BACHILLERATO INTERNACIONAL

Informática

Monografía:

"Análisis y evaluación del rendimiento y optimización de los motores JavaScript modernos"

Colegio San Patricio de Luján
(0508)

Robles, Jeremías
Número de convocatoria: 0508-016

Número de palabras: 4.000

-2013-

Resumen

El principal objetivo de esta monografía es realizar un análisis en profundidad de los métodos de optimización de código y una evaluación del rendimiento de los motores JavaScript más modernos y que estén siendo activamente desarrollados. Sin embargo, no desarrollé en detalle los procesos de interpretación y compilación, las prácticas eficientes de programación y otros aspectos que desviarían el tópico central de la investigación, no sólo con el fin de cumplir con el límite de palabras establecido[1], sino también para mantener un nivel de audiencia general en lo que a conceptos respecta.

En el desarrollo de mi investigación analicé los motores *V8*, *SpiderMonkey* y *JavaScriptCore*. Fueron varios los motivos que me llevaron a elegir estos tres motores en específico de entre la gran variedad de intérpretes JavaScript y sus derivados disponibles actualmente en el mercado. En primer lugar, son los tres más populares debido a su implementación en los navegadores más utilizados[2], por otro lado también son desarrollos de código abierto, lo que me permitió compilar el código por mí mismo y ejecutar los *benchmarks* fuera del entorno del navegador, descartando posibles elementos que afectaran el rendimiento de las pruebas.

A partir de mi investigación llegué a la conclusión de que los motores JavaScript tienen y tendrán un gran impacto en la forma en que desarrollamos y utilizamos aplicaciones, tanto web como de escritorio. Por otro lado, el motor con mejor rendimiento hoy en día es *V8* desarrollado por Google Inc. y utilizado por los navegadores Google Chrome y Opera. La principal diferencia de este motor frente a los otros dos analizados es su metodología, ya que compila todo el código JavaScript en lugar de utilizar un intérprete.

[1] Nota: el recuento de palabras no incluye carátula, encabezados, pies de página, resumen, índice, bibliografía, anexo, títulos, gráficas ni tablas.

[2] "Browser Statistics", w3schools.com - http://w3schools.com/browsers/browsers_stats.asp

Índice

Introducción

Prefacio

Este documento abarcará el análisis del rendimiento y la optimización de los tres principales motores JavaScript actuales: SpiderMonkey, V8 y JavaScriptCore. Mi interés por realizar esta monografía deriva tanto de mi pasión por la tecnología en general (sobre todo en el ámbito de la programación), y debido a que siempre me atañó conocer cómo trabajaban los navegadores web, en especial con la llegada de las páginas con contenidos dinámicos. Por otra parte me gustaba la idea de poder escribir código, que este fuera multiplataforma y que a la vez fuera de rápida ejecución (al igual que Java, pero aún de más fácil acceso).

Cada vez son más y mejores los sitios y aplicaciones web desarrollados a nivel mundial. El principal motivo de esto es debido al avance de las especificaciones de las tecnologías web (HTML, CSS y JavaScript), y consecuentemente de los navegadores. Hasta hace menos de una década, la mayoría, el contenido publicado en Internet era estático. Hoy en día resulta inusual visitar una página de Internet con contenido completamente estático. Incluso muchos de los sitios webs más ordinarios contienen pequeños scripts que miden el tráfico de usuarios que recibe (siendo esto posible gracias a lenguajes como JavaScript).

Es por esto que no podemos ignorar el avance del desarrollo de estas mejoradas e incluso nuevas tecnologías, ya que comprender el impacto que producen en nuestra sociedad y en la forma en que nos comunicamos es indispensable para entender en qué sentido debemos avanzar. Aquí es donde surge mi problema de investigación: ¿qué tan optimizados se encuentran los motores JavaScript actuales al compararlos uno sobre otro?

Conceptos y generalidades

El lenguaje JavaScript

JavaScript es un lenguaje de programación interpretado de alto nivel que fue originalmente implementado como parte de navegadores web con el fin de ejecutar scripts del lado del cliente en lugar del servidor y así poder interactuar con el usuario de una forma mejor y más rápida. Los principales usos de scripts JavaScript en los navegadores web son la comunicación asincrónica entre cliente y servidor para luego alterar el documento cargado sin necesidad de recargar la página entera y la interacción con el usuario desde la validación de campos de un formulario hasta (recientemente) el desarrollo de juegos y aplicaciones.

Motor JavaScript

Un motor JavaScript estándar moderno está compuesto por un *parser* (analizador sintáctico), un intérprete y un compilador. El *parser* analiza el código fuente y construye un *árbol sintáctico* para representar el *flujo del código*. Usualmente comienza por dividir el código en *tokens*, es decir clasificando si una palabra representa una cadena, un número, una palabra clave, un nombre de variable; e interpretando los signos de puntuación (punto y coma, llaves, etc.). A partir de esta lista, la estructura gramática del programa puede ser inferida.

Luego el intérprete utiliza el *árbol sintáctico* generado para ejecutar el código. Sin embargo, las implementaciones en el intérprete pueden variar ampliamente. Actualmente, un intérprete JavaScript

eficiente lleva a cabo diversas optimizaciones antes de ejecutar el código, con el fin de alcanzar la máxima velocidad posible.

Dependiendo del motor, es posible que el intérprete decida, a partir de un conjunto de heurísticas, si debe pasar el código al compilador para que sea traducido y ejecutado en instrucciones de computadora en lugar de interpretarlo. Algunos motores como el V8 de Google incluso ya no incluyen un intérprete, sino que directamente compilan el código entero y realizan optimizaciones. La principal ventaja de utilizar código de máquina para la ejecución de JavaScript es su enorme incremento en la velocidad y rendimiento en comparación con instrucciones virtuales ejecutadas por un intérprete debido a la disminución del tiempo de trabajo.

Por otro lado, un motor JavaScript necesita también poder interactuar con su ambiente, particularmente cuando se encuentra embebido en aplicaciones tales como los navegadores. Para este propósito, el programa embebido puede transferir referencias a objetos desde JavaScript y viceversa. Un claro ejemplo de esto es cuando un script ejecutado en un sitio web manipula el *DOM* (Document Model Object) usando JavaScript: en este caso lo que en realidad sucede es que el motor vincula parte del código con un objeto fuera del entorno JavaScript.

Incluir la interacción del código JavaScript con objetos externos como el DOM en el desarrollo del documento excedería el límite de palabras y desviaría el enfoque de la investigación, por lo que me limitaré a comparar cada motor analizado desde el punto de vista del rendimiento interno.

Compilador JIT (*Just-in-time compiler*)

La principal desventaja de la interpretación de código es su lentitud de ejecución en comparación a ejecutar código compilado. Para resolver este problema, se desarrollaron los compiladores *Just-In-Time* o *JIT Compilers*, los cuales compilan el código fuente a código de máquina antes de que sea ejecutado. Este proceso es ejecutado en mucho menos tiempo de lo que toma la interpretación, pero se requiere de tiempo adicional para compilar cada vez que se inicia el programa. Por este motivo uno de las principales características de muchos de los compiladores JIT es que actúan únicamente cuando una parte del código es ejecutada un determinado número de veces, dejando el resto del trabajo al intérprete.

Recolector de basura (*Garbage collector*)

Cuando un sector en memoria, un objeto creado en JavaScript ya no es accesible puede ser liberado y reasignado a la aplicación, sin que el programador lo haga manualmente. Sin embargo, el cuándo, cómo y si es reasignado o no, dependerá exclusivamente del motor en cuestión. En el caso de V8, por ejemplo, los objetos que ya no son utilizados son liberados de memoria inmediatamente, mientras que en SpiderMonkey se emplea un *Garbage Collector* conservativo que mantiene todos los objetos vigentes hasta que la memoria necesita ser liberada.

Especificación ECMAScript

"ECMAScript es una especificación de lenguaje de programación publicada por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el popular lenguaje JavaScript propuesto como estándar por Netscape Communications. Actualmente está aceptado como el estándar ISO 16262.

ECMAScript define un lenguaje de tipos dinámicos ligeramente inspirado en Java y otros lenguajes del estilo de C. Soporta algunas características de la programación orientada a objetos mediante objetos basados en prototipos y pseudoclases."[3]

Investigación

Motores JavaScript[4]

SpiderMonkey – Mozilla Foundation

SpiderMonkey es el nombre clave otorgado al primer motor JavaScript escrito por Brendan Eich para Netscape Communications, el cual fue liberado como código libre y es ahora mantenido por Mozilla Foundation.

A lo largo de los años este motor fue perfeccionado, dejando de ser únicamente un intérprete para hoy en día incluir múltiples compiladores JIT (anteriormente TraceMonkey, JägerMonkey, y actualmente IonMonkey) y un *Garbage collector*.

A diferencia de sus predecesores, **TraceMonkey** es un *tracing JIT compiler*, el cual compila únicamente los la porción de código ejecutada (en el caso de que existan *ramificaciones*) dentro de los *loops* más usados con la filosofía de que un programa emplea la mayor parte de su tiempo en un *loop*, dejándole el resto del trabajo al interpretador (lo cual resulta en un procesamiento mucho más lento). Mientras que un compilador JIT estándar (metódico) traduce métodos enteros a código de máquina. No obstante, la implementación de un compilador a código nativo al motor resultó en un aumento de rendimiento entre 20 y 40 veces más rápido.

A partir de la versión 4 de Mozilla Firefox se reemplazó al antiguo motor con uno nuevo llamado **JägerMonkey**, que tal como su nombre interno (*MethodJIT*) lo indica, compila métodos enteros con el fin de incrementar el rendimiento en ocasiones donde TraceMonkey no podía generar código nativo estable y requería volver a utilizar el intérprete.

La principal diferencia entre **IonMonkey**, la versión actual del compilador de SpiderMonkey, en relación con sus antecesores, es el hecho de que éste traduce el código JavaScript a un código intermediario (*intermediate representation* o *IR*), el cual es optimizado y finalmente traducido a código de máquina para su ejecución.

OdinMonkey es el nombre de la nueva implementación de la especificación *asm.js* al actual motor de Mozilla. La especificación *asm.js* es un subconjunto de JavaScript diseñado para ser fácilmente compilado a código nativo, debido a que sólo contiene una parte de las funcionalidades de JavaScript y requiere de una sintaxis más estricta, lo que dificulta la comprensión humana pero incrementa el rendimiento de modo que resulta casi nativo. De esta forma los desarrolladores pueden escribir código C/C++ y compilarlo fácilmente a JavaScript utilizando la especificación *asm.js*. Los motores que soporten esta nueva especificación podrán ejecutar el código JavaScript orientado a ésta obteniendo un rendimiento altamente mayor. No obstante, aquellos que no la soporten aún podrán ejecutar el código pero sin aprovechar sus beneficios.

Desde Junio de 2010, SpiderMonkey integra un *Garbage collector* conservativo para administrar la memoria utilizada. Esto significa que el motor no liberará los objetos creados en memoria hasta que no

[3] Wikipedia. "ECMAScript" - http://es.wikipedia.org/wiki/ECMAScript

[4] Con el objetivo de centrar el desarrollo de la investigación en las tecnologías más avanzadas y aprovechar el límite de palabras de manera eficiente, omitiré el motor Chakra desarrollado por Microsoft utilizado en Internet Explorer 10.

sea necesario: tanto ésta se esté agotando o bien la orden sea ejecutada por otro motivo. De esta forma, cuando se es solicitado, el motor realiza una pausa y crea una *Garbage collection* conteniendo todos los objetos innecesarios que pueden ser eliminados de memoria para liberar espacio. Para esto, se identifican todos los objetos *activos*, de modo que los restantes son descartados.

V8 – Google Inc.

V8 es el nombre del motor JavaScript de alto rendimiento y de código libre de Google, escrito en C++ y distribuido junto con su popular navegador, Google Chrome y ahora también en Opera producido por Opera Software. V8 implementa ECMAScript de acuerdo al estándar ECMA-262, 5ta edición. Al igual que otros motores modernos, este compila el código JavaScript en código de máquina para su ejecución y posee un *Garbage collector*. Sin embargo, la forma de tratar a los objetos es sin dudas la principal diferencia entre V8 y los otros motores.

Dado que JavaScript es un lenguaje de programación dinámico, las propiedades de los objetos pueden crearse y eliminarse de forma dinámica. La mayoría de los motores utilizan una estructura de datos de tipo diccionario para almacenar las propiedades de un objeto, lo que conlleva a una búsqueda completa para obtener la ubicación en memoria de la propiedad deseada. Esta metodología hace que el acceso a las propiedades resulte afectando negativamente la velocidad de ejecución en comparación con otros lenguajes como Java que directamente instancian las propiedades al objeto.

Debido a esto, V8 no realiza una búsqueda para obtener las propiedades de un objeto, sino que crea clases ocultas de forma dinámica. Cada vez que una propiedad es creada, el motor crea una nueva clase oculta con las propiedades existentes y sus valores más la nueva propiedad, y finalmente referencia al objeto en cuestión a ésta. De este modo, cada vez que un objeto es creado las clases ocultas pueden ser reutilizadas permitiendo que múltiples objetos con las mismas propiedades compartan una misma clase que almacene los valores de cada objeto en su respectivo *offset*.

Las ventajas de esta metodología son, por un lado, la capacidad de acceder a las propiedades de los objetos sin necesidad de realizar una búsqueda en un diccionario; y por otro, la clásica estructura basada en clases permite una mejor optimización del código.

Otra de las características importantes de V8 es su método de compilación a código de máquina. A diferencia de otros motores, el código JavaScript es compilado desde la primera ejecución, no existen intérpretes ni *bytecode* de por medio. En mi opinión, debería haber algún tipo de control que evite la compilación de partes de código no utilizadas, debido que supone una pérdida de tiempo. Pero a la vez, pienso que la incorporación de un intérprete no sería la solución adecuada, ya que el tiempo ahorrado en la compilación sería utilizado por el *parser*. Tal vez una iniciativa viable sería almacenar en *caché* La optimización se lleva a cabo durante el acceso inicial a una propiedad: V8 determina la clase oculta perteneciente al objeto en cuestión y predice (*cachea*) que la clase será usada para todos los objetos futuros a los que se acceda en la misma parte del código; entonces, si la predicción es correcta, se devuelve el valor de la propiedad desde una única operación. De lo contrario, se ignora la predicción y se busca o crea la clase correspondiente.

Por último, el *Garbage collector* de V8, a diferencia del *GC* conservativo de SpiderMonkey, reclama cada dirección en memoria ocupada por objetos que ya fueron descartados. Con el fin de proporcionar una rápida asignación de memoria a los nuevos objetos, y a la vez evitar la fragmentación de la misma, el motor detiene la ejecución al momento de realizar la limpieza y procesa solamente una parte del objeto *heap*, minimizando el impacto de pausar la aplicación. Y debido a que los objetos son liberados cuando desde el momento que ya no se utilizan, siempre se sabe en donde estarán todos los objetos y punteros en la memoria, evitando confundir objetos con punteros lo que podría resultar en *leaks* de memoria.

JavaScriptCore – WebKit Open Source Project

JavaScriptCore, también conocido anteriormente como KJS, SquirrelFish, SquirrelFish Extreme, -y comerciado por Apple como- Nitro y Nitro Extreme es el nombre del motor JavaScript desarrollado por WebKit y distribuido junto con el navegador Safari[5] de Apple Inc., que implementa ECMAScript de acuerdo a la especificación ECMA-262. Está compuesto por múltiples secciones que incluyen:

- ***Lexer***: responsable del análisis léxico del código JavaScript que es luego traducido en diferentes *tokens.*
- ***Parser***: realiza el análisis sintáctico de los *tokens* producidos por el *lexer* y construye el árbol sintáctico correspondiente.
- ***LLInt***: es la abreviación de *Low Level Interpreter*. Ejecuta el bytecode producido por el *parser*. Es similar a un compilador JIT, pero a diferencia de éste ejecuta instrucciones similares a la del código de máquina. Sin embargo, realiza optimizaciones como *inline caching* presente en muchos de los compiladores JIT actuales.
- ***Baseline JIT*** y ***DFG JIT***: compiladores JIT que se encargan de optimizar iteraciones de código y además realizan una predicción del tipo de variable que se va a utilizar. El primero es invocado únicamente cuando el *LLInt* detectó que una función es llamada 6 veces o un *loop* realizó más de 100 iteraciones aproximadamente, además realiza un perfilamiento del código para mejorar la especulación. El segundo es invocado cuando una función es llamada 60 veces o un *loop* realizó más de 1000 iteraciones aproximadamente, y utiliza los datos perfilados anteriormente para realizar una especulación más agresiva.

Análisis

Análisis teórico

Si bien los motores modernos expuestos en el desarrollo de esta investigación son diferentes, todos ellos realizan una serie de pasos similares para lograr optimizar y ejecutar código JavaScript de la manera más rápida y eficaz posible, lo que en mi opinión nos indica que gran parte del futuro de la aplicaciones tanto web como de escritorio dependerán del rendimiento rápido y eficaz de este lenguaje de alto nivel.

A pesar de que el lenguaje sea por definición interpretado, la mayoría no sólo lo interpreta, sino que también lo compila a código de máquina nativo y lo ejecuta, lo cual provee un enorme aumento en el rendimiento de métodos que son ejecutados en múltiples ocasiones puesto que no es necesario un intérprete que traduzca instrucción por instrucción. Aquí podemos notar una nueva tendencia iniciada por TraceMonkey de incluir compiladores JIT en el proceso de ejecución de los motores JavaScript. En el caso de SpiderMonkey de Mozilla, para lograr esto, el motor lleva la cuenta del número de iteraciones de una parte del código en particular, y si se repite varias veces, pasa a ser compilado a un código intermediario, optimizado, compilado a código nativo y finalmente ejecutado en lugar de ser simplemente interpretado.

Por otro lado, el motor V8 de Google no requiere de manipulaciones de código intermediarias, sino que se compila todo el código JavaScript desde el comienzo, prescindiendo así tanto del intérprete como del *bytecode*. Pienso que esta iniciativa es, dependiendo el contexto, más y a la vez menos eficaz que la utilizada por Mozilla. En primer lugar, debemos tener en cuenta que si el código no contiene partes que son ejecutadas reiteradas veces, en el caso de SpiderMonkey no se realizará optimización alguna, y las instrucciones serán ejecutadas por el intérprete perdiendo la oportunidad de mejorar el rendimiento y la velocidad de ejecución del script. Sin embargo, en el caso de que una página descargue una gran cantidad de código, pero en efecto sólo se use una pequeña parte de este, en el caso de V8 se habría desperdiciado tiempo compilando código que jamás será ejecutado, como es el caso de librerías y *frameworks* como jQuery en la que la mayoría de los desarrolladores no utilizan todas las funciones provistas.

[5] Es importante remarcar que si bien la última versión estable (5.1.7) de Safari publicada por Apple data de del 9 de mayo de 2012, se ha discontinuado su desarrollo en esta plataforma.

A diferencia de estos dos motores, por su parte, JavaScriptCore combina lo mejor de ambos ya que si bien posee un intérprete (LLInt) que se encarga de ejecutar el bytecode producido, éste es de bajo nivel, lo que significa que las funciones son ejecutadas de forma similar a como si estuviesen compiladas en código nativo. Pero además, este motor está integrado con otros dos compiladores JIT (uno principal y otro optimizador) que actúan cuando una porción de código es ejecutada múltiples veces, reduciendo así la carga del intérprete.

Cada navegador tiene sus pros y sus contras, y algunos cuentan (o lo harán en futuras versiones) con características únicas que los diferencian a la hora de compararlos. Por un lado, Mozilla planea implementar la especificación asm.js en su próxima versión del compilador (IonMonkey) de su motor SpiderMonkey, lo que permitirá a los desarrolladores compilar código escrito en C y C++ a JavaScript, beneficiando aún mucho más al rendimiento de los compiladores que la soporten. Google por otra parte crea y asigna clases 'ocultas' a cada objeto para acceder a ellos en lugar de utilizar tablas, lo que permite la reutilización de clases y acaba con la necesidad de tener que realizar una búsqueda secuencial para devolver, por ejemplo, las propiedades de dicho objeto.

Benchmarks

"El *benchmarking* se define como un proceso que consiste en fijar un estándar sobre el cual se puede realizar la comparación, y es una técnica utilizada para medir el rendimiento de un sistema o componente"[6] del mismo, es el resultado de la ejecución de un programa informático o un conjunto de programas en una máquina, con el objetivo de estimar el rendimiento de un elemento concreto, y poder comparar los resultados con máquinas similares".

Generalmente se utilizan los propios navegadores que embeben a los motores JavaScript como medios para ejecutar *benchmarks*. Sin embargo, con el fin de obtener resultados lo más puros posibles, no sólo ejecutaré los principales *benchmarks* JavaScript en los navegadores que utilicen los motores analizados, sino que además yo mismo los compilaré de forma aislada.

Utilizaré la última versión estable publicada de cada navegador al día de la fecha para ejecutar cada prueba tres veces, promediaré los resultados y los expondré en forma de gráfica de barras junto con los valores devueltos por los otros navegadores. Reiniciaré el navegador entre cada prueba para descartar posibles márgenes de errores debido al uso de la memoria.

Todas las pruebas serán ejecutadas en una computadora de escritorio corriendo Microsoft™ Windows™ 7 Service Pack 1 x64 con un procesador Intel® Core™ i5-3540 (3.10GHz) con 4GB de memoria RAM y una tarjeta de video AMD Radeon HD 6800 Series.

El objetivo principal de estas pruebas será determinar cuál de los motores se desempeña mejor ante algo ritmos complejos con el fin de determinar cuál de ellos posee un mejor rendimiento general en cuanto al procesamiento y ejecución de código JavaScript, tanto dentro como fuera del navegador.

Los navegadores utilizados para cada motor JavaScript serán:

Motor JavaScript	Navegador utilizado	Versión del navegador
V8	Google Chrome	29.0.1547.62 m
SpiderMonkey	Mozilla Firefox	24.0
JavaScriptCore	Safari	5.1.7

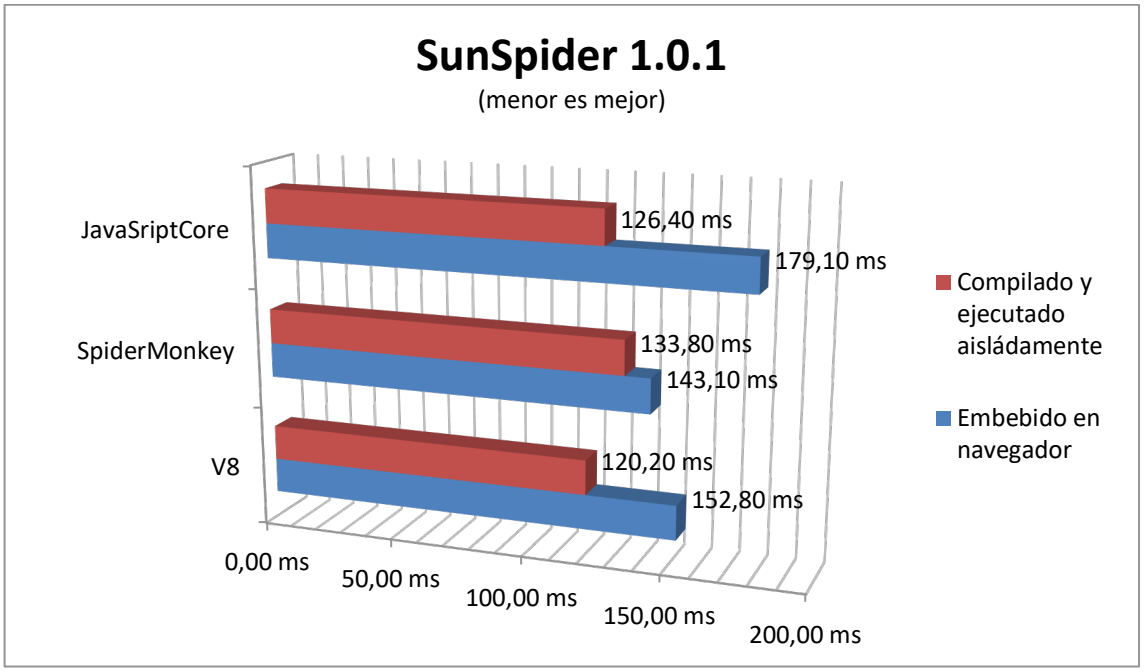
[6] Orozco, Juan. "El benchmarking y su aplicación en las instituciones bancarias", 2010, capítulo 2, página 4 - http://dspace.ups.edu.ec/handle/123456789/495

A pesar de que existen más *benchmarks* para navegadores web que los utilizados en esta monografía, me centraré únicamente en aquellos que fueron programados exclusivamente para medir el rendimiento del motor JavaScript, y no de los componentes del navegador en general (ej.: *DOM*).

Los *benchmarks* utilizados en la prueba con navegadores serán: SunSpider, V8 Benchmark Suite, Kraken, RoboHornet, Octane y Dromaeo. Mientras que en las versiones compiladas por mí, ejecutaré la versión de SunSpider incluida en la última *release* de WebKit.

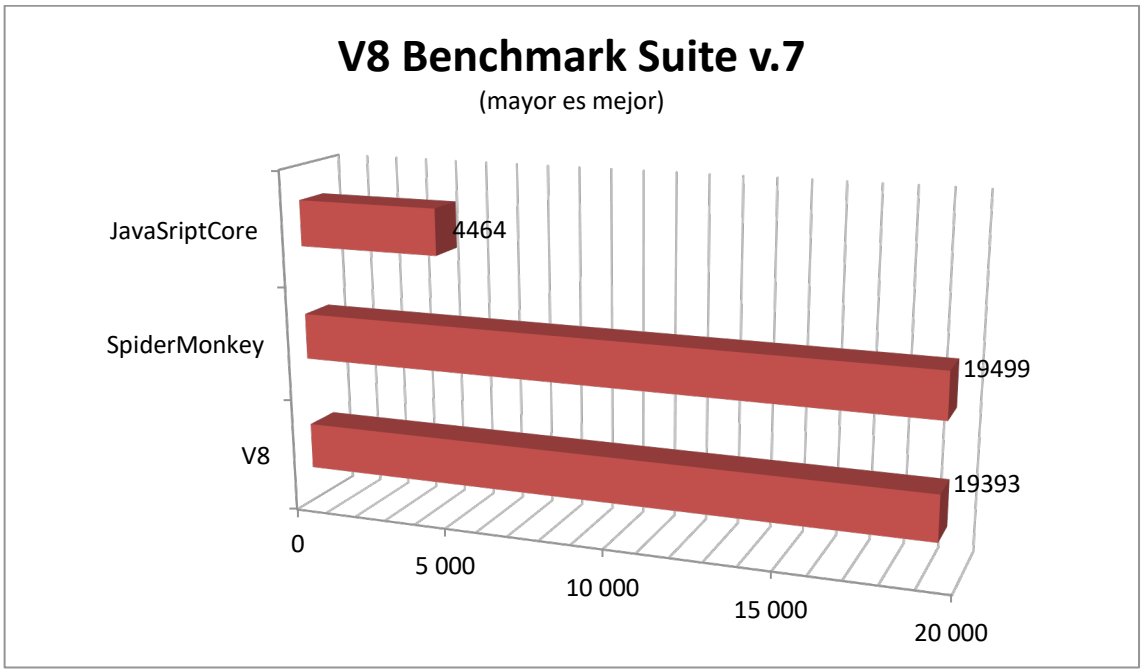
SunSpider 1.0.1

SunSpider es un *benchmark* que evalúa únicamente el rendimiento del motor JavaScript excluyendo el *DOM* y otras APIs del navegador. Esta prueba se caracteriza por ser de tipo *Real World (del mundo real)*, es decir que en lugar de realizar micro optimizaciones, se centra en los principales problemas que los desarrolladores hoy en día resuelven con JavaScript, tales como la manipulación de distintos tipos de variables (*Dates*, *Strings* y expresiones regulares) tanto *array-oriented* como *object-oriented*.



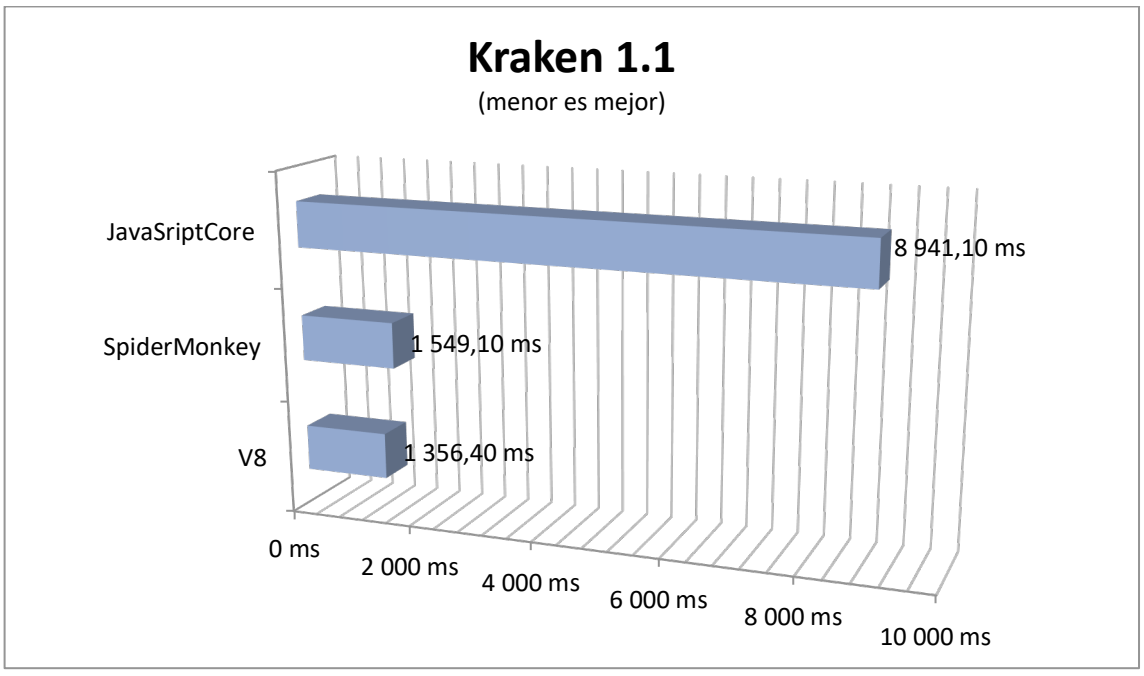
V8 Benchmark Suite v.7

El V8 Benchmark Suite es un conjunto de *benchmarks* para JavaScript desarrollado por Google con el fin de optimizar su motor V8, que trabaja sobre las áreas en las cuales un motor debe tener un rendimiento óptimo para soportar las nuevas aplicaciones web. La puntuación final es determinada a partir de los resultados de ocho algoritmos exigentes.



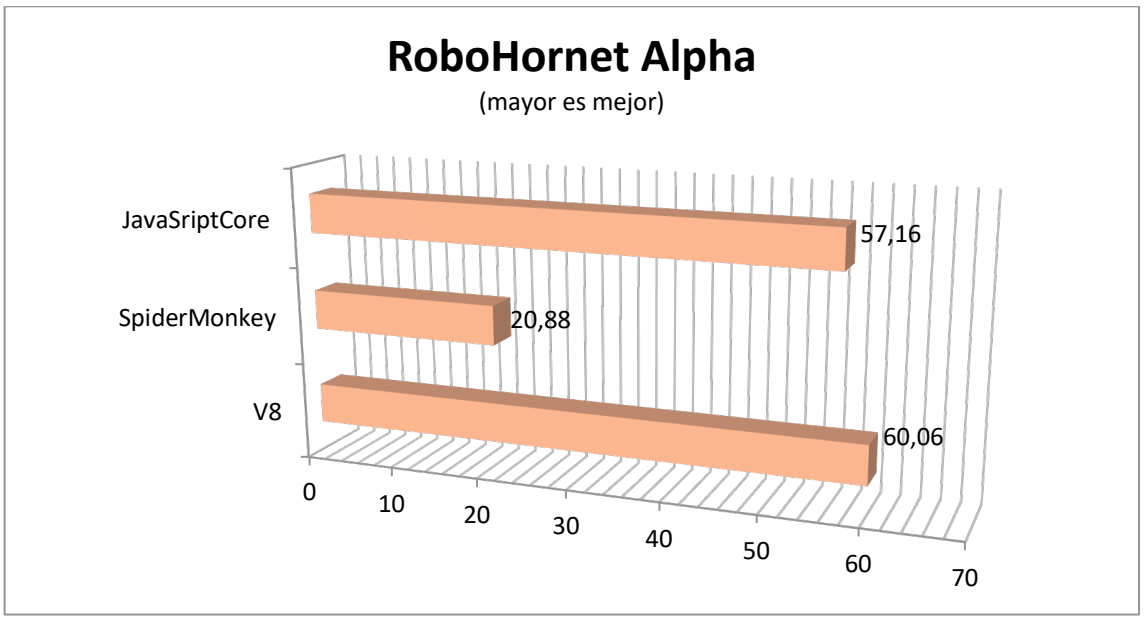
Kraken 1.1

Kraken es el nuevo *benchmark* para navegadores creado por Mozilla que sirve de referencia para analizar el progreso de los navegadores de cara a la ejecución de aplicaciones web y que según ellos, este *benchmark* se centra en cargas de trabajo más realistas.



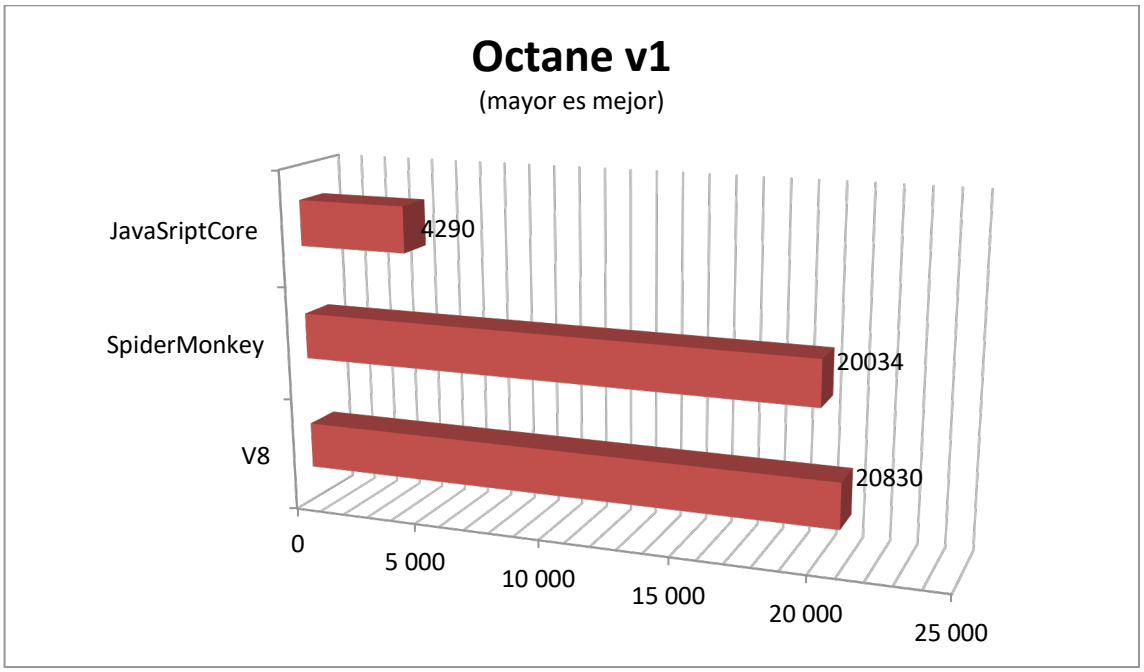
RoboHornet Alpha

RoboHornet es un benchmark de código libre cuyo objetivo según sus desarrolladores es lograr que los creadores de los navegadores solucionen problemas de rendimiento del código JavaScript utilizado comúnmente.



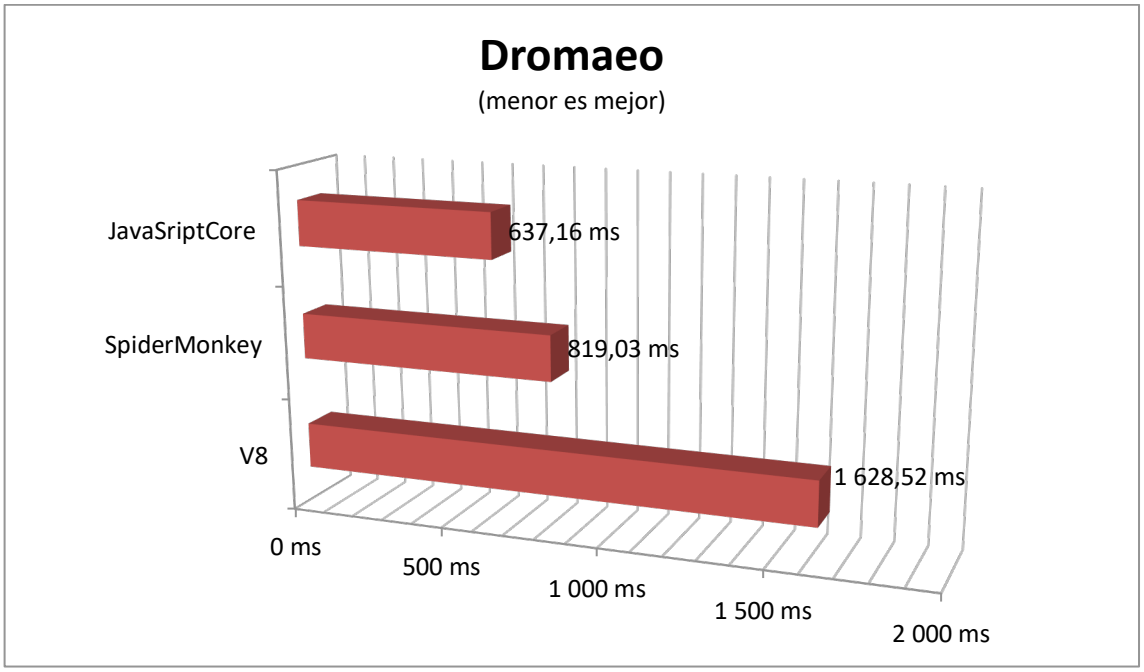
Octane v1

Octane es el sucesor de V8 Benchmark Suite creado por Google para medir el rendimiento del motor JavaScript mediante la ejecución de un número de pruebas representativas de las aplicaciones complejas de hoy. El objetivo de Octane es medir el rendimiento en grandes aplicaciones de la vida real.



Dromaeo

Es una suite de *bechmarks* JavaScript creada por John Resig junto con los fabricantes de los principales navegadores web para llegar a un acuerdo en los puntos de un motor que deben ser evaluados.



Conclusión

En mi análisis acerca de las optimizaciones, luego de comparar tanto las características y funcionalidades de cada motor JavaScript, puedo afirmar que desde su concepción en 1995, tanto el lenguaje como sus implementaciones han avanzado a pasos agigantados y evolucionado para adaptarse a las nuevas tecnologías y formas de comunicación. De hecho, ha sido tal el desarrollo del lenguaje, que en un futuro cercano será factible compararlo en cierto modo con lenguajes de bajo nivel, ya que se podrán portar aplicaciones programadas en C/C++ a otro de alto nivel que inició siendo simplemente uno interpretado, pero que ahora las implementaciones más avanzadas incluyen múltiples compiladores para satisfacer la demanda de velocidad de la comunidad.

Por otro lado, en cuanto a mi evaluación y comparación del rendimiento de cada uno de los motores analizados, puedo observar que en términos generales, el motor V8 desarrollado por Google Chrome ha alcanzado un promedio mayor en los *benchmarks* que ejecutan algoritmos JavaScript complejos. Esto demuestra, al menos a nivel de navegadores, que la iniciativa de compilar el código en su totalidad en lugar de interpretarlo es más eficiente que la utilización de ambos. En segundo lugar se coloca el pionero en este ámbito, SpiderMonkey de Mozilla Foundation. No obstante esta realidad está cada vez más próxima a cambiar debido a la implementación de la nueva especificación *asm.js* en IonMonkey. Por último, y teniendo en cuenta que su portador ya no se encuentra en desarrollo, se posiciona JavaScriptCore. No obstante, observamos que los resultados de SunSpider en la ejecución aislada de los *benchmarks* muestra un

incremento en la velocidad mucho mayor en este motor que en los otros dos, lo cual supone un aumento de rendimiento significativo desde la versión 5.1.7 de Safari.

En mi opinión, tanto la especificación ECMAScript como el desarrollo de los motores JavaScript modernos han demostrado tener un potencial enorme y nos permitirán llegar aún más lejos en la realización de nuestras tareas cotidianas conforme al avance del tiempo sin limitarnos a los navegadores web. No obstante, concluyo con que si bien los niveles de rendimiento alcanzados hoy en día eran inimaginables hace años atrás, aún queda mucho potencial por explotar en este lenguaje, el cual después de todo, es de alto nivel.

Bibliografía

Lista organizada alfabéticamente.

- Brinkmann, Martin. ghacks.net. "SunSpider JavaScript Benchmark 1.0 released" - http://www.ghacks.net/2013/05/02/sunspider-javascript-benchmark-1-0-released/
- Cazzulani, Stefano. Chromium blog. "Octane: the JavaScript benchmark suite for the modern web" - http://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html
- Creative JS. "The race for speed part 1: The JavaScript engine family tree" - http://creativejs.com/2013/06/the-race-for-speed-part-1-the-javascript-engine-family-tree/
- Crockford, Douglas. "JavaScript:
- Github, robohornet, README.md. - https://github.com/robohornet/robohornet/blob/master/README.md
- Google Code. "V8 Benchmark Suite - version 7" - http://v8.googlecode.com/svn/data/benchmarks/v7/run.html
- Google Developers, Chrome V8. "Benchmarks" - https://developers.google.com/v8/benchmarks?hl=es
- Google Developers, Chrome V8. "Design Elements" - https://developers.google.com/v8/design
- Google Developers, Octane - https://developers.google.com/octane/
- Hidayat, Ariya. Sencha blog. "JavaScript Engines: How to Compile Them" - http://www.sencha.com/blog/javascript-engines-how-to-compile-them/
- IBM, "IBM User Guide for Java V7 en z/OS". "El compilador JIT" - http://pic.dhe.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.zos.70.doc%2Fdiag%2Funderstanding%2Fjit_overview.html
- Irish, Paul. "A Browser Benchmark That Has Your Back: RoboHornet" - http://www.paulirish.com/2012/a-browser-benchmark-that-has-your-back-robohornet/
- Lund, Kasper. Chromium blog. "V8 Benchmark Suite Updated" - http://blog.chromium.org/2010/10/v8-benchmark-suite-updated.html
- Mozilla Blog. "JavaScript" - https://blog.mozilla.org/javascript/
- Mozilla Developer Network. "Acerca de JavaScript" - https://developer.mozilla.org/es/docs/JavaScript/Acerca_de_JavaScript
- Mozilla wiki. "Kraken Info" - https://wiki.mozilla.org/Kraken_Info
- Mozilla wiki. Dromaeo - https://wiki.mozilla.org/Dromaeo
- O'Reilly & Associates (2003). "JavaScript: The Definitive Guide, 4th Edition", Capítulo 1: "Introduction to JavaScript" - http://docstore.mik.ua/orelly/webprog/jscript/ch01_01.htm
- Orozco, Juan. "El benchmarking y su aplicación en las instituciones bancarias", 2010, capítulo 2, página 4 - http://dspace.ups.edu.ec/handle/123456789/495
- Resig, John. "JavaScript Engine Speeds" - http://ejohn.org/projects/javascript-engine-speeds/
- StackOverflow. "What does a just-in-time (JIT) compiler do?" - http://stackoverflow.com/questions/95635/what-does-a-just-in-time-jit-compiler-do
- StackOverflow. "What is the use of JVM if JIT is performing bytecode conversion to machine instructions" - http://stackoverflow.com/questions/16439512/what-is-the-use-of-jvm-if-jit-is-performing-bytecode-conversion-to-machine-instr/16440092#16440092
- The World's Most Misunderstood Programming Language" - http://javascript.crockford.com/javascript.html

- WebKit wiki. "JavaScriptCore" - http://trac.webkit.org/wiki/JavaScriptCore
- WebKit.org. "Building WebKit" - http://www.webkit.org/building/build.html
- WebKit.org. "SunSpider 1.0.1 JavaScript Benchmark" - https://www.webkit.org/perf/sunspider/sunspider.html
- Wikipedia. "Browser speed test" - http://en.wikipedia.org/wiki/Browser_speed_test
- Wikipedia. "Garbage collection (computer science)" - http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- Wikipedia. "JavaScript engine" - http://en.wikipedia.org/wiki/JavaScript_engine
- Wikipedia. "Just-in-time compilation" - http://en.wikipedia.org/wiki/Just-in-time_compilation
- Wikipedia. "List of ECMAScript engines" - http://en.wikipedia.org/wiki/List_of_ECMAScript_engines
- Wikipedia. "V8 (JavaScript engine)" - http://en.wikipedia.org/wiki/V8_(JavaScript_engine)
- Wikipedia. "WebKit" - http://en.wikipedia.org/wiki/WebKit
- wingolog. "JavaScriptCore, the WebKit JS implementation" - http://wingolog.org/archives/2011/10/28/javascriptcore-the-webkit-js-implementation

Anexo

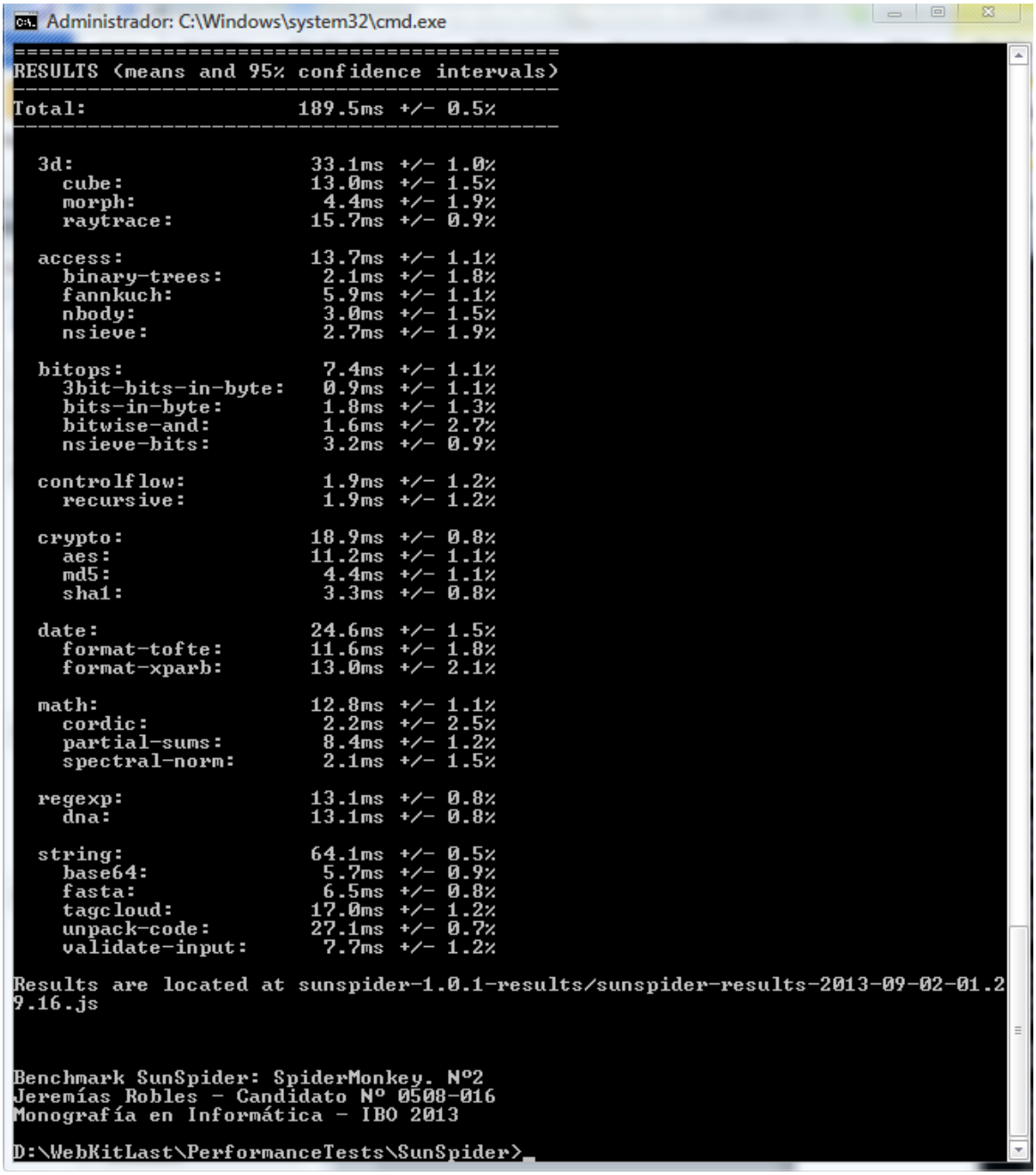


IMAGEN 1: CONSOLA DE COMANDOS DE WINDOWS 7 EJECUTANDO EL BENCHMARK SUNSPIDER EN SPIDERMONKEY MEDIANTE EL COMANDO:

```
01  D:\WebKitLast\PerformanceTests\SunSpider>perl sunspider --
    shell=d:/users/jereees/desktop/mozilla-central/js/src/shell/js & echo. &
    echo. & echo. & echo Benchmark SunSpider: SpiderMonkey. N°2 & echo
    Jeremías Robles - Candidato N° 0508-016 & echo Monografía en Informática
    - IBO 2013
02
03
04
```