Investigating the Evolvability of Web Page Load Time

Brendan Cody-Kenny[1], Umberto Manganiello[2], John Farrelly[2], Adrian Ronayne[2], Eoghan Considine[2], Thomas McGuire[2], and Michael O'Neill[1]

[1]Natural Computing Research and Applications Group (NCRA), Michael Smurfit Graduate Business School, University College Dublin, Ireland.
[2]Fidelity Investments, Ireland.

February 22, 2018

Abstract

Client-side Javascript execution environments (browsers) allow anonymous functions and event-based programming concepts such as callbacks. We investigate whether a mutate-and-test approach can be used to optimise web page load time in these environments. First, we characterise a web page load issue in a benchmark web page and derive performance metrics from page load event traces. We parse Javascript source code to an AST and make changes to method calls which appear in a web page load event trace. We present an operator based solely on code deletion and evaluate an existing "community-contributed" performance optimising code transform. By exploring Javascript code changes and exploiting combinations of non-destructive changes, we can optimise page load time by 41% in our benchmark web page.

Keywords: Javascript, Performance, Web Applications, Genetic Programming, Search-Based Software Engineering

1 Introduction

Performance characteristics vary across browsers where improvements in one version may degrade performance in another [13]. Performance characteristics also change frequently as a Javascript engine is subject to re-design. As a result, performance tuning is a never-ending task. Javascript developers optimise code for Javascript engines while Javascript engine developers optimise for how the engine will likely be used.

While a range of work has looked at mutation-based performance [9, 5] and energy improvement [3], no work we are aware of has inspected source code mutation for page load time in the browser. Related work has looked at web service component selection [4], though this targets components which are a higher level granularity of software unit.

In this paper we investigate the base mechanisms needed for a Genetic Programming (GP) code improvement system: fitness measures and operators. We trace page load for a simple benchmark web app and calculate (i) time, (ii) number of events, and (iii) the largest depth of event chains found in the trace. We inspect two operators, one which deletes statements and expressions containing method names found in the trace, and another which transforms loops to more optimal versions. To validate these mechanisms, we apply each

operator iteratively to all source code locations where they are applicable using a greedy search loop. After applying an operator, if the web page appears as expected, we keep the source code mutation.

Our code deletion operator was able to reduce (i) page load time by 41%, (ii) total events by 30% and (iii) event depth by 26%. While these results are encouraging they are relevant only to our benchmark application, which contains much redundancy by design.

We cover related work in the following section, with Section 3 providing the experimental setup, the target web application is described in Section 4. We then outline the observations arising from the experiments in Section 5 before drawing conclusions and suggesting directions for future work (Section 6).

2 Related Work

Previous work on performance improvement has focused on run-time [9, 5] and energy improvement [3]. Program performance improvements frequently result from code deletion [8], motivating us to initially investigate a deletion operator. Repeatedly applying deletion produces a sub-program similar to one which could be found using program slicing techniques [2, 15]. Designing operators for performance improvement is an open problem, though it has been proposed that new operators may be derived by mining existing code and code generated during GP runs [11]. Program transforms are also written and released by developers in the spirit of making useful transforms reusable [14], adding another potential source of operators.

Exhaustive mutation has been used to find how robust program functionality is to source code change. Mutating small programs with fine-grained operators in a relatively statically typed language such as Java appears to result in relatively low mutational robustness of 30% [5], while larger programs in C++ appear to have high mutational robustness of up to 89% [10, 12] and 68% in more recent work [7]. As mutational robustness varies depending on the software under evolution, we currently have relatively few data points to draw comparisons.

When using search algorithms on large programs it is important to focus operators to reduce search space size [6, 8]. More targeted mutation operators perform program transforms which are highly likely to impact performance while leaving functionality unchanged. For example, multiple different list implementations which have the same interface can be evaluated [1].

3 Experimental Setup

We gather performance measures based on (i) page load time, (ii) number of events during this time as well as (iii) the depth of event chains. We investigate two operators, one written by the authors which deletes code based on method names and one which has been made available as a community contribution. The search loop used simply keeps a code change if the page does not show any error.

3.1 Metrics for Web Page Load

We are mainly interested in improving **page load time** for a web app. We gather traces of the web app loading via chrome browser's devtools functionality[1] (with caching disabled) using a client for NodeJS[2]. Page load traces list events which can be used to build a call graph for the entire page load process. The elapsed time for all events to complete and the depth of call sequences can be calculated from a page trace. We take the time between the first and last event recorded in the browser to give load time.

We sum browser **events** as a pseudo-measure of performance, which is subject to less variation than time-based measures. The number of events in total gives us an idea of how much work is being performed by the browser. We also take a measure of the most deeply nested sequence of event calls, which gives us a rough idea of how interdependent events are. Our intuition is that time can elapse while this call graph is traversed even though more computation could be performed during the same elapsed time if dependent method calls could be rearranged.

We assume we are beginning with an "oracle" web app which is considered **correct**. During page load, we check what elements have been loaded onto the page, and the page is only considered fully loaded when a string appears as part of the Document Object Model (DOM) for the page. We sum the number of pixels by which screenshots differ and use this as our only measure of functional correctness[3] (as this is the only functionality our benchmark provides). We compare two oracle screenshots to get a measure of acceptable variation while still considering the web app correct. We use a multiple of this acceptable pixel variability measure to get a threshold value, above which the page is considered incorrect, that is, too different to be considered the same as the original. Subsequent screenshots taken of web app variants are compared with the original oracle screenshot.

A screenshot captures the final state of the page after it has been loaded. The screenshot does not capture anything about the underlying state of the web page or the structure of the web page Document Object Model (DOM). As a result, using screenshots to measure correctness relaxes the constraints on what is considered correct and frees the evolutionary search process to make changes which affect the underlying structure of the web page HTML. A screenshot only tells us if the page loaded as expected or not, and does not give us any gradient or measure of subsequent page functionality. A screenshot gives no way of telling apart a completely disfunctional page load from a partially functional version of the app. Some functionality is better than none, and Evolutionary algorithms rely on this gradient. In future work the HTML state could additionally be used to give partial functionality gradient.

3.2 Operators

We investigate the use of two operators, one which was developed by the authors of this paper, and the other, which was developed and made freely available by open source contributors. Both operators build an Abstract Syntax Tree (AST) of Javascript source files which are searched, manipulated and written back to a file.

[1] https://developer.chrome.com/devtools
[2] https://github.com/cyrus-and/chrome-remote-interface
[3] http://www.imagemagick.org/Usage/compare/#statistics

3.2.1 Deletion Operator

The simplest way to reduce page load time is to reduce the amount of computation done. To inspect this, we delete portions of the web app source code. Deleting code gives us some indication of the mutational robustness of the software we are targeting. If any portion of the code can be deleted without affecting the correctness of the resulting web page, then we can say that there is some level of redundancy in the code. Claims of robustness can only be made within the context of the operators and test cases used. If the correctness tests (pixel differences between screenshots) do not capture important features or functionality of the web app, then deletion may find performance improvements which turn out to have an undesirable effect on functionality.

A page load trace contains method calls but also which Javascript file the method calls were made from. We use this information to find where methods are called and defined within the Javascript source. We should also note that deletion is distinct from dead code removal. Only method names which appear in the page load trace are considered for deletion. As these methods appear in the trace, we know they have been executed and would not be removed by dead code removal.

3.2.2 Loop Optimiser

The "loop optimiser" operator has made freely available[4] as a "community" contribution to the Babel project[5]. This source code transform is intended to improve the performance of loops. A potential issue with this type of operator is that it is not well tested[6]. While community written program transforms can contain useful improvements, they are not written or tested with the same rigour that might be found in, for example, the transforms performed by a compiler.

Similar community efforts to produce code transforms gives the Evolutionary Computation community a potential source of operators. Though an operator may perform a beneficial transform in most contexts, a developer will still need to test and validate the effect this transform has on their code. A potential impediment to the uptake of these transforms is that we do not have a high level of assurance that these transforms are rigorously semantics preserving. For languages that are dynamically typed, it is more difficult to ensure new transforms preserve semantics. If program transforms are to be crowd-sourced to some extent, we may end up with a scenario where the proliferation of transforms becomes an issue in itself for a developer as time is required to choose, apply and validate transforms. This motivates a Genetic Programming mutate-and-test approach to the application of community operators.

3.3 Search Loop

The search loop iterates through a list of file names from the web app and applies an operator to each one. When applying deletion, the loop iterates through a list of method names and associated files. Method names and files are extracted from a page load trace and used to focus operators on statements and expressions which contain the method name. For each file, each mention of a method name in the AST is found. From each element in the AST which contains the method name, the parent statement or expression is deleted. The app is then loaded over HTTP via the chrome browser which captures a trace of the page load and a

[4] https://github.com/vihanb/babel-plugin-loop-optimizer
[5] https://babeljs.io/
[6] The version we used was specifically marked as experimental

screenshot. Tracing will timeout after twice the time the oracle page load takes (3 seconds). Runtime and event counts are extracted from the trace (used in post-analysis only) and the screenshot is compared with the oracle screenshot to give correctness values. If the page is considered correct, the mutation is kept, otherwise the mutation is discarded and the next operator application is made. To be clear, the performance metrics are not used to guide the search process. The only metric which determines whether a change is kept, is whether the page loads as expected. The performance metrics are not used directly as part of the search loop but are used for post-analysis. When all possible operator applications have been made, we compare the most recent patch, which contains the culmination of all code changes, to the original. By virtue of the operator we inspect, if the page loads correctly after many deletions, we can perform more in-depth performance bench-marking to quantify the performance improvement found.

4 Web Application

In this paper we use an Angular2 web app which is written in typescript and is compiled to Javascript[7] It is this generated Javascript that we target for optimisation. The app is designed to contain redundancy and as an exemplar app with performance issues. The functionality provided by the app is to load a paragraph on the screen and display page load time[8]. Page load takes under two seconds even though the resulting page would be far more effectively delivered as static HTML, instead of being loaded via Angular2 functionality. Any optimisations found will show a clearly measurable improvement in runtime as well as being easy to understand and validate. In this scenario, Angular is far too heavyweight a solution for the resulting web page state. While this appears an easy optimisation target, it is however a challenging benchmark problem for an evolutionary approach and an excellent initial problem on which to validate our approach. The challenge lies in disentangling the extensive sequence of method calls which ultimately produce a static page of text. The oracle web application triggers over 5000 events to a depth of 242 nested method calls. Although Angular is designed to deliver more complex functionality, the increased level of abstraction nonetheless makes it difficult to understand interactions between many layers of sometimes anonymous and interleaved method calls for this simple app. Our hope is that an evolutionary algorithm utilising a range of operators which can reduce the dependencies and redundancies in this benchmark application will also be able to reduce other "real world" web applications in terms of load time.

5 Results

Figure 1 shows 100 repeated measurements of page load time for the original web app and an optimised version of the web app which was derived by the exhaustive application of the **deletion operator**. Each sequence of traces shows roughly the same pattern, though they appear offset by roughly 500ms. We notice that there is a warm-up factor when traces are repeatedly gathered on the same page as initial page load time measures are high in the first couple of traces when compared to the relatively stable measures gathered subsequently.

Figure 2 shows average values for time, number of events and largest event depth from 1000 samples taken after browser warm-up. On average, there is a 41% saving in page load

[7]Full web app code: https://github.com/mlaval/optimize-angular-app
[8]Demo: https://mlaval.github.io/optimize-angular-app/dev

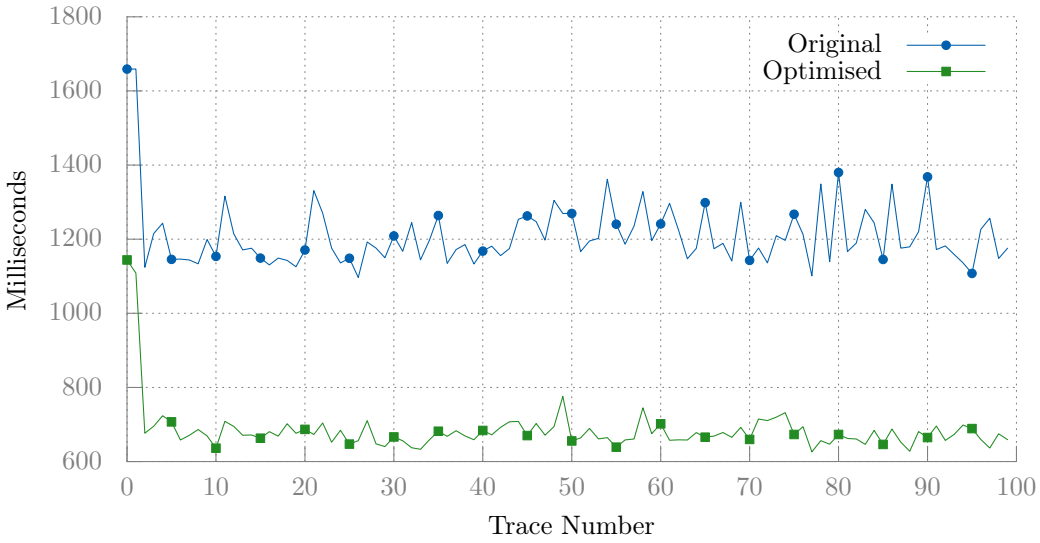
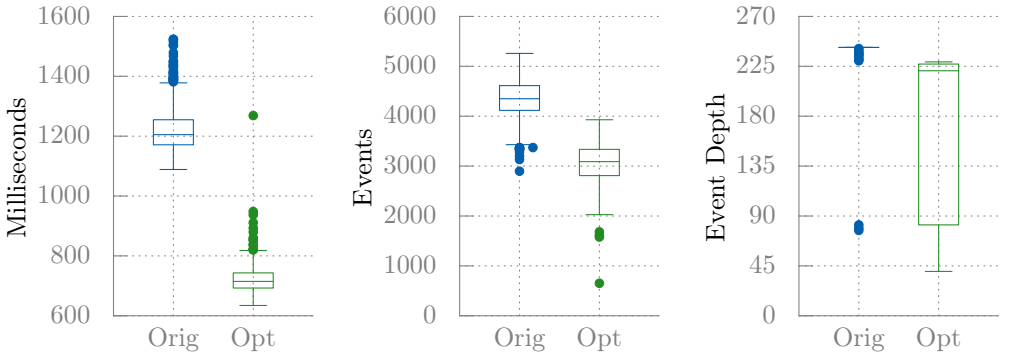


Figure 1: Trace of Original and Optimised web app versions



(a) Time (b) Events (c) Event Depth

Figure 2: Average Metrics (1000 samples) for Original (Orig) and Optimised (Opt) versions of the web app

time. 30% less events occurred and event depth was 26% lower. Variance in measures was roughly the same except for event depth which varied significantly in the optimised version of the app. This may indicate that there are reduced dependencies in this version of the app which leave the browser more freedom in when and how events are triggered. Although a saving of 500ms is a lot in terms of page load time, the mean page load time for the optimised version of the app was still 720ms. Given that the final rendered page is mostly static and the page is loading from the local machine we should be able to achieve far lower load times with additional operators and further refinements of our approach leaving us ample room for future improvement on this benchmark problem.

22% of deletions had no effect on functionality as measured and we can say that the benchmark app we used is relatively robust to deletions. This is an interesting result as we expected removing code which appears in the load trace to be more destructive. 8966 lines were deleted out of a total of 17022 (52%).

Method names found in traces of the app appear 982 times in the app source code. For this particular benchmark app, as method calls were deleted, other method calls would appear in the trace. This is due to the redundancy contained in this benchmark application.

We found one iteration of our search loop, which performs parsing, AST traversal, file writing, page load over HTTPS, trace call tree building, call tree traversal and source code diffing took between 7 and 13 seconds. We feel this is not prohibitive for evolution in the browser, especially given that it is likely to be reduced on further refinement of our approach.

We used the **Loop Optimiser** transform in our search loop but unfortunately no performance improvements were found. During experimentation we also applied the loop optimiser to all Javascript files in the web app and found that it resulted in a page that did not load. The existence and availability of such experimental operators justify a search-based approach which can discover what operators are applicable where.

For GI research, these results reiterate the question as to what is the best way to go about operator design. We could have a wide range of specific operators which are only applicable to certain code patterns (loop optimisation), or have a few very general operators (delete, clone, replace) which are unlikely to leave a program in a fully functional state and only rarely improve performance.

5.0.1 Limitations

The main limitation of this work is that the benchmark app is very simple and contains a lot of redundancy by design. As a result, the findings give little indication as to how generalisable the approach is. When considering more real-world applications of this approach we feel it is unlikely that they will contain the same level of redundancy or be so cleanly structured that deleting lines of code will improve performance while leaving functionality similarly unaffected. Additionally, though comparing screenshots is enough to detect a "correct" page load for our benchmark app, a web app which provides user functionality would need extensive additional testing.

6 Conclusion

We find that improvement of browser-based Javascript using iterative mutation and testing is possible. We reduce a benchmark app to only what code is necessary to provide the desired functionality, saving 41% in runtime. We may compare code changes against those derived through program slicing techniques [15] and investigate if they produce similar performance

improvements. Finding performance improvements further to a minimal code "slice" will likely require more inventive operators. What operators to use is a pertinent open question. We highlight that community contributed code transforms may make ideal candidate operators for search-based approaches. Our findings also give a good base for expanding the work toward more complex "real world" applications with more detailed measures of functionality (e.g. HTML diff or navigation testing) and performance (e.g. user interface jitter).

Acknowledgements This research is based upon works supported by Science Foundation Ireland under grant 13/IA/1850 and 13/RC/2094 which is co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

References

[1] Basios, M., Li, L., Wu, F., Kanthan, L., Barr, E.T.: Optimising Darwinian Data Structures on Google Guava. International Symposium on Search Based Software Engineering (SSBSE) (2017)

[2] Binkley, D., Harman, M.: A Survey of Empirical Results on Program Slicing. Advances in Computers 62 (2004)

[3] Bokhari, M.A., Bruce, B.R., Alexander, B., Wagner, M.: Deep Parameter Optimisation on Android Smartphones for Energy Minimisation. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO). Berlin (2017)

[4] Chang, W.C., Wu, C.S., Chang, C.: Optimizing Dynamic Web Service Component Composition by Using Evolutionary Algorithms. In: International Conference on Web Intelligence, WIC (2005)

[5] Cody-Kenny, B., Galván-López, E., Barrett, S.: locoGP: Improving Performance by Genetic Programming Java Source Code. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO) (2015)

[6] Forrest, S., Nguyen, T., Weimer, W., Goues, C.L.: A Genetic Programming Approach to Automated Software Repair. In: Genetic and Evolutionary Computation Conference (GECCO) (2009)

[7] Langdon, W.: Evolving better RNAfold C source code. bioRxiv (2017)

[8] Langdon, W.B.: Which is Faster: Bowtie2 GP Bowtie> Bowtie2> BWA. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO) (2013)

[9] Langdon, W.B.: Performance of Genetic Programming Optimised Bowtie2 on Genome Comparison and Analytic Testing (GCAT) Benchmarks. BioData mining 8(1) (2015)

[10] Langdon, W.B., Petke, J.: Software is Not Fragile. In: Complex Systems Digital Campus E-conference, (CS-DC). Proceedings in Complexity (2015)

[11] Petke, J.: New Operators for Non-functional Genetic Improvement. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO) (2017)

[12] Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software Mutational Robustness. Genetic Programming and Evolvable Machines (GPEM) (2014)

[13] Selakovic, M., Pradel, M.: Performance issues and optimizations in JavaScript: An empirical study. In: International Conference on Software Engineering (ICSE) (2016)

[14] Various: jscodeshift. https://github.com/facebook/jscodeshift (2017)

[15] Ye, J., Zhang, C., Ma, L., Yu, H., Zhao, J.: Efficient and Precise Dynamic Slicing for Client-Side JavaScript Programs. In: International Conference on Software Analysis, Evolution, and Reengineering (SANER). vol. 1 (2016)