Tachyon: a Self-Hosted JavaScript VM

Maxime Chevalier-Boisvert
Erick Lavoie
Bruno Dufour
Marc Feeley

Université de Montréal

Outline

Tachyon:
a hypothetical particle that travels faster than light.
– Oxford Dictionary

- **System overview** – *Marc Feeley*
- **IR and optimization** – *Maxime Chevalier-Boisvert*
- **Back-end, profiling and benchmarks** – *Bruno Dufour*

Unfortunately *Erick Lavoie* could not be here (responsible for back-end and register allocator)

Goals

- Tachyon JS project started March 2010
- JS is a key language with a bright future
- 2010 JS VMs performance... **We can do better!**

- Goal #1: compiler for **research on dynamic languages**
  - optimistic optimization
  - object representation / garbage collection
  - profiling real-world applications
  - language extensions: exact arithmetic, continuations, tail calls, exceptions, concurrency, distributed computing, Harmony, ...
  - **must stay flexible!** e.g. calling protocol, object layout, ...

- Goal #2: **production-quality** high-performance open-source JS compiler for client-side and server-side scripting

Self-Hosting in a Dynamic Language

- JIT compilers are efficient because they
  1. compile the program parts where speed matters
  2. **adapt/specialize the program** to the run time conditions
- **Particularly useful to speed up dynamic languages**
- **Dynamic compilation may outperform static compilation**
- ... but only if **compilation time is kept small**
  - avoid expensive optimizations
  - use hand-tuned algorithms
  - break abstraction barriers
- **Not obvious when host is a dynamic language!**

Self-Hosting JIT Thesis

- Our idea: **dynamically recompile the JIT compiler**

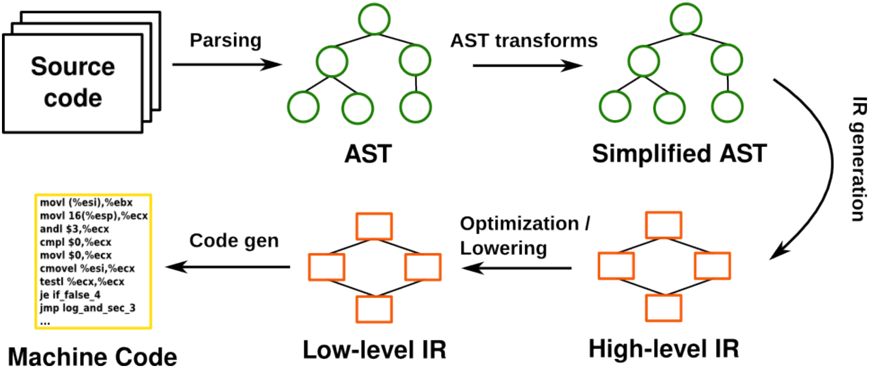
- **Adapts the JIT compiler to the program being compiled**
- The cost of JIT compiler adaptation is amortized over **repeated recompilations of the same program**
- Host language must also be JIT compiled... **Recursion!**

- Other self-hosting benefits:
  - single runtime system (one heap, GC, I/O, ...)
  - easy introspection and dynamic language extension
  - we are more productive in JS than C/C++!

Current State

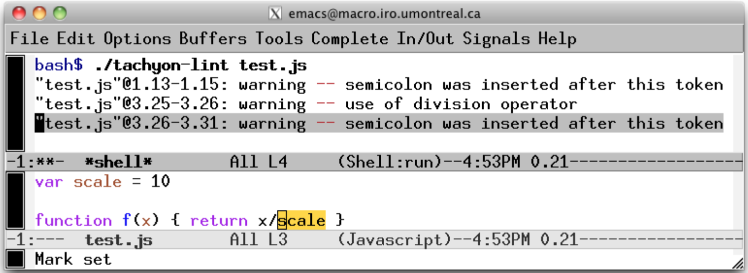
- Compiler is written in a subset of JS (roughly 40 KLOC):
  - parsing
  - analysis/optimization of IR
  - linear-scan reg. alloc.
  - assembler and linker for **x86** and **x86_64**
  - stdlib in JS
- Implements ES5 spec except:
  - floating point (only fixnums are supported)
  - regular expressions
  - getters and setters, property attributes
  - `eval` and most meta-object protocol features
  - no GC yet!
- Goodies: interactive shell, profiler, benchmarking framework
- March 2011: Tachyon self-bootstrap using V8

Tachyon Compiler Pipeline



Parser

- Based on WebKit's yacc grammar
  - `Grammar.y` converted to LALR-SCM parser generator spec
  - LALR-SCM tables pretty-printed as JS arrays (6 KLOC)
  - hand written scanner (1 KLOC) and LALR driver (.3 KLOC)
- Parses 100 KLOC per second
- Source location attached to all AST nodes



Parser Derivatives for Debugging

- **JS pretty-printer** – dump or pretty print AST
- **js2scm** – compile JS to Scheme
- **js2js** – instrument JS code with function entry/exit tracing

```
bash$ ./js2js -debug tachyon.js > tachyon-debug.js
bash$ d8 tachyon-debug.js
| (( "utility/iterators.js"@13.1-17.2: Iterator
| | (( "utility/debug.js"@73.1-77.2: assertNew
| | | (( "utility/debug.js"@62.1-68.2: isGlobal
| | | | (( "utility/debug.js"@65.19-65.47: isGlobal
| | | | )) "utility/debug.js"@65.33-65.45: isGlobal
| | | )) "utility/debug.js"@67.5-67.27: isGlobal
| | )) "utility/debug.js"@73.1-77.2: assertNew
| )) "utility/iterators.js"@16.5-16.17: Iterator
| (( "utility/iterators.js"@13.1-17.2: Iterator
| | (( "utility/debug.js"@73.1-77.2: assertNew
...
```

AST Transformations

- Simplification of AST
- Resolve scopes and compute free variables
- Detect uses of `eval` and `arguments`

```
// rewrite property access
obj.prop  ==>  obj["prop"]

// variable declaration hoisting to function top
function f(...) {        for (var x = E; ...) ... }  ==>
function f(...) { var x; for (    x = E; ...) ... }

// alias parameters to arguments object
function f(x,y) {            ... arguments ...; x = y; }  ==>
function f()    { var a = arguments; ... a ...; a[0] = a[1]; }
```

High-Level IR

- Core JS semantics expressed in terms of HIR instructions
  - Act on boxed, high-level (dynamic) types
    - Strings, numbers, objects, ...
  - May produce exceptions in some cases
- Examples
  - Arithmetic/comparison operators (+, −, ==, ...)
  - Property accesses (getProp, putProp, hasProp, ...)
  - Misc. JavaScript operators (`typeof`, `instanceof`, ...)
- Currently, HIR implemented directly by **primitive functions**

Primitive Functions

- All Tachyon functions can use our extended JS, meaning:
  - Special function annotations
  - Typed local variables
  - Inline IR (IIR) instructions
- Core of the runtime makes more use of this
  - Rest of Tachyon mostly uses 'standard' JS
- Annotations allow functions to:
  - Have static linkage (no dynamic lookup)
  - Always be inlined
  - Be prevented from accessing the global object
  - Use typed argument values
  - Use a typed return value

```
/**
Implementation of HIR less-than instruction
*/
function lt(v1, v2)
{
    "tachyon:inline";
    "tachyon:nothrow";

    // If both values are immediate integers
    if (boxIsInt(v1) && boxIsInt(v2))
    {
        // Compare immediate integers without unboxing
        var tv = iir.lt(v1, v2);
    }
    else
    {
        // Call a function for the general case
        var tv = ltGeneral(v1, v2);
    }

    return tv? true:false;
}
```

```
/**
Test if a boxed value is integer
*/
function boxIsInt(boxVal)
{
    "tachyon:inline";
    "tachyon:nothrow";
    "tachyon:ret bool";

    // Test if the value has the int tag
    return (boxVal & TAG_INT_MASK) == TAG_INT;
}
```

Tachyon's Extended JavaScript

- JavaScript has no access to raw memory
  - Essential to implement a VM/JIT
- Tachyon is written in JS w/ unsafe extensions
  - Minimizes the need to write C code (FFI)
  - Exposes potential optimization opportunities
    - FFIs are optimization boundaries
- JS code translated to low-level typed IR
  - JS extension: insert inline IR (IIR) in source code

Inline IR example

```
if (boxIsInt(v1) && boxIsInt(v2)) {
   // Compare immediate integers without unboxing
   var tv = iir.lt(v1, v2);
}
```

Low-Level IR

- Some similarities with LLVM
- SSA-based
- Type-annotated
  - Integers, floats, booleans, raw pointers
  - Boxed values, references
- Low-level
  - Mirrors instructions commonly found on most CPUs
    - add/sub/mul/div, and/or/shift, jump/if/call, load/store, etc.
  - Still tries to be machine agnostic
    - No specific endianness, no registers
  - Allows expressing more optimizations (specialization)

IIR $\neq$ Assembly

- Writing code using IIR not as painful as it sounds
- Using IIR does not take away JS capabilities!
  - Still get dynamic typing, strings, closures
- Don't need to annotate the type of every local variable
- Also facilitated by auto-generated code
- Auto-generated accessor methods for heap objects
  - alloc_str(len), get_str_len(str), get_str_data(str, i)
- Auto-generated wrappers for C FFI wrappers
  - puts('Hello World!'), malloc(size), free(ptr), exit(intval)

```
function newObject(proto)
{
    "tachyon:static";
    "tachyon:noglobal";

    assert (
        proto === null || boolToBox(boxIsObjExt(proto)),
        'invalid object prototype'
    );

    // Allocate space for an object
    var obj = alloc_obj();

    // Initialize the prototype object
    set_obj_proto(obj, proto);

    // Initialize the number of properties
    set_obj_numprops(obj, u32(0));

    // Allocate space for a hash table and set the hash table reference
    var hashtbl = alloc_hashtbl(HASH_MAP_INIT_SIZE);
    set_obj_tbl(obj, hashtbl);

    // Return the object reference
    return obj;
}
```

Foreign Function Interface (FFI)

- Can both import and export functions
- Auto-generated wrapper functions
  - Present C functions as JS functions
  - Present JS functions as C functions
  - Automatic type conversions
- For now: C code never touches JS objects
- Minimal number of C functions exposed
  - Memory management, file/console IO, profiling
- Built for speed
  - FFI calls are JITed and statically linked
  - Wrapper functions are inlinable

V8 Extensions

- Memory allocation (for data and code)
  - `allocMemoryBlock`, `freeMemoryBlock`
  - `readFromMemoryBlock`, `writeToMemoryBlock`
  - `execMachineCodeBlock`
- I/O (for source code and REPL)
  - `readFile`, `writeFile`
  - `readConsole`
- Profiling
  - `currentTimeMillis`, `memAllocatedKBs`
  - `pauseV8Profile`, `resumeV8Profile`
  - `shellCommand`
- FFI
  - `getFuncAddr` (to get `puts`, `malloc`, `free`, `runtimeError`, ...)
  - `getBlockAddr`
  - `callTachyonFFI`

Example: Simple JS Function

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

Example: Abstract Syntax Tree (AST)

```
Program                                          ("inc.js"@1.1-1.35:)
|-var= inc [global]                              ("inc.js"@1.1-1.35:)
|-func= inc [global]                             ("inc.js"@1.1-1.35:)
|-block=
|   BlockStatement                               ("inc.js"@1.1-1.35:)
|   |-statements=
|   |   FunctionDeclaration                      ("inc.js"@1.1-1.34:)
|   |   |-id= inc [global]                       ("inc.js"@1.1-1.35:)
|   |   |-funct=
|   |   |   FunctionExpr                         ("inc.js"@1.1-1.34:)
|   |   |   |-param= n                           ("inc.js"@1.14-1.15:)
|   |   |   |-var= n [local]                     ("inc.js"@1.1-1.34:)
|   |   |   |-body=
|   |   |   |   ReturnStatement                  ("inc.js"@1.19-1.32:)
|   |   |   |   |-expr=
|   |   |   |   |   OpExpr                       ("inc.js"@1.26-1.31:)
|   |   |   |   |   |-op= "x + y"
|   |   |   |   |   |-exprs=
|   |   |   |   |   |   Ref                      ("inc.js"@1.26-1.27:)
|   |   |   |   |   |   |-id= n [local]          ("inc.js"@1.1-1.34:)
|   |   |   |   |   |   Literal                  ("inc.js"@1.30-1.31:)
|   |   |   |   |   |   |-value= 1
```

Example: High-Level IR (HIR)

```
box function () []
{
    box function inc(box n) []
    {
        entry:
        box n = arg 2;
        box $t_4 = call <fn "add">, undef, undef, n, box:1;
        ret $t_4;
    }

    entry:
    box $t_3 = call <fn "makeClos">, undef, undef, <fn "inc">, pint:0;
    ref $t_4 = get_ctx;
    pint $t_5 = add_pint pint:0, pint:36;
    box global = load_box $t_4, $t_5;
    box $t_7 = call <fn "putPropVal">, undef, undef, global, "inc", $t_3;
    ret undef;
}
```

Example: Low-Level IR (LIR)

```
box function inc(box n) []
{
    entry:
    box n = arg 2;
    pint $t_4 = and_box_pint n, pint:3;
    bool $t_6 = eq_pint $t_4, pint:0;
    if_bool $t_6 then and_sec else if_false;

    and_sec:
    box $t_16 = add_ovf n, box:1 normal call_res overflow iir_false;

    iir_false:
    ref $t_10 = get_ctx;
    box global_2 = load_box $t_10, pint:36;
    box $t_12 = call <fn "addOverflow">, undef, global_2, n, box:1;
    jump call_res;

    ...
```

Example: Low-Level IR (LIR) contd.

```
    ...

    if_false:
    ref $t_19 = get_ctx;
    box global_3 = load_box $t_19, pint:36;
    box $t_21 = call <fn "addGeneral">, undef, global_3, n, box:1;
    jump call_res;

    call_res:
    box phires = phi [$t_12 iir_false], [$t_21 if_false], [$t_16 and_sec];
    ret phires;
}
```

Example: x86 Machine Code

```
<fn:inc>
movl 4(%ecx),%edi
subl $3,%edi
testl %edi,%edi
je L7828
cmpl $0,%edi
jg L7829
movl $25,%ebp
movl 4(%ecx),%edi
cmpl $0,%edi
cmovlel %ebp,%edx
cmpl $1,%edi
cmovlel %ebp,%ebx
cmpl $2,%edi
cmovlel %ebp,%eax
jmp L7828
L7829:
movl %eax,12(%ecx)
movl %esp,%ebp
subl $1,%edi
cmpl $0,%edi
jle L7828
L7831:
cmpl %esp,%ebp
jl L7830
movl (%ebp),%eax
movl %eax,(%ebp, %edi, 4)
subl $4,%ebp
jmp L7831

L7830:
movl 12(%ecx),%eax
sall $2,%edi
addl %edi,%esp
L7828:
entry:
movl %eax,%ebx
andl $3,%ebx
testl %ebx,%ebx
movl $0,%ebx
cmovzl %esp,%ebx
testl %ebx,%ebx
je if_false
jmp log_and_sec

if_false:
movl %ecx,%ebx
movl 36(%ebx),%ebx
movl <addGeneral_fast>,%edi
movl $25,%edx
movl $4,%esi
movl $4,4(%ecx)
call *%edi
jmp call_res

log_and_sec:
movl %eax,%ebx
addl $4,%ebx
jno ssa_dec
jmp iir_false

ssa_dec:
movl %ebx,%eax
jmp call_res

iir_false:
movl %ecx,%ebx
movl 36(%ebx),%ebx
movl <addOverflow_fast>,%edi
movl $25,%edx
movl $4,%esi
movl $4,4(%ecx)
call *%edi
jmp call_res

call_res:
ret $0
```

Optimistic Optimizations

- Traditional optimizations are conservative
  - Can't prove it, can't do it
  - Dynamic languages offer little static type information
  - Dynamic constructs problematic for analysis
    - `eval`, `load`
  - Often can't prove validity conservatively
- Optimistic optimizations
  - Most JavaScript programs not that dynamic
  - Many optimizations do apply in practice
    - But you can't always prove it conservatively
  - Valid now, presume valid until proven otherwise
    - Innocent until proven guilty (faulty?)

Example: Optimization Issues

```
var zero = 0;

function sum(list) {
    var sum = zero;
    for (var i = 0; i < list.length; ++i)
    {
        var t = list[i];
        sum = sum + t; // Addition or concatenation
    }
    return sum;
}

function f(x) { zero = x; }

print(sum([1,2,3,4,5]));
```

- Don't know type of `list` and its elements
- Type of `zero` could change
- Dynamic type checks needed

Example: Dynamic Checks - Rough Sketch

```
var zero = 0;

function sum(list) {
   var sum = zero;
   for (var i = 0; i < list.length; ++i) {
      var t = list[i];
      if (typeof sum === 'number' && typeof t === 'number')
         sum = numberAdd(sum, t);
      else
         sum = genericAdd(sum, t);
   }
   return sum;
}

function f(x) { zero = x; }

print(sum([1,2,3,4,5]));
```

What Would Tachyon Do (WWTD)?

- A VM can observe global variables types during execution
  - Can assume that these types will not change
  - Compile functions with these assumptions
- A VM can observe function arguments types
  - Can specialize functions based on these
- Types inside of function bodies can be inferred from types of globals and arguments
  - Type propagation, simple dataflow analysis

Example: Guarded Code - Rough Sketch

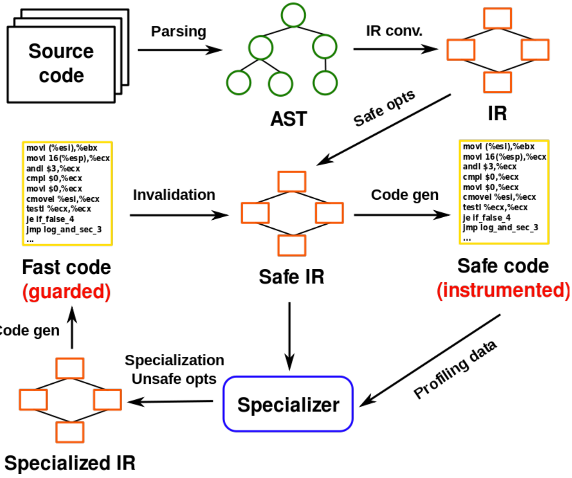
```
var zero = 0;

function sum(list) {
    var sum = zero;
    for (var i = 0; i < list.length; ++i) {
        var t = list[i];
        sum = numberAdd(sum, t);
    }
    return sum;
}

function f(x)
{
    zero = x;
    if (!(zero instanceof Number))
        recompile(sum);
}

print(sum([1,2,3,4,5]));
```

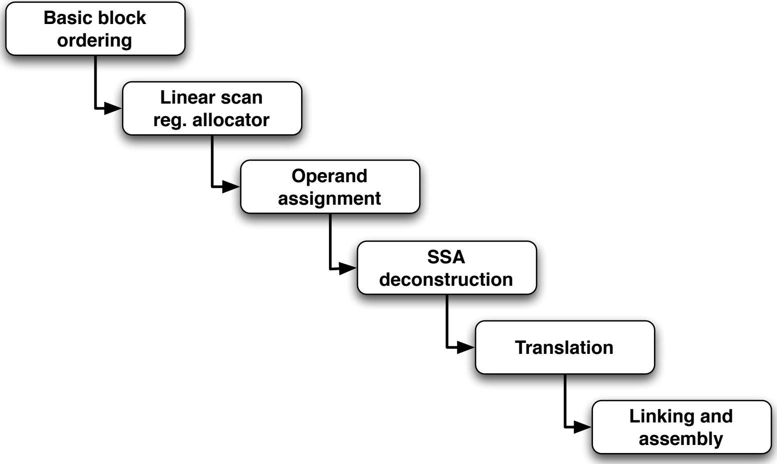
What Would Tachyon Do (WWTD)?



Key Ideas

- Crucial to capture info about run time behavior
  - Tracing JITs do this, but leave many run time checks
- Program needs to be correct at all times
  - Don't need to run the same code at all times
  - Multiple optimized versions correct at different times
- Can make optimistic assumptions that may be invalidated
  - So long as we can repair our mistakes in time
  - Code with broken assumptions must never be executed
  - Ideally, want invalidation to be unlikely

Backend Overview



Register Allocator

- Based on a modified linear scan allocator
- Extended the algorithm to allow hints
  - Required to support intricacies of x86 architecture
- 2 types of hints supported for a given position in the code:
  - Register should be free at that point
  - SSA variable should be assigned to a particular register at that point
- Hints are weak properties and may not be respected, so code generation must enforce them when required

Supported Architectures

- Tachyon uses its own assembler for maximum flexibility
- Retargettable backend currently supports x86 and x86_64
- Assembly code produced by a chain of calls that resemble ASM listings

Assembly framework example

```
this.asm = new x86.Assembler(x86.target.x86);

this.asm.

mov(temp, ctxTemp).
mov($(0), temp).

mov($(argsRegNb), argPtr).
sub(numArgs, argPtr).
cmovl(temp, argPtr).
```

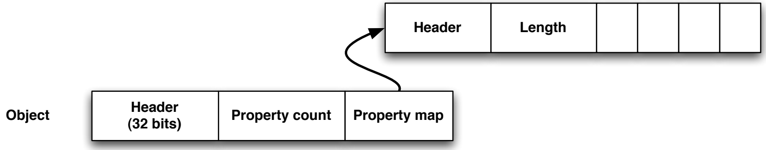
Calling Protocol

- Flexible calling convention through configurable parameters
- Stack pointer and context pointer have dedicated registers
- Return values are passed in a register (currently EAX)
- Up to n first arguments passed using registers (currently, $n = 4$)
- Caller-save protocol
- Callee pops the activation record to support tail call optimisations

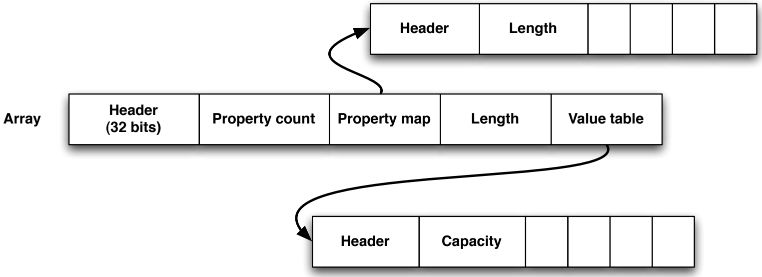
Object Representation

- Flexible object representation through JS layout objects
  - Accessor functions dynamically generated for each layout
  - 3 basic layouts: basic objects, functions and arrays
- Other objects are heap objects but not JS objects
  - context objects, strings, etc.

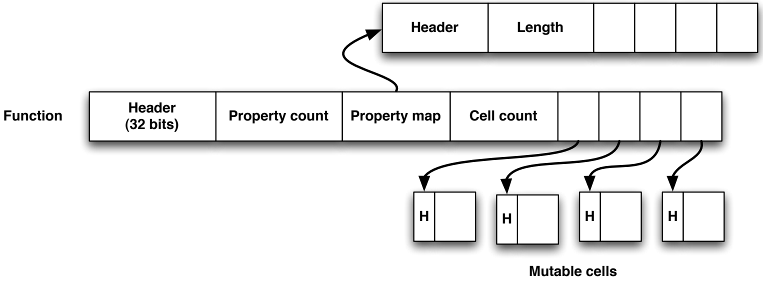
Basic Object Layout



Array Object Layout



Function Object Layout

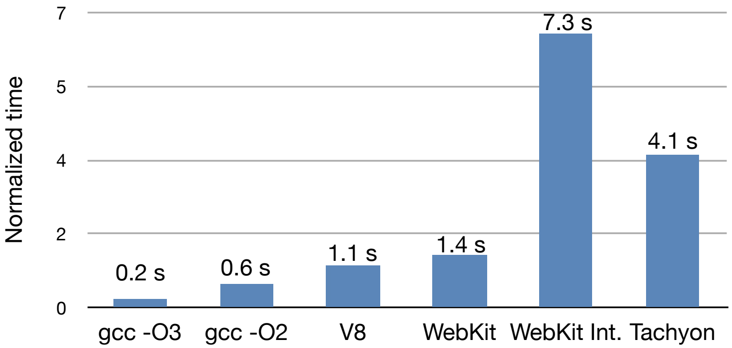


String Object Layout

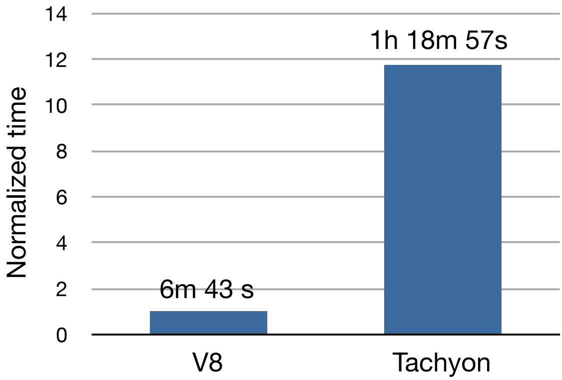
Limitations of JS for Compiler Writing

- Limited precision in number representation
  - e.g. 64-bit precision numbers
- Minimal standard library
  - No standard data structures
- Bitwise operations limited to 32 bits
- Unpredictable allocation behaviour of common operations
- Lack of modules
- No standard I/O operations
- No direct access to memory

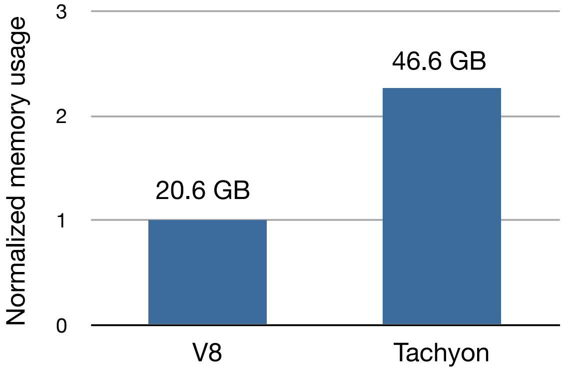
Ballpark Performance Comparison on `fib(38)`



Performance - Tachyon Bootstrap



Allocated Memory - Tachyon Bootstrap



Time Profiler

- Sampling profiler at the level of machine code instructions
    - Currently supports fixed and variable-length intervals
    - Profiling results are currently displayed with the assembly listing
    - Support for more advanced profiling reports is planned (e.g. HTML report, SeeSoft-like tool, etc.)

Allocation Profiler

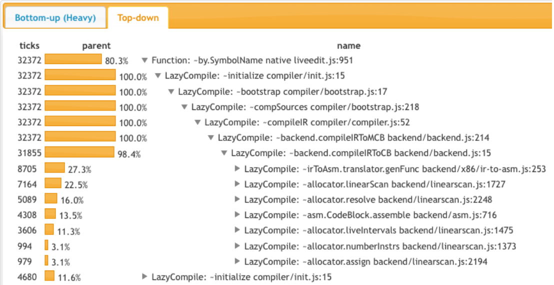
- Allocation profiler currently implemented for the host environment
  - Needs to be ported to the bootstrapped environment
  - Implementation almost entirely in JavaScript
  - Integration into web-based profiler planned

Sample output

```
Types for Assembly (70.6% bytes accounted for):
   Code: 1197 instances, 118112 bytes
   FixedArray: 650 instances, 59880 bytes
   JSFunctionResultCache: 30 instances, 23440 bytes
   ByteArray: 1197 instances, 15192 bytes
   HeapObject: 300 instances, 3600 bytes
   Primitive: 10 instances, 120 bytes
```

Profile Analyzer

- Interactive, web-based tool
- Uses JSON for profiles
- Can easily be extended to support new features



Browser Integration

- Browser integration is necessary to execute real-world JS programs
  - Currently investigating possible avenues for Tachyon in a production-quality browser
  - Looked at Safari, Chrome and Firefox APIs
- Research questions
  - Can we expose Tachyon objects as DOM objects in the browser ?
  - Could it reduce the DOM barrier cost using a pure JS implementation from the browser to the VM ?

Distinguishing Features of Tachyon

- Implementation flexibility
- Self-hosted with dynamic language
- Self-optimizing JIT
- Extensions allow using JS for stdlib (possibly even GC)
- Systematic optimistic optimizations
- Multithreaded compiler

Project Roadmap

1. Bootstrap (March 2011)
   - Simple object representation, stdlib, FFI
   - **very few optimizations** (but careful to allow future opt)
2. Competitive (September 2011)
   - GC + better object representation, full stdlib, FFI
   - Image loader/writer
   - Improve tools: profiler, debugger
   - **speed within factor of 2 of best JS JIT**
3. Gravy (January 2012)
   - Continuations, threads
   - Code-patching, optimistic optimizations
   - Browser integration, real benchmarks
4. Groovy (long term)
   - Persistance, contextualization
   - On-stack replacement

Thanks for listening!

We welcome your questions/comments

Feel free to contact the Tachyon team:
{chevalma,lavoeric,dufour,feeley}@iro.umontreal.ca