Review of Applied Artificial Intelligence Principles in JavaScript Engines

Saint Wesonga

September 29, 2010

Abstract

This article is a review of some Artificial Intelligence concepts and issues in a real world scenario - JavaScript engine optimization. In particular, the focus is on how to best integrate two JavaScript engines, each with its own performance properties, for the best overall performance. I came across this in the Mozilla ticketing system[1] in my endeavors as a part time Mozilla contributor.

1 A Brief History of the Mozilla JavaScript Engines

SpiderMonkey[1], owned and maintained by Mozilla, was the first JavaScript engine ever written. It was written by Brendan Eich, now the CTO of Mozilla Corporation[2]. It's modular design allows it to be embedded in applications such as the Firefox browser and Adobe's Acrobat software. This engine is strictly a JavaScript interpreter.

In the quest for performance, an upgrade to SpiderMonkey called TraceMonkey debutted in 2009 in Firefox 3.5. It included a trace based compiler[3] which works by dynamically discovering loop headers and then recording and compiling all paths through a loop that are executed with sufficient frequency. TraceMonkey can be 3-4 times as fast as SpiderMonkey[4].

Unfortunately, TraceMonkey does not speed up all JavaScript code. The engine may have to abort tracing e.g. if it encounters code that is too branchy (which could lead to an explosion in the number of trace trees). In such cases, it falls back into the original SpiderMonkey interpreter. Consequently, Mozilla began work on JaegerMonkey, a new method JIT for SpiderMonkey

[1]See https://bugzilla.mozilla.org/show_bug.cgi?id=580468

in order to get reliable baseline performance similar to other engines[4] such as Google's V8 and Webkit's JavaScriptCore.

2 Some AI Principles at Work

In order to track the performance of these engines, the Mozilla developers set up a site with graphs comparing performance on the SunSpider[5] and Google V8 benchmarks[6]. The engines are compared to Apple's Nitro engine and Google's V8 engine. Mozilla's next step forward is now integrating the tracing engine and the method JIT since these are complementary compilation techniques.

As of this writing, the combined JaegerMonkey/TraceMonkey engine is about 660ms faster than JaegerMonkey alone on the V8 benchmark. However, running the two engines together on the Sunspider benchmark results in a 10ms slowdown in comparison to running JaegerMonkey alone. It is therefore important to determine when code should run in just one engine, and if so, which one, or in both engines.

In the bug report related to this tuning[8], David Anderson, one of the engine developers, manually instrumented the benchmarks with comments denoting loop bodies and then used a Python script to preprocess them to time individual loops. The script then outputs a chart describing how long each loop took, and which engine mode (combination) was the best for that loop. The initial data is attached to the bug report[9]. Below is a sampling of loop running times for four of the sunspider tests.

sunspider/bitops-3bit-bits-in-byte.js

line	mjit	tjit	m+tjit	best
28	4	1	0	m+tjit
30	4	1	0	m+tjit

sunspider/bitops-bits-in-byte.js

line	mjit	tjit	m+tjit	best
8	18	14	19	tjit
19	32	28	28	tjit
21	32	28	28	tjit

sunspider/bitops-nsieve-bits.js

line	mjit	tjit	m+tjit	best
8	0	0	0	mjit
17	1	1	1	mjit
21	9	12	12	mjit
24	4	9	7	mjit
34	10	13	13	mjit

2.1 Breakdown of Analysis Procedures

The following analysis was performed by the engine developers[2]. Since TraceMonkey performance is affected by the percentage of "hot" paths in the code, a natural question that arises is how this data correlates to the branchiness of the loops. Initially, Anderson was (and probably still is) doing such correlation manually. In order to provide even more useful information for this engine analysis, Anderson created a new script with the ability to selectively enable tracing on any combination of loops on a script. The script then performed a tree search on the various possible engine configurations to find a supposedly optimal benchmark result (the search suggested an optimal benchmark time of 326ms compared to the current time of 358ms in the current engine configuration. The initial searches performed were mostly exhaustive. For the benchmarks on which the exhaustive technique did not work, results were obtained using a random search.

The results collected from this portion of the analysis seemed to suggest to Anderson that there were certain benchmarks that were definitely not worth tracing at all. Some features these had in common were (statically detectable) high nesting levels and calls to the *eval* function. Weaker indicators were large number of *if*s, *call*s, and *return*s.

The focus of these searches had been finding which loops to avoid tracing. Savings of up to 32ms were found on one of the benchmarks. Next, these searches were run to find the most helpful benchmarks to trace. Speedups of up to 6ms were found. This seemed to suggest that it was more important to avoid tracing bad loops than to try tracing better loops (where better means a lot of arithmetic, short loop bodies, and few if statements).

Based on these findings, Mandelin suggested the following heuristic to determine whether loops should be traced:

1. Do not trace loops containing:
   (a) a return out of the loop
   (b) a call to *eval*

[2]Most of this section is a selective transcription of the actual conversation held by the Mozilla developers in their ticketing system.

    (c) more than 2 nested loops (not including the outer loop; at any level)

2. If a loop contains these, probably do not trace it:
    (a) more than 5 conditionals
    (b) maybe-recursive calls (calls to the same name/prop name)
3. If a loop contains these, question tracing it:
    (a) more than 4 function calls; methods seem to hurt tracing more than global function calls
4. If a loop has these characteristics, do trace it:
    (a) short loop body (where 'short' needs to be precisely defined)
    (b) mostly arithmetic (where this needs to be precisely defined - it's mostly about the proportion of arithmetic operations to other kinds)

Further examination of the data raised questions such as whether trace compile time is linear in loop length (which is an issue for benchmarks with over 1000 lines of code, for example). This would determine whether the heuristic would need to account for loop length. It's at this point that Mandelin raises the possibility of (at some point) converting the heuristics into 'features' that can then be plugged into a machine learning algorithm to compute the best formula based on a training set.

Anderson also tried a different profiling technique using runtime *rdtsc* profiling. From a practical point of view, he points out that it's important to find deterministic heuristics first, since timing-dependent results could severely complicate debugging. Other decisions that these engine developers are taking into account include:

1. Whether tracing decisions are best made on a loop-by-loop basis, or on a file-by-file basis.

2. Whethere there is a relationship between loops in a script (does tracing just one and leaving the other harm overall performance)?

3. Are the current benchmarks being used in this training very similar to each other (and thus less representative of real world scripts)?

4. Do the 'dont care' loops (those for which either/both engine is/are fine affect the results, and if so, how?

5. Are there better heuristics for this task? McCloskey suggested the following heuristic:

   Do not trace if and only if any of these conditions hold, otherwise trace:

   (a) there are short loops of the form —for (...; x LT c; ...)—, where —c— is a constant LTE 8?
   (b) there are triply nested loops
   (c) *eval* is called within a loop

2.2 Conclusion

Presented in this paper was an analysis of a real life example of Artificial Intelligence concepts such as searching and heuristics at play in the tuning of the performance of a JavaScript engine. This may well be the first such application of machine learning related techniques as well in the integration of more than one compiler for the same code, each with it's own resultant code performance characteristics.

3 References

1. What is SpiderMonkey?
   http://www.mozilla.org/js/spidermonkey/

2. SpiderMonkey (JavaScript engine)
   http://en.wikipedia.org/wiki/SpiderMonkey_

3. http://www.ics.uci.edu/ franz/Site/pubs-pdf/ICS-TR-07-12.pdf

4. An Overview of TraceMonkey
   http://hacks.mozilla.org/2009/07/tracemonkey-overview/

5. JaegerMonkey
   https://wiki.mozilla.org/JaegerMonkey

6. http://www.arewefastyet.com

7. SunSpider JavaScript Benchmark
   http://www2.webkit.org/perf/sunspider-0.9/sunspider.html

8. V8 Benchmark Suite
   http://v8.googlecode.com/svn/data/benchmarks/v5/run.html
9. JM: Tune Trace JIT Heuristics
   https://bugzilla.mozilla.org/show_bug.cgi?id=580468
10. https://bug580468.bugzilla.mozilla.org/attachment.cgi?id=461848