Hot-Rodding the Browser Engine: Automatic Configuration of JavaScript Compilers

Chris Fawcett, Lars Kotthoff, Holger H. Hoos

University of British Columbia, Department of Computer Science

{fawcettc, larsko, hoos}@cs.ubc.ca

a place of mind

Overview

- Modern optimising compilers offer their users many command-line parameters and options affecting performance.
- Difficult to identify the best parameter configurations for a given application scenario.
- Predefined default configurations are likely not ideal for specific cases.
- We investigate the performance of general-purpose algorithm configuration tools for finding optimised JavaScript compiler parameter configurations.
- We find that there is great potential for these tools to produce configurations with performance significantly better than the defaults.

JavaScript Compilers

- JavaScript (ECMAScript): among the most popular languages.
- Increasing complexity of web applications → advanced compilers, highly-optimised code.
- These compilers are complex, make many performance-affecting parameters available.
- How code should be optimised depends on the application, problem data and execution environment.
- Users and developers want to find configurations optimising performance: runtime, memory, battery usage, and more.
- Identical code can be run millions of times a day, small improvements translate to massive resource savings.
  → Automated approaches to efficiently find these configurations are of great benefit.

Algorithm Configuration

- Many previous approaches to algorithm configuration: random sampling, local search, racing methods.
- More recently, sequential model-based (Bayesian) optimisation, with excellent results.
  → Iteratively construct models of how parameters affect performance.
  → Use models to select new configurations to evaluate.
- Best known model-based configurator (current state of the art) is SMAC.
- SMAC uses random forest models:
  → supports extremely large parameter spaces,
  → mixed discrete and continuous spaces,
  → conditionality and other constraints between parameters.
- Has been successfully used to advance the state of the art in SAT, TSP, MIP, scheduling, AI planning, machine learning hyperparameter optimisation, and more.

Experiment Design

- We considered two state-of-the-art JavaScript compilers, JavaScriptCore (107 parameters) and V8 (173).
- Four state-of-the-art industry benchmark suites: Octane 2.0, Sunspider 1.0.2, Kraken and Ostrich.
- Performed 1 CPU-day runs of SMAC targeting each suite, as well as each component benchmark.
- Focussed on optimising for runtime performance, but our approach is directly usable for other metrics.
- Validated performance of the default and optimised configurations using 100 independent runs on the considered benchmarks and suites.

Important Parameters

We used **ablation analysis** to obtain the parameters most responsible for improved performance. These tended to be related to memory management, garbage collection, and aggressiveness of the code optimisation.

JavaScriptCore, Octane splay

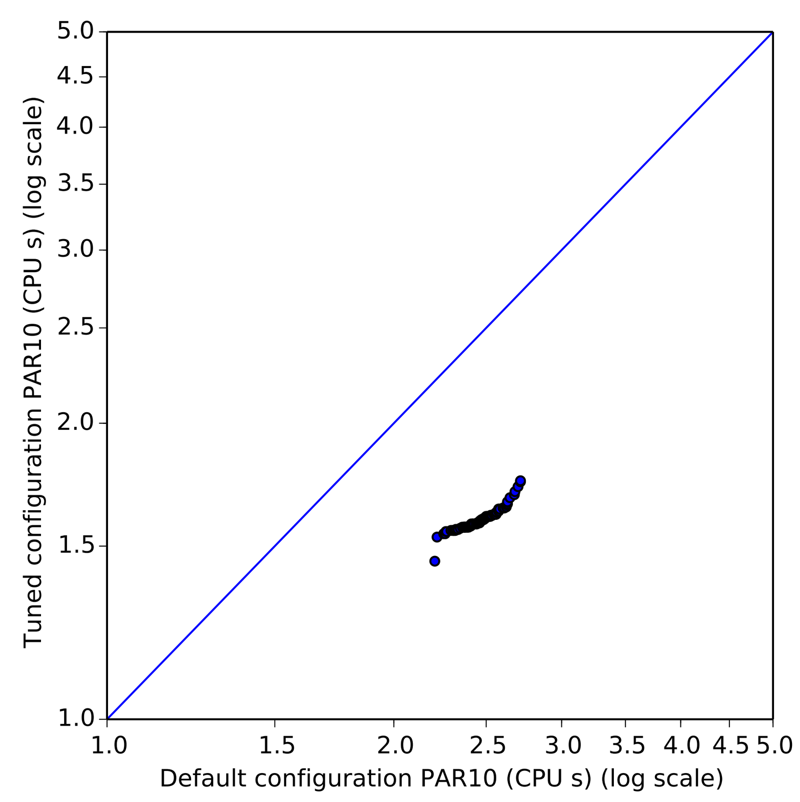
- minCopiedBlockUtilization (47% of improvement)
- numberOfGCMarkers (38%)
- collectionTimerMaxPercentCPU (6%)

JavaScriptCore, Octane PDFjs

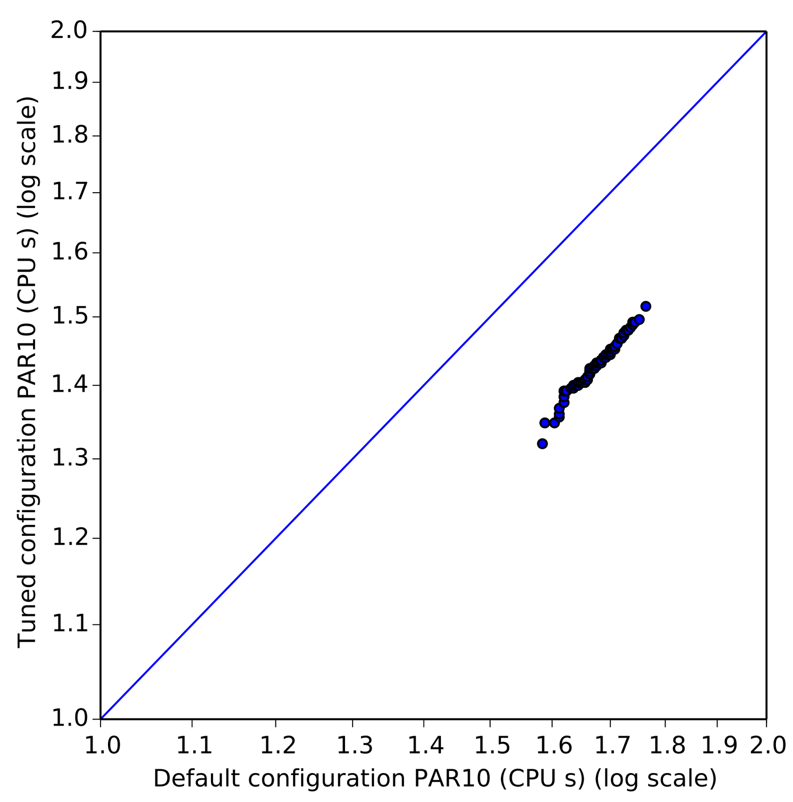
- likelyToTakeSlowCaseMinimumCount (41%)
- numberOfGCMarkers (40%)
- forceDFGCodeBlockLiveness (16%)

Important Results

- General-purpose algorithm configuration tools can obtain configurations with much higher performance than the defaults, for both JavaScriptCore and V8.
- As problem domains get more specific (single benchmarks), optimised configurations obtain higher performance.
- Relatively few parameters are responsible for the increased performance, but differ between considered problems.



(a) JavaScriptCore - Octane - Splay



(b) JavaScriptCore - Octane - PDFjs

Optimised vs. default configurations of JavaScriptCore: Scatter plots of runtime for 100 runs on the Octane benchmark instances corresponding to splay tree manipulation and PDF decoding. We see runtime performance improvements of 35.23% and 14.76% over the default, respectively, on these scenarios.

	running time [PAR10 CPU sec]	
Instance set	JSC default → configured	V8 default → configured
Octane 2.0	1.653 → 1.556 (5.89%)	1.324 → 1.322 (0.18%)
Sunspider 1.0.2	4.546 → 4.010 (11.79%)	3.058 → 3.056 (0.06%)
Kraken	1.214 → 1.205 (0.72%)	0.650 → 0.650 (0.03%)
Ostrich	9.739 → 9.263 (4.88%)	7.109 → 7.062 (0.66%)

Validation results: 100 runs per problem instance for each of our 4 complete-suite configuration scenarios. Results for the JavaScriptCore and V8 default configurations, as well as for the best configuration obtained by SMAC (as identified by training performance). We note that the configurations found for JSC exhibit significant performance improvements over the entire instance set, while those for V8 show only marginal improvement over the defaults.

	running time [PAR10 CPU sec]	
Instance set	JSC default → configured	V8 default → configured
Ostrich sparse-lin.-alg.	11.290 → 11.107 (1.62%)	11.401 → 10.246 (10.13%)
Octane splay	2.467 → 1.598 (35.23%)	1.127 → 1.093 (2.95%)
Octane PDFjs	1.679 → 1.431 (14.76%)	1.654 → 1.645 (0.57%)

Validation results: 100 runs per problem instance for 3 single-instance configuration scenarios, one from our Ostrich set (Sparse Linear Algebra) and two from the Octane set (Splay and PDFjs). We omit two other experiments on Ostrich instances (Graph Traversal and Structured Grid), where neither compiler showed any improvement after configuration. We give results for the JavaScriptCore and V8 default configurations, as well as for the best configuration obtained by SMAC (as identified by training performance).

Future Work

This is an ongoing research project, and we plan to further this work by:

- Applying our approach to LLVM optimisation passes, including C/C++ benchmarks and the Swift compiler.
- Including, and defining the parameter space for, Microsoft's ChakraCore JavaScript engine.
- More thorough investigation of optimised configuration performance on single benchmark instances.
- Further classification and identification of trends for important parameters across scenarios.
- Optimisation for metrics other than runtime, including memory footprint and battery consumption.

Selected Publications

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe.
Opentuner: An extensible framework for program autotuning.
In *PACT'14*, pages 303–316, 2014.

[2] Yuriy Kashnikov, Jean Christophe Beyler, and William Jalby.
Compiler Optimizations: Machine Learning versus O3.
In *Languages and Compilers for Parallel Computing*, pages 32–45, 2013.

[3] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown.
Sequential Model-Based Optimization for General Algorithm Configuration.
In *LION 5*, pages 507–523, 2011.

[4] Chris Fawcett and Holger H. Hoos.
Analysing differences between algorithm configurations through ablation.
Journal of Heuristics, pages 1–28, 2015.

Acknowledgements

The experimental portion of this work has been supported by a Microsoft Azure grant of cloud computing resources.

Further Information

We're sorry to be unable to present this poster in person. We'd much prefer to be in sunny Barcelona than rainy Vancouver! If you would like more information about this ongoing project, please visit our project site:

https://www.cs.ubc.ca/labs/beta/Projects/CompilerParameterOptimisation