A Study of Performance Variations in the Mozilla Firefox Web Browser

Jan Larres[1] **Alex Potanin**[1] **Yuichi Hirose**[2]

[1] School of Engineering and Computer Science
Email: {larresjan,alex}@ecs.vuw.ac.nz

[2] School of Mathematics, Statistics and Operations Research
Email: hirose@msor.vuw.ac.nz
Victoria University of Wellington, New Zealand

Abstract

In order to evaluate software performance and find regressions, many developers use automated performance tests. However, the test results often contain a certain amount of noise that is not caused by actual performance changes in the programs. They are instead caused by external factors like operating system decisions or unexpected non-determinisms inside the programs. This makes interpreting the test results difficult since results that differ from previous results cannot easily be attributed to either genuine changes or noise. In this paper we present an analysis of a subset of the various factors that are likely to contribute to this noise using the Mozilla Firefox browser as an example. In addition we present a statistical technique for identifying outliers in Mozilla's automatic testing framework. Our results show that a significant amount of noise is caused by memory randomization and other external factors, that there is variance in Firefox internals that does not seem to be correlated with test result variance, and that our suggested statistical forecasting technique can give more reliable detection of genuine performance changes than the one currently in use by Mozilla.

Keywords: performance variance; performance evaluation; automated testing

1 Introduction

Performance is an important aspect of almost every field of computer science, be it development of efficient algorithms, compiler optimizations, or processor speed-ups via ever smaller transistors. This is apparent even in everyday computer usage – no one likes using sluggish programs. But the impact of performance changes can be more far-reaching than that: it can enable novel applications of a program that would not have been possible without significant performance gains.

In the context of browsers this is very visible with the proliferation of so-called "web apps" in recent years. These websites make heavy use of JavaScript to create a user experience similar to local applications, which creates an obvious incentive for browser vendors to optimize their JavaScript execution speed to stay ahead of the competition.

A situation like that poses a problem for developers, though. Speed is not the only important aspect of a browser; features like security, extensibility and support for new web standards are at least as important. But more code can negatively impact the speed of an application: start-up becomes slower due to more data that needs to be loaded, the number of conditional tests increases, and increasingly complex code can make it less than obvious if a simple change might have a serious performance impact due to unforeseen side effects.

Automated tests help with this balance by alerting developers of unintended consequences of their code changes. For example, a new feature might have the unintended consequence of slowing certain operations down, and based on this new information the developers can then decide on how to proceed. However, in order to not create a large number of false positives whose investigation creates more problems than it solves the tests need to be reliable. But even though computers are deterministic at heart, there are several factors that can make higher-level operations non-deterministic enough to have a significant impact on these performance measurements, making the detection of genuine changes very challenging.

1.1 Contributions

This paper tries to determine what the most significant factors are that cause non-determinism and thus variation in the performance measurements, and how they can be reduced as much as possible, with the ultimate goal of being able to distinguish between noise and real changes for new performance test results. Mozilla Firefox is used as a case study since as an Open Source project it can be studied in-depth. This will hopefully significantly improve the value of these measurements and enable developers to concentrate on real regressions instead of wasting time on non-existent ones.

In concrete terms, we present:

- An analysis of factors that are outside of the control, i.e. *external* to the program of interest, and how it impacts the performance variance, with suggestions on how to minimize these factors,
- an analysis of some of the *internal* workings of Firefox in particular and their relationship with performance variance, and
- a statistical technique that would allow automated test analyses to better evaluate whether there has been a genuine change in performance recently, i.e. one that has not been caused by noise.

Table 1: The various performance tests employed by Mozilla

Test name	Test subject	Unit
`a11y`	Accessibility	Milliseconds
`dromaeo_basics`	JavaScript	Runs/second
`dromaeo_css`	JS/CSS manipulation	Runs/second
`dromaeo_dom`	JS/DOM manipulation	Runs/second
`dromaeo_jslib`	JS libraries	Runs/second
`dromaeo_sunspider`	SunSpider benchmark through Dromaeo suite	Runs/second
`dromaeo_v8`	V8 suite benchmark Dromaeo suite	Runs/second
`tdhtml`	JS DOM animation	Milliseconds
`tgfx`	Graphics operations	Milliseconds
`tp_dist`	Page loading	Milliseconds
`tp_dist_shutdown`	Shutdown time after page loading	Milliseconds
`tsspider`	SunSpider benchmark	Milliseconds
`tsvg`	SVG rendering	Milliseconds
`tsvg_opacity`	Transparent SVG rendering	Milliseconds
`ts`	Startup time	Milliseconds
`ts_shutdown`	Shutdown time	Milliseconds
`v8`	V8 benchmark	Milliseconds

More details and complete plots for all of our experiments can be found in the accompanying technical report (Larres et al. 2012).

1.2 Outline

The rest of this paper is organized as follows. Section 2 gives an overview of the problem using an example produced with the official Firefox test framework. Section 3 looks at external factors that can influence the performance variance like multitasking and hard drive access. Section 4 looks at what is happening inside of Firefox while a test is running and how these internal factors might have an effect on performance variance. Section 5 presents a statistical technique that improves on the current capability of detecting genuine performance changes that are not caused by noise. Section 6 gives an overview of related work done in this area. Finally, Section 7 summarizes our results and gives some suggestions for future work.

2 Background

2.1 The Talos Test Suite

The *Talos* test suite is a collection of 17 different tests that evaluate the performance of various aspects of Firefox. A list of those tests is given in Table 1. The purpose of this test suite is to evaluate the performance of a specific Firefox *build*. This is done as part of a process of *Continuous Integration* (Fowler 2006), where newly committed code gets immediately compiled and tested to find problems as early as possible.

The focus of this work is on the Talos performance evaluation part of the continuous integration process. We will also mostly focus on variance in unchanging code and the detection of regressions in order to limit the scope to a manageable degree (O'Callahan 2010).

2.2 An Illustrative Example

Figure 1 illustrates some example data from the `tp_dist` part of the test suite over most of the year 2010. This test loads a number of web pages from the local disk and averages over the rendering times. We

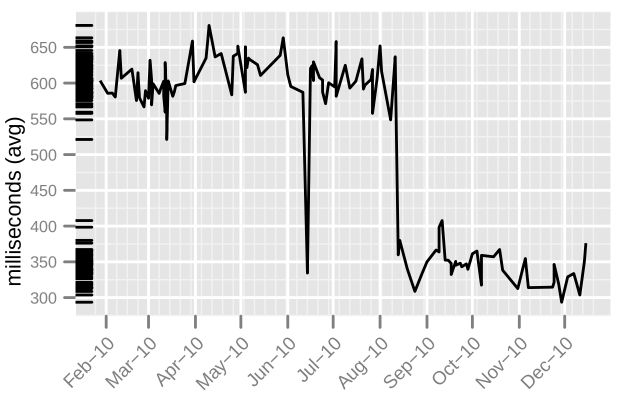


Figure 1: Page load speed `tp_dist` example sequence with data taken from `graphs.mozilla.org`

can see two distinct change patterns in the graph: two big drops in June and August, and seemingly random changes the rest of the time. Since the second drop causes the rest of the results to stay around that level, it suggests a code optimization that led to an overall better performance. The earlier drop of similar magnitude could be a previous application of the optimization that exposed some bugs and was therefore reverted until the bugs were fixed.

Unfortunately we do not have an explanation for the other changes that is as simple as that. But could we apply the same heuristic that lets us explain the big changes – seeing it "sticking out" of the general trend – and use it in a more statistically sound way to try to explain the other results? To some degree, yes.

The exact details of the best way to do this will be explained in Section 5, but let us first have a very simple look at how we could put a number on the variance of a test suite series. We will do this by running a base line series using a standard setup without any special optimizations.

2.3 Statistics Preliminaries

The Talos suite already employs a few techniques that are meant to mitigate the effect of random variance on the test results. One of the most important is that each test is run 5-20 times, depending on the test, and the results are averaged. A statistical optimization that is already being done here is that the maximum result of these repetitions is discarded before the average is calculated. Since in almost all cases this is the first result, which includes the time of the file being fetched from the hard disk, it serves as a simple case of steady-state analysis where only the results using the cache – which has relatively stable access times – are going to be used.

For our statistical significance analyses we will use the common significance level of 0.05.

2.4 The Base Line Test

2.4.1 Experimental Setup

For this and all the following experiments in this paper we used a Dell Optiplex 780 computer with an Intel Core 2 Duo 3.0 GHz processor and 4 GB of RAM running Ubuntu Linux 10.04 with Kernel 2.6.32. To start with we ran the whole test suite 30 times back-to-back as a series using the same executable in an idle GNOME desktop, with 30 being a compromise between reasonable test run times and possible steady-state detection. The only adjustments

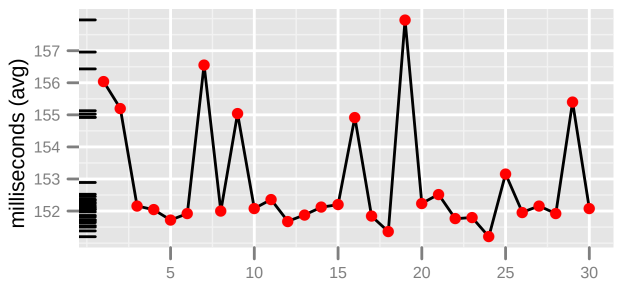


Figure 2: `tp_dist` results of 30 runs

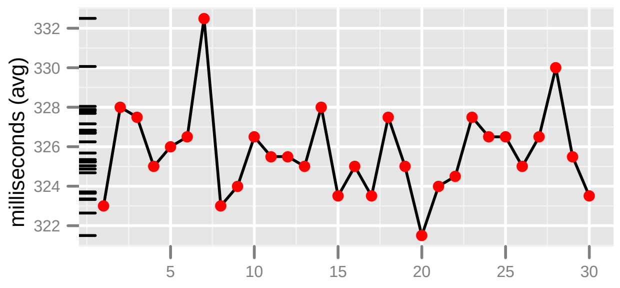


Figure 3: `a11y` results of 30 runs

that we made were two techniques used on the official Talos machines[1], namely replacing the `/dev/random` device with `/dev/urandom` and disabling CPU frequency scaling.

In the following we use the term *run* to refer to a single execution of the whole or part of the Talos test suite and *series* to refer to a sequence of runs, consisting of 30 single runs unless noted otherwise.

2.4.2 Results

Figure 2 shows the results of the `tp_dist` page loading test, and Figure 3 shows the results of the `a11y` accessibility test – both serve as good examples for the complete test suite results. Here we have – as expected – no drastic outliers, but we do still have a non-trivial amount of variance.

Table 2 shows a few properties of the results for the complete test suite. As a typical statistical measure we included the standard deviation and the coefficient of variation (CoV) for easier comparison between different tests. The standard deviation shows us that, indeed, the variation for some of the tests is quite high. The general goal is that we want to be able to detect regressions that are as small as 0.5 % (O'Callahan 2010), so it should be possible to analyse the results in a way so that we can distinguish between genuine changes and noise at this level of precision.

We first look at the maximum difference between all of the values in our series taken as a percentage of the mean, similar to Georges et al. (2007), Mytkowicz et al. (2009) and Alameldeen & Wood (2003). In other words we take the difference between the highest and the lowest value in our series and divide it by the mean. If a new result would increase this value, it would be assumed to not be noise. Looking at the table we can see that almost none of the tests are anywhere near our desired accuracy, so using this method would give us no useful information. If we measure the difference from the mean instead of between the highest and lowest result we can see that the values

[1]`https://wiki.mozilla.org/ReferencePlatforms/Test/FedoraLinux`

Table 2: Results of the base line test

Test name	StdDev	CoV[1]	Max diff (%) Absolute[2]	Max diff (%) To mean[3]
`a11y`	2.23	0.69	3.38	2.08
`dromaeo_basics`	4.41	0.53	2.57	1.62
`dromaeo_css`	11.36	0.30	1.39	0.88
`dromaeo_dom`	1.02	0.41	1.99	1.14
`dromaeo_jslib`	0.53	0.30	1.19	0.60
`dromaeo_sunspider`	5.65	0.54	2.09	1.16
`dromaeo_v8`	2.02	0.86	3.03	1.77
`tdhtml`	0.94	0.33	1.31	0.73
`tgfx`	0.80	5.68	25.60	18.88
`tp_dist`	1.77	1.16	4.42	3.30
`tp_dist_shutdown`	27.09	5.14	16.51	8.72
`ts`	2.27	0.59	2.45	1.66
`ts_shutdown`	7.28	2.00	6.88	3.44
`tsspider`	0.11	1.15	4.04	2.57
`tsvg`	1.43	0.04	0.17	0.10
`tsvg_opacity`	0.62	0.74	3.56	2.02
`v8`	0.11	1.42	4.31	3.59

[1]Coefficient of variation: $\frac{StdDev}{mean}$
[2]Difference between highest and lowest values: $(highest - lowest)/mean * 100$
[3]$max(highest - mean, mean - lowest)/mean * 100$

obviously do look better, but they are still too far away from being actually useful. An additional problem with these techniques is that they have problems with significant genuine changes in the performance like the ones in Figure 1, which are usually much larger than the variance caused by noise.

Section 5 will pursue more sophisticated methods to try to address these concerns. However, even with better statistical methods it will be challenging to reach our goal – the noise is simply too much. Therefore in the next two sections we will first have a look at the physical causes for the noise and try to reduce the noise itself as much as possible before we continue with our statistical analysis.

An important thing to note here is that it is clearly impossible to account for all possible environments that an application may be run in, but that even an artificial environment like ours should still be effective in uncovering the most common issues.

3 External Factors: Hardware, multitasking and other issues

3.1 Overview of External Factors

3.1.1 Multitasking

Multitasking allows several programs to be executed nearly simultaneously, and the kernel tries to schedule them in a way so that the reality of them actually running sequentially (at least on one CPU) is hidden from the user. The consequence of this is that the more programs are running, the less CPU time is available for each one. So the amount of work that can be achieved by any one program in a given amount of real (wall clock) time depends on how many other programs are running. This means that care should be takes as to which programs are active during tests, and also that wall clock time is not very useful for precise measurements. The actual CPU time is of more interest to us. In addition the scheduling may differ from one run to the next, potentially leading to more variance.

3.1.2 Multi-processor systems

In recent years systems with more than one processor, or at least more than one processor core, have become commonplace. This has both good and bad effects on our testing scenario. The upside of it is that processes that use kernel-level threads (as Firefox does) can now be split onto different processors, with in the extreme case only one process or thread running exclusively on one CPU. This prevents interference from other processes as described above. "Spreading out" a process in this way is possible since typical multi-processor desktop systems normally use a shared-memory architecture. This allows threads, which all share the same address space, to run on different processors. The only thing that will not get shared in this case is CPU-local caches – which creates a problem for us if a thread gets moved to a different processor, requiring the data to be fetched from the main memory again. So if the operating systems is trying to balance processes and threads globally and thus moves threads from our Firefox process around this could potentially lead to additional variance.

3.1.3 Address-space layout randomization

Address-space layout randomization (ASLR) (Shacham et al. 2004) is a technique to prevent exploiting buffer overflows by randomizing the address-space layout of a program for each run. This way an attacker cannot know in advance what data structures will lie at the addresses after a specific buffer, making overwriting them with data that facilitates an attack much harder.

Unfortunately, for our purposes this normally very useful technique can do more harm than good. For example, the randomization can lead to data structures being aligned differently in memory during different executions of the same program, introducing variance as observed by Mytkowicz et al. (2009) and Gu et al. (2004).

Additionally, in *Non-Uniform Memory Access* (NUMA) architectures the available memory is divided up and directly attached to the processors, with the possibility of accessing another processor's memory through an interconnect. This decreases the time it takes a processor to access its own memory, but increases the time to the rest of the memory. So depending on where the requested memory region is located the access time can vary. In addition the randomization makes prefetching virtually impossible, increasing page faults and cache misses (Drepper 2007).

3.1.4 Hard disk access

Running Firefox with the Talos test suite involves accessing the hard disk at two important points: when loading the program and the files needed for the tests, and when writing the results to log files. Hard disk access is however both significantly slower than RAM access and much more prone to variance. This is mainly for two reasons: (1) hard disks have to be accessed sequentially, which makes the actual position of data on them much more important than for random-access memory and can lead to significant seek times, and (2) hard drives can be put into a suspended mode that they then have to be woken up from, which can take up to several seconds.

3.1.5 Other factors

Other factors that can play a role are the UNIX environment size and linking order of the program as investigated by Mytkowicz et al. (2009). In our case we worked on the same executables using the same environment and so those effects have not been investigated further.

3.2 Experimental setup

Our experimental setup was designed to mitigate the effect of the issues mentioned in the previous section on the performance variance. The goal was to evaluate how much of the variance observed in the performance tests was actually caused by those external factors as compared to internal ones.

The following list details the way the setup of our test machine was changed for our experiments.

- Every process that was not absolutely needed, including network, was terminated.
- Address-space randomization was disabled in the kernel.
- The Firefox process was exclusively bound to one of our two CPUs, and all other processes to the second one.
- The test suite and the Firefox binary were copied to a RAM disk and run from there. The results and log files were also written to the RAM disk.

Using this setup we ran a test series again and compared the results with our previous results from Section 2.4.2. In our first experiment we tested all of these changes at the same time instead of each individually to see how big the cumulative effect is.

3.3 Results

A comparison of the results of our initial tests and the external optimization approach are shown in Table 3. Overall the results show a clear improvement, most of the performance differences have been significantly reduced. For example, the maximum difference to the mean for the `a11y` test went down from 2.08 % to 0.46 % and for `tsspider` it went down from 2.57 % to 1.34 %.

In order to give a better visual impression of how the results differ Figure 4 shows a violin plot of some of their density functions, normalized to the percentage of their means, with red dots indicating outliers, the white bar the inter-quartile range similar to boxplots and the green dot the median.

Looking at the plots we can see that in the cases of for example `tgfx` and `tp_dist` the modifications got rid of all the extreme outliers. The curious shape of the `v8` plot means that all of the results from the test had the same value, our ideal outcome for all of the tests. Also even though the result table indicates that the max diff metric for `ts` and `tsvg_opacity` increased, the plots show that this is caused by a few extreme outliers and that the rest of the results seem to have gotten better.

3.3.1 The Levene Test

In order to test whether the perceived differences in variance between our setups are actually statistically significant, we made use of the *Levene test for the equality of variances* (Levene 1960, Brown & Forsythe 1974). This test determines whether the null hypothesis of the variances being the same can be rejected or not – similar to the ANOVA test which does the same thing for means. This test is robust against non-normality of the distributions, so even though not all

Table 3: Results after all external optimizations

	StdDev		CoV		Max diff (%)		
Test name	nomod	cumul	nomod	cumul	nomod	cumul	Levene p-value
a11y	2.23	0.54	0.69	0.17	2.08	0.46	< 0.001***
dromaeo_basics	4.41	2.39	0.53	0.29	1.62	1.01	0.028*
dromaeo_css	11.36	7.95	0.30	0.21	0.88	0.46	0.314
dromaeo_dom	1.02	1.00	0.41	0.40	1.14	0.74	0.562
dromaeo_jslib	0.53	0.44	0.30	0.25	0.60	0.79	0.280
dromaeo_sunspider	5.65	3.77	0.54	0.36	1.16	0.74	0.086
dromaeo_v8	2.02	1.20	0.86	0.52	1.77	0.81	0.075
tdhtml	0.94	0.30	0.33	0.10	0.73	0.39	< 0.001***
tgfx	0.80	0.14	5.68	1.37	18.88	2.93	< 0.001***
tp_dist	1.77	0.19	1.16	0.14	3.30	0.35	0.002**
tp_dist_shutdown	27.09	8.59	5.14	1.75	8.72	5.41	< 0.001***
ts	2.27	2.46	0.59	0.74	1.66	3.26	0.282
ts_shutdown	7.28	3.75	2.00	1.19	3.44	2.89	< 0.001***
tsspider	0.11	0.05	1.15	0.64	2.57	1.34	< 0.001***
tsvg	1.43	0.68	0.04	0.02	0.10	0.05	0.006**
tsvg_opacity	0.62	1.11	0.74	1.35	2.02	6.82	0.639
v8	0.11	0.00	1.42	0.00	3.59	0.00	0.008**

nomod: unmodified setup; cumul: cumulative modifications; * $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

of the tests follow a normal distribution the test will still be valid.

Table 3 shows the resulting p-value after applying the Levene test to all of our test results. The results confirm our initial observations: 10 out of 17 tests have a very significant difference, except for most of the dromaeo tests and the ts and tsvg_opacity tests. The dromaeo tests are especially interesting in that most of them are a good way away from a statistically significant difference, and even the one test that does have one is less significant than all the other positive tests. It seems as if the framework used in those tests is less susceptible to external influences than the other, stand-alone tests.

3.4 Isolated Parameter Tests

In order to determine which of our modifications had the most effect on the tests and whether maybe some modifications have a larger impact on their own we also created four setups where only one of our modifications was in use: (1) disabling all unnecessary processes (plain), (2) disabling address-space randomization (norand), (3) exclusive CPU use (exclcpu) and (4) usage of a RAM disk (ramfs).

Table 4 shows the results of comparing the isolated parameters to the unmodified version using the Levene test. We can see that the modification that led to the highest number of significant differences is the deactivation of memory randomization. Especially in the v8 test it was the only modification that had any effect at all – it was solely responsible for the test always resulting in the same value. Equally interesting is that this modification also causes two of the dromaeo tests to become significant that were not in the cumulative case, dromaeo_jslib and dromaeo_sunspider. That suggest that the other modifications seem to "muddle" the effect somehow. Also, in the dromaeo_basics case the disabled memory randomization is the only modification that got rid of all the outliers. Interesting to note is that in the tgfx and tp_dist cases all of the modifications have an influence on the outliers.

3.5 Conclusions

Our modified test setup was a definite improvement on the default state without any modifications. Even though the results did not quite match our goals, they still signified a step in the right direction. Based on that we can safely assume that part of the originally observed variation is caused by the external factors investigated in this section.

Table 4: Levene p-values for isolated modifications, compared to the unmodified setup

Test	plain	norand	exclcpu	ramfs
a11y	0.141	0.831	0.072	0.419
dromaeo_basics	0.617	0.001**	0.199	0.984
dromaeo_css	0.357	0.156	0.926	0.347
dromaeo_dom	0.226	0.112	0.921	0.316
dromaeo_jslib	0.316	0.020*	0.069	0.212
dromaeo_sunspider	0.915	0.028*	0.401	0.743
dromaeo_v8	0.205	0.443	0.995	0.555
tdhtml	0.626	0.983	0.168	0.248
tgfx	0.018*	< 0.001***	0.005**	0.002**
tp_dist	0.006**	0.041*	0.039*	0.038*
tp_dist_shutdown	0.316	0.213	0.031*	0.697
ts	0.086	0.433	0.291	0.296
ts_shutdown	0.080	0.149	0.002**	0.786
tsspider	0.315	< 0.001***	0.004**	0.001**
tsvg	0.893	0.157	0.951	0.679
tsvg_opacity	0.127	< 0.001***	0.262	0.698
v8	0.851	0.008**	0.550	0.857

* $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

Even with the significant improvements from this section the results do not quite match our expectations, unfortunately: only 6 of the 17 tests have a maximum difference of less than 0.5 %. This shows that there are other factors to consider that we do not yet have accounted for.

4 Internal factors: CPU Time, Threads and Events

After dealing with external influences in the last section we will now look at factors that involve the internals of Firefox, specifically, as the title indicates, the time the Firefox process actually runs and the threads and events that are used by it. This involves both investigating how these factors are handled internally and modifying the source code of Firefox and the test suite in an attempt to reduce the variance created by them. Due to space constraints the experiments in this section are only presented in summarised form here. The complete results are available in the technical report (Larres et al. 2012).

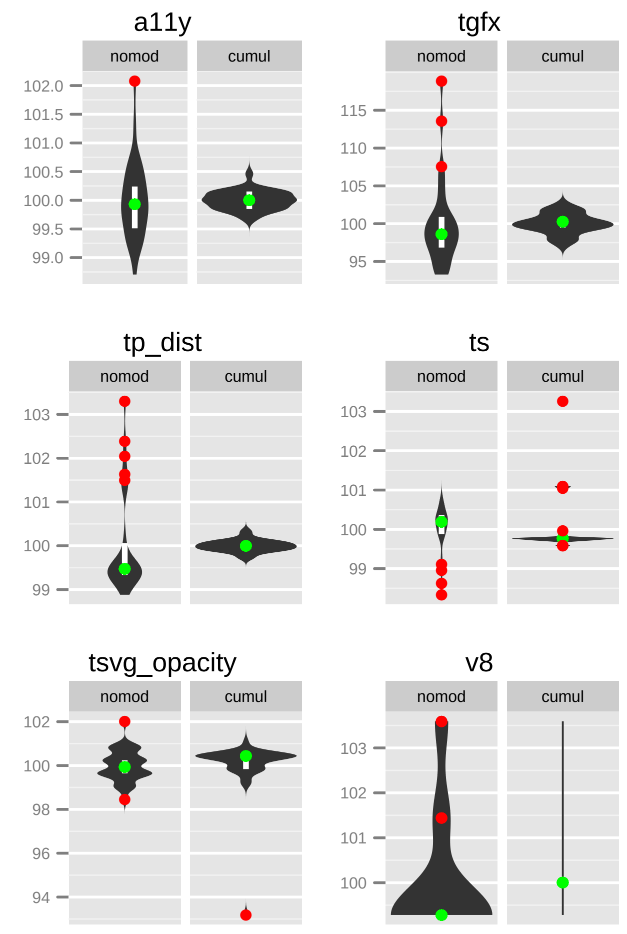


Figure 4: Some of the tests after external optimizations, displayed as the percentage of their mean

4.1 CPU Time

As already mentioned in Section 3.1.1, wall clock time is not necessarily the best way to measure program performance since it will be influenced by other factors of the whole system like concurrently running processes. Since we are running Firefox on an exclusive CPU there is less direct influence by other processes, but context switch time could still matter. We therefore modified Firefox and the test suite to record the CPU time at the start end end of every test run. This was done using the `clock_gettime()` system call for the CLOCK_PROCESS_CPUTIME_ID timer.

Unfortunately only a few tests make direct use of the time that the Talos framework gathers in this manner, namely `tgfx`, `tp_dist`, `tsvg`, and `tsvg_opacity`; most tests, especially the JavaScript tests, do their own timing since they are not interested in the pure page loading time. The results show that only one of them, `tsvg_opacity`, had a statistically significant difference from the results from the external optimizations, and the variance actually seems to have gotten worse (CoV 1.35 to 1.88, $p = 0.005$). This indicates that the method of time recording and the number of context switches are not major factors in contributing to the variance in the tests. Interesting to note is that two other tests that should not have been affected also had significant differences (`dromeao_v8`: CoV 0.52 to 0.7, $p = 0.009$; `tsspider`: CoV 0.64 to 1.02, $p = 0.016$).

Table 5: Correlation analysis for the total number of events

Test name	Coefficient	Pearson p-value
`dromaeo_css`	0.30	0.623
`dromaeo_jslib`	0.36	0.554
`dromaeo_sunspider`	0.76	0.135
`dromaeo_v8`	0.41	0.492
`tgfx`	0.95	0.012*
`tp_dist`	0.97	0.033*
`tsvg_opacity`	−0.76	0.236

* $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

4.2 Thread pools

Firefox uses two different mechanisms for handling work like rendering web pages and UI interaction: threads and events. The majority of work is done using events, but threads are used for a few cases where asynchronous operations like I/O and database transactions are needed, for example for bookmark and history handling. In addition Firefox makes use of a thread pool for one-off asynchronous events. Since this thread pool requires creating and destroying threads on a regular basis, changes in the timings of when a new thread is needed could lead to measurable variance caused by these thread interactions.

We investigated this hypothesis by modifying the thread pool code to only ever create one thread that then stays alive for the entirety of the program lifetime, keeping the pool from creating and destroying threads arbitrarily. Unfortunately the results mirror the ones from our first experiment: only two of the tests had statistically significant differences, and in both cases the variance was worse than without our modifications (`dromaeo_dom`: CoV 0.33 to 0.5, $p = 0.002$; `tgfx`: CoV 1.28 to 1.69, $p = 0.026$). So again the thread pool does not seem to be responsible for the variance that we are seeing.

4.3 Event Variance

As mentioned above, events are the main mechanism by which work is done in Firefox. So for our third experiment we wanted to see whether the events used to execute a certain task, like running a test of the test suite, was always done using the exact same events and in the exact same order of dispatch.

For this we again modified Firefox and the test suite to print out special messages at the points where events get dispatched during the tests, and ran a test series. Due to the size of the generated log files and the time it took to run our analysis script afterwards this series consisted of only five distinct runs.

Using the information from our log analyses we can indeed see that there is variation in the number of events being used during the tests. What is interesting is that there are some events that occur several times in some of the runs but not at all in others, but the overall sum of the events differs far less, proportionally speaking. Since the events are identified by their complete backtrace instead of just their class we suspect that this is because those events get dispatched on a slightly different path through the program even though they belong to the same class.

In order to establish whether the event variance is actually correlated to the test result variance we used the Pearson product-moment correlation coefficient (Rodgers & Nicewander 1988), with the null hypothesis being that there is no correlation between the variables.

Table 6: Correlation analysis for the order of events

Test name	Coefficient	Pearson p-value
dromaeo_sunspider	0.58	0.079
tp_dist	0.98	< 0.001***
tsvg	0.44	0.386
tsvg_opacity	0.71	0.113

* $p \leq 0.05$, ** $p < 0.01$, *** $p < 0.001$

The tests that had at least moderate correlation (abs(coefficient) ≥ 0.3) are presented in tables 5 and 6. Only two respectively one of them are actually statistically significant, though, which may be a result of our small sample size. But it does demonstrate that it could be worthwhile to investigate this direction further. One finding that should be studied more closely is that the only test that showed significant correlation in both cases is also the one with by far the longest running time, suggesting that the correlation may only become significant after the test has been running for a while, overshadowing other influences beyond that point.

Interesting to note is that most of the events that appear out of order depend on external or at least asynchronous factors, for example ones that interact with database transaction threads or that make use of hardware timers.

5 Forecasting

After trying to actually reduce the variance as much as possible, we will now look at statistical techniques that aim to separate the remaining noise from genuine performance changes. Since we need test results that contain both of these in order to do that, in this Section we will use data taken from the official Mozilla test servers instead of generating our own. Note that this means that all the results used in this section will be from *different* builds, in contrast to our previous experiments.

5.1 t-tests: The current Talos method

There are essentially three cases that a new value in our results could fall into, and the goal is for us to be able to distinguish between them. The first case is that there are no performance-relevant code changes and the noise is so small that it can easily be classified as a non-significant difference from the previous results. The second one is that there are still no relevant code changes, but this time the noise is much larger so that it looks like there may actually be relevant changes. The last one is that there are relevant code changes and the difference in value we see is therefore one that will stay as long as the new code is in place.

This suggests one potential solution to our problem: if we check more than one new value and determine if – on average – they differ from the previous results in a significant way, we know that there must have been a code change that introduced a long-lasting change in performance. Unfortunately this method has a problem of its own: we cannot immediately determine whether a single new value is significantly different, we have to wait for a few more in order to compute the average.

This is essentially what the method that is currently employed by Mozilla does. In more detail, there are two parts to it:

1. Compute the means of the 30 results before the current one (the *back-window*) and of the 5 runs starting from it (the *fore-window*), that is create two *moving averages.*
2. Use a t-test to determine whether the difference between the means is statistically significant.

The size of these windows again has to be a trade-off: the back-window should be relatively immune to short-term noise but also not be distorted by large changes in the past, and the fore-window should be small enough to allow detecting changes quickly without producing too many false positives due to one or two noisy results.

An important thing to note with regard to the fore window is that it *starts* at the value we are currently investigating, not ends. This is because we are interested in the *first* value where a regression happens. If we interpret the performance change as a “step” like in a step-wise function then starting from the first value after the step means that all of the values that are taken into account for the window will share the same change and thus should ideally lead to a mean that reflects that, pointing back at the “step” that caused it.

In order to determine whether there is a significant difference between the two window sample means we need a statistical test, and Mozilla chose the so-called *Welch’s t-test* which works for independent samples with unequal variances:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

where $\overline{X}_i$, s_i^2 and N_i are the i^{th} sample mean, sample variance and sample size, respectively.

This test statistic t can then be used to compute the significance level of the difference in means as it moves away from zero the more significant the difference is. The default t threshold that is considered to be significant in the Talos analysis is 9. This seems to be another heuristic based on experience, but it can hardly be justified statistically – in order to properly calculate the significance level another value is needed: the *degree of freedom.* Once that is known the significance level can be easily looked up in standard t-test significance tables[2]. However, this degree of freedom has to be computed from the actual data, it cannot be known in advance, and it also would be different for different tests. Using a single threshold for all of the tests is therefore not very reliable.

5.2 Forecasting with Exponential Smoothing

As already mentioned in the previous section, the current method has a few problems. For one thing, the window sizes used are rather arbitrary – they seem to be reasonable, but there is no real statistical justification for them, and the fact that all the values in the window are treated equally presents problems in cases where there have been recent genuine changes. Also, due to the need for the fore window a regression can usually not be found immediately, only after a few more results have come in. Apart from this unfortunate delay this can also lead to changes that go unnoticed because they only exist for a short time, for example because a subsequent change had the opposite effect on performance and the mean would therefore

[2]See for example `http://www.statsoft.com/textbook/distribution-tables/#t`.

hardly be affected. So instead of a potential performance gain the performance will then stay the same since the regression will not get detected.

We therefore need a more statistically valid way that can ideally report outliers immediately and that does not depend on guesses for the best number of previous values to consider.

A common solution to the problem of equal weights in the window average is to introduce weighting, that is a *weighted average*. In the case of our back window we would give the highest weights to the most recent results and gradually less to earlier ones. This would also eliminate the need for a specific window size, since as the weights will be negligible a certain distance away from the current value we can just include *all* (available) previous values in our computation. The only issue in this case is the way in which we assign concrete weights to the previous results.

Exponential smoothing is a popular statistical technique that employs this idea by assigning the weights in an exponentially decreasing fashion, modulated by a smoothing factor, and is therefore also called *exponentially weighted moving average*. The simplest and most common form of this was first suggested by Holt (1957) and is described by the following equations:

$$\begin{aligned} s_1 &= x_0 \\ s_t &= \alpha x_{t-1} + (1-\alpha)s_{t-1} \\ &= s_{t-1} + \alpha(x_{t-1} - s_{t-1}), t > 1 \end{aligned}$$

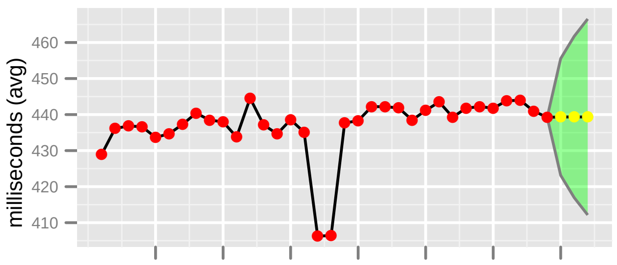
Here s_t is the smoothed statistic and α with $0 < \alpha < 1$ is the smoothing factor mentioned above. Note that the higher the smoothing factor, the *less* smoothing is applied – in the case of $\alpha = 1$ the resulting function would be identical to the original one, and in the case of $\alpha = 0$ it would be a constant with the value of the first result.

The obvious question here is: what is the optimal value for α? That depends on the concrete values of our time series. Manually determining α is infeasible in our case, though, so we would need a way to do it automatically. Luckily this is possible: common implementations of exponential smoothing can use a method that tries to minimize the squared one-step prediction error in order to determine the best value for α in each case[3].

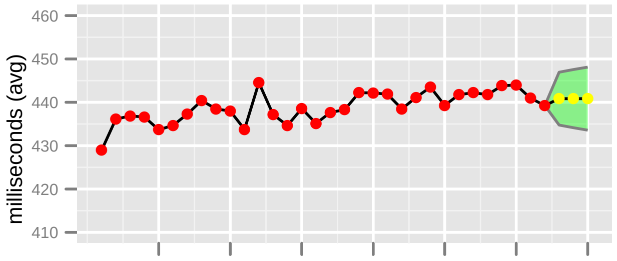
The property that is most important to us about this technique is that it allows us to *forecast* future values based on the current ones. This relieves us of the need to wait for a few new values before we can compute the proper moving average for our fore window, and instead we can operate on a new value immediately. Similarly we do not have to wait until we have enough data for our back window before we can start our analysis. In theory we can start using it with only one value, although in practice we would still need a few values for our analysis to "settle" before the forecasts become reliable.

Normally the exponential smoothing forecast will produce a concrete new value, which is useful for the field of economics where it is most commonly applied. In our case, however, we want to instead know whether a new value that we already have can be considered an outlier. For this we need a modification that will produce *confidence intervals*. Yar & Chatfield (1990) developed a technique for that using the assumption that the underlying statistical model of exponential smoothing is the ARIMA (autoregressive integrated moving average) model, calling the intervals *prediction intervals*. The details of the method are not really relevant here and are also rather complex, so we will refer interested readers to the actual paper instead of repeating them here. We used this modification as it was implemented in the `HoltWinters` package for R (R Core Team 2012).

[3] see for example `http://stat.ethz.ch/R-manual/R-patched/library/stats/html/HoltWinters.html`



(a) Jump included



(b) Jump removed

Figure 5: Prediction intervals for three values

Figure 5a shows an example from the `tp_dist` test with official test server data and the 95 % prediction interval for the next three values. We used three here to make the interval easier to identify, but in practice only one would be needed.

The figure also demonstrates what influence big changes in the past have on the prediction intervals. The big jump in performance in the middle is still reflected in the intervals at the end, although the results themselves would by now clearly lie outside of them if they were to reoccur. Figure 5b shows the same data except that the two outliers have been removed, and we can immediately see that the prediction intervals are now much more narrow – for example the first value would now lie outside of them, which was not the case in the previous figure. Therefore in the case of such apparently genuine changes that have been reverted it might still make sense to remove the values from the ones that are used for future predictions to avoid intervals that are unnecessarily wide.

Note that there are a few extensions to this simple exponential smoothing technique that have been developed in order to deal with data that exhibits trends, but our data does not contain any trends and therefore we did not make use of any of these extensions.

5.3 Comparison of the Methods

We now want to compare our two methods on an example to give an impression of how they differ in their ability to distinguish between noise and genuine changes. For this we used a long stretch of official test data for the `tp_dist` test and ran both methods on it, marking the points where they reported a significant change.

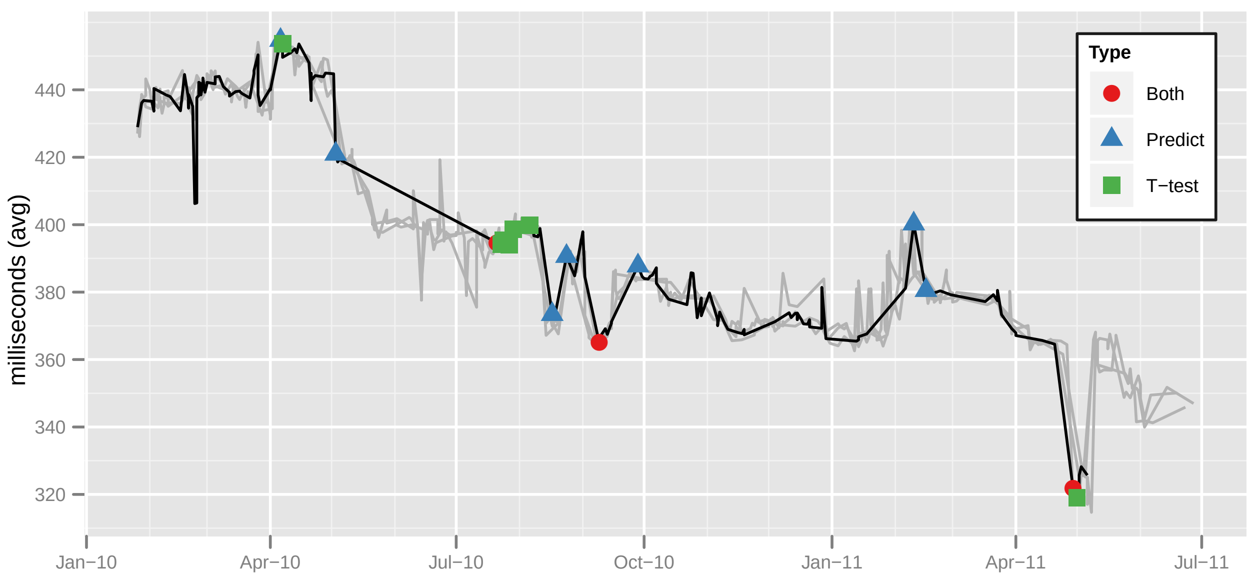


Figure 6: Comparison of the two analysis methods

Figure 6 shows the result of this comparison. The test results from three other machines are also depicted greyed out in the background to easier determine which changes are genuine and which are noise, since the genuine changes will show up in all of the machines.

Two things can be learned from the graph: first, and most importantly, our prediction interval method detects more of the genuine changes than the current *t*-test method. For example, the big jumps in August 2010 and February 2011 go undetected by the current method since they are followed by equally big jumps back soon after. This is a result of the need for more than one value in the respective analysis, obscuring single extreme values in the process. On the other hand, all of the changes that are detected by the old method are also detected by our suggested method, thus demonstrating that previously detectable changes would not get lost with it.

The second difference can be seen during July/August 2010: the current method can sometimes report the same change multiple times for subsequent values, so additional care has to be taken to not raise more alarms than necessary.

This example demonstrates that our proposed statistical analysis offers various benefits over the one that is currently employed. Not only does it give better results, it also needs only the newest value in order to run its analysis. In addition it is also straightforward to implement, several implementations already exist in popular software like R[3] and Python[4].

One disadvantage of our method should be mentioned, however. If there is a series of small regressions, each too small to be detected as an outlier, then the performance could slowly degrade without any warnings being given. Depending on the exact circumstances this degradation might be able to be detected by the old method, but it would probably be better to develop a different method that is specifically tuned for this case and use this method in addition to ours.

6 Related Work

Mytkowicz et al. (2009) investigated the effects of UNIX environment size and the program link order on performance measurements. The found that those factors can have an effect of up to 8 % and 4 %, respectively, on benchmark results and attributed the variance to memory layout changes. As a partial solution they proposed randomizing the setup. Gu et al. (2004) came to a similar conclusion of memory layout changes through the introduction of new code, but found that this variance was not well correlated with the benchmark variance.

Multi-threading variability was investigated by Alameldeen & Wood (2003), including the possibility of executing different code paths due to OS scheduling differences, which they called *space variability*. Georges et al. (2007) demonstrated that performance measurements in published papers often lack a rigorous statistical background and presented some standard techniques that would lead to more valid conclusions.

Kalibera et al. (2005) investigated the dependency of benchmarks on the initial, random state of the system, finding that the between-runs variance was much higher in their experiments than the within-runs variance. They proposed averaging over several benchmark runs to counter this as much as possible, which is similar to what our experiments did.

Tsafrir et al. (2007) demonstrated that influences outside of the control of a benchmark can lead to disproportionally large variance in the results, and suggested "shaking"/fuzzing the input by carefully adding noise so as to make averages more reliable.

7 Conclusions and future work

This paper had three main goals: (1) Identifying the cause(s) of variance in performance tests on the example of Mozilla Firefox, (2) trying to eliminate them as much as possible, and (3) investigating a statistical technique that would allow for better distinction between real performance changes and noise.

Section 3 demonstrated that all of the external factors that we investigated had a certain degree of influence on the variance, with memory randomization being the most influential one. This is consistent with much of the work mentioned in Section 6 that identified memory layout as having a significant impact on performance measurements. We also proposed some

[4]http://adorio-research.org/wordpress/?p=1230

strategies to minimize this variance without the additional resources needed for the averaging solutions that others have suggested.

The studying of the internal factors in Section 4 proved to be less useful than we had hoped for, but it provided us with evidence that they did *not* have a significant amount of influence on the result variance. This suggests that whatever variance remains more likely has to do with the external environment instead of the internal workings of the applications to be measured, allowing better focused future studies.

Finally, in Section 5 we presented a statistical technique for assessing whether a new result in a test series falls outside of the current trend and is therefore most likely not noise. This technique was shown to have various benefits over the currently used one, most importantly it could report some changes that the one that is currently being used by Mozilla missed. Additional advantages include being able to run the analysis on new values immediately instead of having to wait for a certain number of values that are needed for a moving average, and similarly the analysis can start when only a few values are available for a machine unlike the 30 values that are required for the current moving average.

In summary we managed to achieve a certain degree of success for all three of our goals. We identified various external influences and offered solutions to mitigate them, and suggested a statistical technique that improves the quality of change detection. Unfortunately we did not conclusively find a connection between the inner workings of Firefox and the measured variance, but we did find a certain amount of internal variance. Investigating this discrepancy could be a promising topic for future work.

Another worthwhile direction would be to apply our research to other applications, especially other browsers like Google Chrome. This was outside the scope of this paper, not the least because those browsers use entirely different – and not in all cases even publicly accessible – performance test suites. The general principle should be the same, though, so it would be interesting to see whether there are any differences between the amount of and the causes of variance. At least our statistical technique is not tied to any specific application and should work for anything that can be represented as a time series, regardless of how the data was produced.

Acknowledgements

The authors would like to thank Mozilla for funding this research and especially Robert O'Callahan for his invaluable insights whenever a question about Firefox internals came up.

References

Alameldeen, A. R. & Wood, D. A. (2003), Variability in architectural simulations of multi-threaded workloads, *in* 'High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on', pp. 7–18.

Brown, M. B. & Forsythe, A. B. (1974), 'Robust tests for the equality of variances', *Journal of the American Statistical Association* **69**(346), 364–367.

Drepper, U. (2007), 'What every programmer should know about memory', `http://people.redhat.com/drepper/cpumemory.pdf` [22 April 2012].

Fowler, M. (2006), 'Continuous integration', `http://www.martinfowler.com/articles/continuousIntegration.html` [22 April 2012].

Georges, A., Buytaert, D. & Eeckhout, L. (2007), Statistically rigorous java performance evaluation, *in* 'Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07', Montreal, Quebec, Canada, p. 57.

Gu, D., Verbrugge, C. & Gagnon, E. (2004), 'Code layout as a source of noise in JVM performance', *In Component and Middleware Performance Workshop, OOPSLA* .

Holt, C. C. (1957), 'Forecasting seasonals and trends by exponentially weighted moving averages', *International Journal of Forecasting* **20**(1), 5–10.

Kalibera, T., Bulej, L. & Tuma, P. (2005), 'Benchmark precision and random initial state', *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems* pp. 853—862.

Larres, J., Potanin, A. & Hirose, Y. (2012), A study of performance variations in the mozilla firefox web browser, Technical Report 12-14, Victoria University of Wellington.

Levene, H. (1960), Robust tests for equality of variances, *in* I. Olkin, S. G. Ghurye, W. Hoeffding, W. G. Madow & H. B. Mann, eds, 'Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling', Stanford University Press, pp. 278–292.

Mytkowicz, T., Diwan, A., Hauswirth, M. & Sweeney, P. F. (2009), Producing wrong data without doing anything obviously wrong!, *in* 'Proceeding of the 14th international conference on Architectural support for programming languages and operating systems', ACM, Washington, DC, USA, pp. 265–276.

O'Callahan, R. (2010), 'Private communication'.

R Core Team (2012), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Rodgers, J. L. & Nicewander, W. A. (1988), 'Thirteen ways to look at the correlation coefficient', *The American Statistician* **42**(1), 59–66.

Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N. & Boneh, D. (2004), On the effectiveness of address-space randomization, *in* 'Proceedings of the 11th ACM conference on Computer and communications security', CCS '04, ACM, Washington DC, USA, p. 298–307.

Tsafrir, D., Ouaknine, K. & Feitelson, D. G. (2007), Reducing performance evaluation sensitivity and variability by input shaking, *in* 'Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS '07. 15th International Symposium on', pp. 231–237.

Yar, M. & Chatfield, C. (1990), 'Prediction intervals for the Holt-Winters forecasting procedure', *International Journal of Forecasting* **6**(1), 127–137.