MuscalietJS (MCJS):
Rethinking Layered Dynamic Web Runtimes

Behnam Robatmili, Calin Cascaval, Mehrdad Reshadi,
Madhukar N. Kedlaya, Seth Fowler, Vrajesh Bhavsar,
Michael Weber, Ben Hardekopf
March 1, 2014
JavaScript

• One of the most popular language on all platforms
  – Different workloads: pages, servers, gaming, mobile apps

• Dynamic language
  – Types of variables
  – Object fields
  – Inheritance hierarchy
  – Function bindings/definitions
  – Source code (eval), etc.
  – Standard libraries

• Still an evolving language
  – Formally (ECMA 7 after 10 years) and informally changing!
Need for Extensibility and Efficiency

• Extensible and quickly optimize and implement new features and improve compilers and runtimes
  – Useful for developers and researchers
  – Implement new features or execution mode quickly
  – Adapt to variety of workloads

• Traditional JavaScript engines
  – Many years of development by large teams
  – Takes months (or years) to adopt new features or compilers!
MuscalietJS: A Solution!

- Supports full language and browser interfaces
- Supports
  - Variety of workloads
    - Page loads, repetitive code, DOM accesses
  - Large number of optimizations
- Allows quick and efficient development of JavaScript features and advanced optimizations
- Performant (80% of V8 on Sunspider)
- Developed in 4 person years!

github.com/mcjs/mcjs.git
The Story of MCJS

• ZOOMM: A parallel web browser engine for multicore mobile devices [Cascaval et al. PPOPP 2013]

• Needed a JavaScript engine that supports parallelism (dynamic parallel compilation)
  – Did not exist!
  – Adding support to current engines required rewriting most parts of the engine such as garbage collector and compilers

• .NET or JVM provide parallelism and a lot more features that can be used by a high-level engine

• But how exactly?
MuscalietJS Approach

• Delegates managed features and traditional compiler optimizations to the low-level engine
  – Garbage collector
  – Function inlining + register allocation

• A high-level, JavaScript-specific component implements many language-level dynamic optimizations
  – Hidden classes and property lookup
  – Type analysis and profiling
  – Type specialization

• Exploitation of low-level language-agnostic VM APIs to produce efficient code for hot functions
  – .NET Reflection and Method Property
MCJS Engine Architecture

Web Runtime
(Browser DOM Bindings, HTML5 APIs, JS Events, Timers, etc.)

JavaScript Virtual Machine
Parser → IR → JavaScript Optimizations → CIL JIT → Interpreter

Dynamic Runtime
Basic support needed for dynamic languages
(Dynamic Objects, Types, Hidden classes)

Common IL
Hot func CIL & hints

Common Language Runtime
(CLR: Mono/.NET)
Code Gen, Garbage Collection

ARM / x86
MCJS JavaScript Optimizations

- Adaptive function-level execution
  - Interpreter for page load, graph-base IR, different CIL JIT code generator for hot functions
  - Low over-head switching between interpreter and multiple CIL JITed codes
  - Adding new interpreter or codegen in few weeks!

- Fast property lookup
  - Hidden classes and multi-level property caches
  - Speeds up property lookup significantly

- Parallel compilation (1-month project)
  - Different specialized versions of the same functions
MCJS JavaScript Optimizations (Cont.)

• Type analysis
  – Signature-based method dispatch (3-week project)
  – Profile-driven type inference (2-month internship!)

• Deoptimization for .NET (2-month internship!)

• Specialized CIL JIT code generation for hot code using .NET reflection and method attribute
Example: Efficient CIL JIT for Operations

- In JavaScript operators change behavior based on operand types
- A series of overloaded functions, each implementing one particular instance of an operation’s behavior.
  - Mostly automatically created
  - Simplified to generate of many operation in JavaScript

```csharp
public static class Add
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static float Run(bool i0, float i1) {
        return Run((float)Convert.ToNumber.Run(i0), i1); }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static double Run(mdr.DUndefined i0, bool i1) { return double.NaN; }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static int Run(int i0, int i1) { return i0 + i1; }

    ...
}
```
Example: Efficient CIL JIT for Operations (Cont.)

Typed IR

```
+ 
1  
* 
 a  2
```

Operation DB

```
public static int Add::Run(int i0, int i1) { return i0 + i1; }

public static int Mult::Run(int i0, int i1) { return i0 * i1; }
```

CIL Code Generation using Operation DB (reflection)

```
ldc.i4 1 ;push 1
ldloc.1 ;push a
ldc.i4 2 ;push 2
call Int32 Binary.Multiply:Run(Int32,Int32)
call Int32 Binary.Add:Run(Int32,Int32)
```

Mono JIT x86
Aggressive Inlining
Method attribute

```
shl %ecx ;inlined/optimized int multiply
incl %ecx ;inlined/optimized int addition
```

... CIL Stack machine ....
... x86 ....
Proposal for New .NET APIs

• What else other than .NET Reflection and Method Attributes API to fill the gap?

• Data Attributes
  – Array bound check for property access operations
  – Avoiding object zero initialization for JavaScript objects created by MCJS

• CIL Attributes
  – Avoiding IL validation security checks for CIL code emitted by MCJS
  – Allowing on-stack replacement (OSR) between two CIL emitted methods
Experimental Methodology

<table>
<thead>
<tr>
<th>Engine</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhino</td>
<td>Source-to-source translation from JS Java classes + sophisticated static</td>
</tr>
<tr>
<td></td>
<td>analysis (limitation on the input file size)</td>
</tr>
<tr>
<td>IronJS</td>
<td>Source-to-source JS to Dynamic Language Runtime (DLR) syntax trees, dynamic</td>
</tr>
<tr>
<td></td>
<td>callsites to generate type specialized versions of each operation (not the</td>
</tr>
<tr>
<td></td>
<td>full language)</td>
</tr>
<tr>
<td>V8</td>
<td>Inline-cache based fast compiler and optimizing compiler for hot code</td>
</tr>
<tr>
<td>MCJS</td>
<td>Adaptive compilation and dynamic optimization on top of .NET CLR</td>
</tr>
</tbody>
</table>

- All experiments on a 2.80GHz Intel Core i7, 8 GB of RAM, running Ubuntu 11.04.
- Benchmarks
  - Sunspider
  - Web replay, reconstructed from real webpages
  - Normalized against Rhino interpreter (the higher the better)
SunSpider performance

### Speedup relative to Rhino Interpreter

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Rhino_C</th>
<th>Rhino_C+</th>
<th>MCJS_I</th>
<th>MCJS_J</th>
<th>MCJS_J+</th>
<th>IronJS</th>
<th>V8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3d-cube</td>
<td>176.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3d-morph</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3d-raytrace</td>
<td>124.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-binary-trees</td>
<td>160.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-fannkuch</td>
<td>167.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-nbody</td>
<td></td>
<td>176.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitops-3bits-bits-in-byte</td>
<td></td>
<td>167.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitops-bits-in-byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitops-bitwise-and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>controlflow-recursive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypto-aes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypto-md5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypto-sha1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date-format-xparb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>math-cordic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>math-partial-sums</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>176.6</td>
<td>167.8</td>
<td>176.6</td>
<td>167.8</td>
<td>176.6</td>
<td>176.6</td>
<td>176.6</td>
</tr>
</tbody>
</table>

Qualcomm Research
SunSpider performance

Speedup relative to Rhino Interpreter

MCJS_I: interpreter
MCJS_J: JIT with TI
MCJS_J+: Full JIT Type Specialization
Rhino_C+: Compiler+

Rhino_C  Rhino_C+  MCJS_I  MCJS_J  MCJS_J+  IronJS  V8

0  10  20  30  40  50

0  5  10  15  20  25  30  35  40  45  50

bitops-bits-in-byte

Total
Website performance

- Interpreter performs better than Rhino interpreter (3x)
- Slowdowns due to .NET string and regex library and recursive interpreter
  – Lots of rooms for improvement
Conclusions

• A layered JavaScript engine
  – A high-level runtime handling language-specific dynamic optimizations
  – A low-level host virtual machine to provide runtime services and machine-specific optimizations

• Improve performance
  – Layered architecture combined with general-purpose code generation hints passed from the top-level VM to the .NET

• Made available for your experiments and feedback:
  
  *github.com/mcjs/mcjs.git*
Thanks!