allocated structures in Tachyon are referenced through boxed values whose least significant bits (tag bits) identify the kind of object being referred to. Those structures all begin with a 32-bit header that encodes more precise information about the exact layout and size of the structures so that they can be traversed by a GC.

JS objects are currently represented in memory in a straightforward way. The object structure stores a property count, a prototype reference and an indirect reference to another structure which is a hash map of property names to property values (see Figure 8). This indirect reference is present so that the property map can be reallocated when the number of properties grows beyond the current capacity of the property map. The prototype property refers to the object's prototype object. It may be `null` if the object has no prototype. It is stored outside of the property map because every object must have this field and it must not be directly visible as a property of the object.

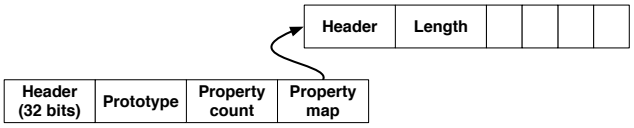

Figure 8: JS object memory layout.

In JS, it is possible to use arrays and functions as regular JS objects. That is, named properties can be stored onto them. They also have a prototype field, just as with regular objects. Because of this, we have chosen to implement arrays and functions as extensions of regular JS objects. This means that arrays and functions share a common part of their memory layout with regular objects. Namely, the prototype, property count and property map fields. They also possess additional fields specific to their implementation.

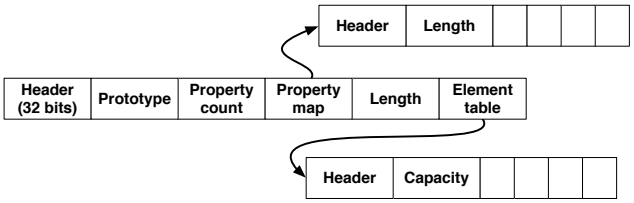

Figure 9: Array memory layout.

In the case of arrays, they also store a length field (the length of the array, or number of indexed values stored) and a reference to an element table (see Figure 9). The element table stores a capacity field so that additional space beyond the length of the array can be reserved for future resizing. This table will be reallocated if the array size increases beyond the capacity.

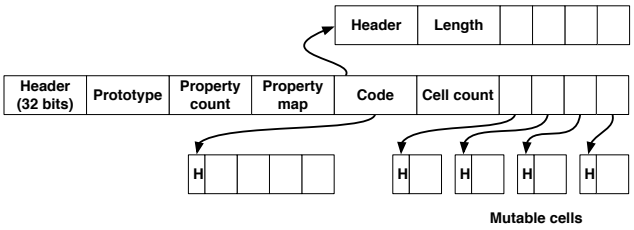

Figure 10: Function memory layout.

Function objects in JS are a representation of closure instances. In addition to the normal object fields, they also store a pointer to the function's machine code and a fixed number of references to mutable cells (see Figure 10). Each mutable cell is heap-allocated and contains a header and a mutable boxed value field. These mutable cells serve to store mutable variables captured by the closure. This representation of closures is one favored by many Scheme im-

plementations. We plan to eventually optimize our closure representation by allocating mutable cells only for the captured variables that are shared among multiple closures.

Figure 11: String memory layout.

Strings in JS are not objects. They are immutable primitives, and as such, cannot store properties like objects, arrays and functions. Because of this, we have designed a layout (see Figure 11) for them in which only the length and the raw UTF-16 character data are stored.

5. BOOTSTRAP

The boostrap process of Tachyon is performed in memory to avoid the creation of a separate executable or image. This is a temporary measure until support for an image writer is added.

We define the hosted compiler to be the compiler executing in the host environment, the bootstrapped compiler to be the compiler produced by the hosted compiler and executing in the client environment, and the boostrapped client environment to be the environment initialized with the bootstrapped compiler. An illustration of the hosted and bootstrapped compilers is given in Figure 12.

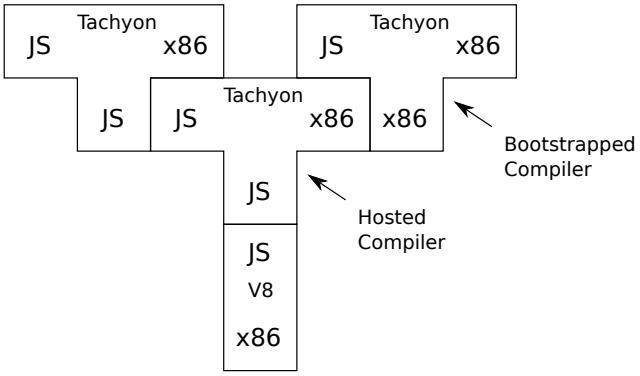

Figure 12: Bootstrap stages.

The following steps are performed to achieve a bootstrap in memory:

1. Initialization of the client environment with the hosted compiler.
2. Compilation of the Tachyon source code and shell with the hosted compiler.
3. Initialization of the bootstrapped client environment with the bootstrapped compiler.
4. Execution of the compiler and shell.

Note that when a bootstrap is performed in memory, the control never returns to the initial host environment, frames from the host environment are kept on the stack and the memory allocated by the host process is still active.

Initialization of the bootstrapped client environment could be avoided if the client environment was reused instead, since their run-time behavior is identical. They were kept separate for simplicity of implementation for this version of the compiler, although we plan on merging them in the near future.

6. PERFORMANCE

In this section, we present some performance comparisons of Tachyon against other JS implementations. Performance has not been an important concern up to this point in Tachyon's development. Therefore, these numbers are meant as ballpark figures only. The benchmark numbers shown were measured on a computer with quad Intel Xeon X5650 CPUs, running the Linux 2.6 kernel. Tachyon[5] was compared against Google V8 revision 7878 and WebKit's interpreter revision 88541. All implementations were compiled in 64-bit mode.

A large proportion of widely-used JS benchmarks available rely on features not yet supported by Tachyon, such as regular expressions, the `Date` object and floating-point numbers. We have chosen to use the benchmarks from the *SunSpider* JS benchmark suite which can run in Tachyon without modification to compare our VM against Google V8 and the WebKit interpreter. The resulting times are shown in Table 1. Because the original benchmarks run very quickly (< 10ms), the times measured are for a total of 400 runs of each benchmark.

Benchmark	Tachyon	Google V8	WebKit Int.
access-binary-trees	20.522	0.473	4.553
access-fannkuck	18.002	2.445	34.871
access-nsieve	4.839	0.686	7.355
bitops-3bit-bits-in-byte	1.890	0.636	9.019
bitops-bits-in-byte	6.301	2.294	11.723
bitops-bitwise-and	30.971	3.436	7.915
bitops-nsieve-bits	7.005	2.347	15.39
controlflow-recursive	6.275	0.739	6.076
crypto-md5	13.789	0.724	7.357
crypto-sha1	15.050	0.870	7.372

Table 1: Running times (in seconds) of *SunSpider* benchmarks under Tachyon, Google V8 and the WebKit interpreter.

The results in Table 1 show that the Tachyon JIT currently produces code that is several times slower on average than that produced by Google V8's JIT. We believe this is largely due to the fact that object property accesses (including global variable accesses and global function calls) are unoptimized in our system. Each such access requires a function call and a hash table lookup on the object. Benchmarks which do not involve property accesses, such as `bitops-3bit-bits-in-byte` are those where Tachyon compares the least unfavorably to V8. Conversely, Tachyon does significantly worse than both V8 and the WebKit interpreter in `access-binary-trees`, a benchmark that is very heavy in terms of property accesses.

At the time of this writing, the Tachyon source code, excluding unit tests and automatically generated parser code, occupies approximately 75 KLOC, compared to around 375 KLOC for V8 and 550 KLOC for SpiderMonkey. Tachyon is

[5]https://github.com/Tachyon-Team/Tachyon/tree/dls2011

clearly still in its infancy, but since it is a large and complex piece of software, we have decided to also use the time it takes Tachyon to compile itself as a benchmark. We believe this is a more representative measure of Tachyon's performance than the JS microbenchmarks widely used today.

Benchmark	Tachyon	Google V8
Tachyon compilation time	1991	165

Table 2: Compilation times (in seconds) for Tachyon under Tachyon and Google V8.

Compilation times for Tachyon, first running under V8 and then running under itself are given in Table 2. These numbers indicate that Tachyon's overall performance is about an order of magnitude slower when compiled by itself than when running under V8.

We believe that these numbers are encouraging. Tachyon, despite its limitations, does significantly better than WebKit's bytecode interpreter on several benchmarks. We have already started work to improve Tachyon's performance. For instance, we have started prototyping a code patching mechanism to optimize accesses to globals. The substantial performance improvements obtained encourage us to explore other "low-hanging fruit" optimizations. We believe that we may soon reach a level of performance that is more competitive with that of V8.

7. FUTURE WORK

The Tachyon bootstrap currently occurs inside a custom heap allocated inside the Google V8 process. As such, if one wants to execute Tachyon while bootstrapped, we currently have to begin the bootstrap compilation of Tachyon anew. We are currently working on the implementation of a memory dump of the Tachyon heap (all machine code and heap data) into an ELF binary image file. This will allow Tachyon to become truly independent of its host platform.

Tachyon currently has no garbage collector. It has been built with a compacting, generational GC in mind, but this GC is not yet complete. This currently imposes a limit on the amount of memory programs can allocate during their execution. Work has begun on the implementation of a GC. This collector will initially be a single-threaded compacting GC, and will be written in plain C, because of the wide availability of debugging tools. However, we plan to eventually rewrite this in our extended JS dialect. We believe this will make the GC easier to maintain in the long run, as C code does not have direct access to the definitions of memory layouts used by our heap-allocated objects.

Our current object representation is very simple. Each object stores a hash map of property names to property values. While easy to implement, this is inefficient both in terms of running-time and space usage. As such, we plan to factor out the name of properties and their position by introducing a layout object, called maps in SELF [3] and hidden classes in V8. An object then keeps a reference to its layout object and only stores the values of its properties. Layout objects can be shared by multiple objects with the same layout.

The subset of ES5 supported by Tachyon has proven sufficient to compile Tachyon itself. A few features are still missing, however. Namely floating-point numbers, exceptions, regular expressions, `eval` and object property attributes. We do not foresee any difficulties in implementing the missing features. The compiler has been designed with support for features like `eval` in mind. Once these missing features are implemented, we believe Tachyon will be able to run most JS benchmarks currently available.

Besides getting Tachyon to support all of ES5, one of our medium-term goals is to bring its performance to a competitive level. Our project has thus far been mostly focused on rapidly achieving a working bootstrap compilation. However, we believe it is important for our compiler to generate quality code if we are to compare it to other existing implementations. As such, we aim to reach a performance level within a factor of two of the best existing JS implementations within the next year. Another performance goal is to improve the speed at which Tachyon compiles source code. This will be partly achieved by having Tachyon generate better code when compiling itself.

Since Tachyon is being developed as a research platform, we intend to use it to experiment with novel ideas. One area we plan to explore is the use of more aggressive speculative optimizations. We intend to test the concept of "optimistic" optimizations: optimizations that are likely to be safe given the current state of a running program, but are not guaranteed safe for its entire execution. Such opportunities for optimizations can be discovered using combination of profiling and static analysis. Aggressively optimized code will then be generated under the optimistic assumption that the said optimizations will remain applicable, but guards need to be inserted in the code so that it can be deoptimized should this assumption be invalidated.

Tachyon currently runs inside of the Google V8 shell, which is a console program. This allows us to implement the ES5 specification, but does not allow us to use Tachyon inside a web browser. Since the main use for JS at this time is within web pages, it is one of our main goals to eventually integrate Tachyon into a web browser of some sort. Mozilla Corp. has expressed interest in implementing an HTML DOM tree in JS for an experimental web browser. We believe this may be a very interesting platform for Tachyon to integrate into. We are also looking at *node.js* as a possible alternative.

8. CONCLUSION

JS is currently one of the most widespread dynamic languages. As web applications become more complex, JS performance is becoming increasingly important. Existing JS engines, even when they are open-source, are difficult to modify. A flexible research platform is therefore needed in order to design, implement and evaluate new compilation techniques for JS. We have presented Tachyon, a self-hosted JS virtual machine that aims to fill this void. Tachyon is itself written in an extended dialect of JS. We have shown that this implementation decision allows the system to be quickly developped and easily modified. Tachyon is thus well-suited to rapid prototyping of new compilation strategies. For instance, it already supports two different register allocators and two target architectures. Tachyon currently supports a subset of the full ES5 specification that is sufficient to enable the bootstrapping process.

We have shown that the current version of Tachyon, despite not having been optimized for performance of the generated code, is faster than the WebKit interpreter on some

benchmarks from the popular *SunSpider* suite. We believe that, once some straightforward optimizations are added, the performance of Tachyon will be competitive with existing JS JIT-based VMs. The flexibility of the compiler will also allow deeper optimizations to be investigated.

Tachyon is publicly available under a Modified BSD license on GitHub. Contributions are welcome!

9. ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and Mozilla Corporation.

We wish to thank David Haguenauer, Éric Thivierge, Olivier Matz and Alexandre St-Aubin for reviewing drafts of this paper.

10. REFERENCES

[1] C. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 2009 workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 2004 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM.

[3] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the 1989 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM.

[4] D. Crockford. *JavaScript: The Good Parts*, chapter 4, page 42. O'Reilly, 2008.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 1989 ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478. ACM, 2009.

[7] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 32, pages 318–326. ACM, ACM Press, 1997.

[8] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the 2004 IEEE/ACM international symposium on Code generation and optimization*, 0:75, 2004.

[10] F. Logozzo and H. Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation–Application to JavaScript Optimization. In *Proceedings of the 2010 international conference on Compiler construction*, pages 66–83. Springer, 2010.

[11] F. Loitsch. JavaScript to Scheme compilation. In *Proceedings of the 2005 Workshop on Scheme and Functional Programming*, pages 101–116, september 2005.

[12] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, September 1999.

[13] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, page 3. USENIX Association, 2010.

[14] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do – a large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming*. Springer, 2011.

[15] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12. ACM, 2010.

[16] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 2006 ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[17] R. Sol, C. Guillon, F. M. Q. a. Pereira, and M. A. S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 2011 international conference on Compiler construction*, pages 2–21, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 2005 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[19] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[20] C. Wimmer and M. Franz. Linear scan register allocation on SSA form. In *Proceedings of the 2010 IEEE/ACM international symposium on Code generation and optimization*, pages 170–179, New York, NY, USA, 2010. ACM.