Performance Variance Evaluation on Mozilla Firefox

by

Jan Larres

A thesis
submitted to the Victoria University of Wellington
in partial fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2011

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 hard since results that differ from previous results cannot easily be attributed to either genuine changes or noise.

In this thesis we use Mozilla Firefox as an example to try to find the causes for this *performance variance*, develop ways to reduce the noise and present a statistical technique that makes identifying genuine performance changes more reliable.

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.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my family for always supporting me and making this course of study possible at all.

I would also like to thank my supervisors Alex Potanin and Yuichi Hirose for their continuous help and guidance, and John Haywood for his help with statistical issues concerning time series.

Many thanks to Robert O'Callahan for making this thesis possible in the first place, for his valuable insights whenever I had a question about Firefox and for inviting me to the Mozilla All Hands meeting in California.

More thanks to Craig Anslow, Yi-Jing Chung, Harsha Raja, Juan Rada-Vilela, and several others too numerous to mention for their help and just being good friends and office mates.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

1

> Anything that happens, happens.
> Anything that, in happening, causes something
> else to happen, causes something else to happen.
> Anything that, in happening, causes it-
> self to happen again, happens again.
> It doesn't necessarily do it in chronological order, though.
>
> *Mostly Harmless*
> Douglas Adams

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.

A recent example of this is the huge growth of the so-called "Web 2.0". This collection of techniques relies heavily on JavaScript to build applications in websites that are as easy and fast to use as local applications. The bottleneck here is obvious: the performance of the applications depends on how fast the browser is able to execute the JavaScript code. This has led to a speed race in recent years, especially the last one, with each browser vendor trying to outperform the competition.

A competition 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.

It is therefore important to determine as soon as possible whether performance changes have taken place. This is traditionally being done with automated tests. If a regression is detected an investigation has to be made: Is it caused by the fulfilment of a different requirement that is more important? Then it cannot be avoided. But if it is an unexpected side effect then this change could be reverted until a better solution without side effects is found. There is one important catch with this technique, however: the performance data has to be reliable. In this case that means it should reflect the actual performance as accurately as possible without any noise. Unfortunately this is much more difficult than it might seem. Even though computers are deterministic at heart, there are countless 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 work tries to determine what exactly those 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.

1.2 OUTLINE

The rest of this thesis is organized as follows.

Chapter 2 gives an overview of the problem using an example produced with the official Firefox test framework and presents related work.

Chapter 3 looks at external factors that can influence the performance variance like multitasking and hard drive access.

Chapter 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.

Chapter 5 presents a statistical technique that improves on the current capability of detecting genuine performance changes that are not caused by noise.

Finally, Chapter 6 summarizes our results and gives some suggestions for future work.

BACKGROUND

2

This chapter will give an overview of Mozilla Firefox and the Talos performance test suite that Mozilla employs to detect performance changes, namely improvements and regressions, in new code. It will also give an example of the problem of variance in this test suite and list previous work done in the area.

2.1 MOZILLA AND MOZILLA FIREFOX

The Mozilla Foundation[1] is a global non-profit organization with its headquarters in the USA. Its mission is to "promote openness [...] on the web"[2] and make sure that it is accessible for everyone using Free and Open Source tools[3,4].

The main means of pursuing this goal is by having The Mozilla Corporation[5], a subsidiary of the Mozilla Foundation, develop the Firefox web browser and releasing it as Free Software[6]. Firefox is a modern web browser that supports a wide range of web-related standards like HTML5, CSS in various versions, JavaScript, and a lot more. The Firefox source code is kept in a publicly accessible *Mercurial*[7] version control repository[8]. For this work version 5.0 of Firefox was used which was the current version at the time when we started collecting the final data.

[1] http://www.mozilla.org/
[2] http://www.mozilla.org/about/mission.html
[3] http://www.mozilla.org/about/
[4] http://www.mozilla.org/about/manifesto.en.html
[5] http://www.mozilla.com/
[6] http://www.mozilla.org/MPL/license-policy.html
[7] http://mercurial.selenic.com/
[8] http://hg.mozilla.org/

2.2 THE TALOS TEST SUITE

The *Talos* test suite (named after the bronze giant from Greek legend that protected Crete's coasts) 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 2.1. One thing to note here is that there are two types of results for the individual tests: most of them measure the milliseconds it takes for a specific action to complete, so the lower the result the better, but the tests that are part of the `dromaeo` framework measure the number of times a specific test can be run during one second, so here a higher number is better.

The purpose of this test suite is to evaluate the performance of a specific Firefox *build*, meaning the result of the compilation of a specific version of the source code. As new code is checked in into the Mercurial repository, various actions (see below) are performed on it in order to assess the quality of the new code. Since a complete run of the whole process takes about 4 hours or more (Stoica, 2010), a build infrastructure consisting of about 1000 machines, mostly Mac Minis (Gasparnian, 2010), is used to carry out those actions. In a slightly simplified overview this process consists of three parts:

1. The new code is compiled on a range of different operating systems, namely Windows, Mac OS X and Linux. This is both to ensure that the current version actually cleanly compiles on all of these operating systems and to have a working build for the next two parts.

2. A number of unit tests are run on the new build. This tries to ensure that the code changes did not introduce any bugs, like for example crashing when a certain action is performed or displaying popular pages incorrectly.

3. The Talos test suite is run on the build. This is done both to track improvements in performance and to detect regressions.

This process of *Continuous Integration* (Fowler, 2006) allows for quick detection of problems that could otherwise lead to a lengthy search for the cause and ensures a consistent quality throughout the project.

TABLE 2.1
The various performance tests employed by Mozilla

Test name	Test subject	Unit
a11y	Accessibility features	Milliseconds
dromaeo_basics	Basic JavaScript operations like array manipulation and string handling	Runs/second
dromaeo_css	CSS (Cascading Style Sheets) manipulation with JavaScript	Runs/second
dromaeo_dom	DOM (Document Object Model) node manipulation with JavaScript	Runs/second
dromaeo_jslib	DOM node manipulation using the 'jQuery' and 'Prototype' JavaScript libraries	Runs/second
dromaeo_sunspider	Various JavaScript tests from the 'SunSpider' WebKit test suite (Stachowiak, 2007), integrated into the 'Dromaeo' suite	Runs/second
dromaeo_v8	Various JavaScript tests from the 'V8' Google Chrome test suite (Google Inc., 2008), integrated into the 'Dromaeo' suite	Runs/second
tdhtml	Various tests that create animations using JavaScript DOM manipulation	Milliseconds
tgfx	Some graphics operations like displaying a large amount of text, tiled images, image transformations and various borders	Milliseconds
tp_dist	A page loading test that loads a number of popular websites and measures the speed it takes to render them	Milliseconds
tp_dist_shutdown	The time it takes to completely shut down the browser after the page loading test	Milliseconds
tsspider	The unaltered SunSpider JavaScript benchmark	Milliseconds
tsvg	Rendering of SVG images	Milliseconds
tsvg_opacity	Rendering of partially-transparent SVG images	Milliseconds
ts	Startup time until the first page gets loaded	Milliseconds
ts_shutdown	Shutdown speed directly after starting up the browser	Milliseconds
v8	The unaltered V8 JavaScript benchmark	Milliseconds

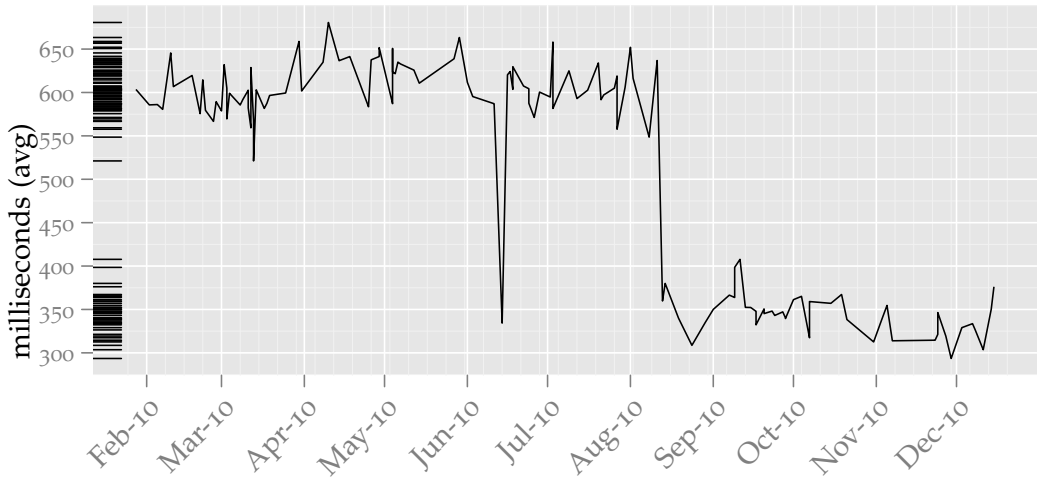


Figure 2.1: Page load speed `tp_dist` example sequence with data taken from `graphs.mozilla.org`

Ideally this process would be run on every check-in into the repository. However, in order to reduce the load on the machines used and to allow for quick fixes of mistakes in a commit (like for example forgetting to add a file) there is a short wait period before the build starts. If there is a new check-in during that time it will be included in the next build.

The focus of this work is on part three, the Talos performance evaluation. 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).

2.3 AN ILLUSTRATIVE EXAMPLE

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, usually consisting of 30 single runs (see also Section 2.5.1).

Figure 2.1 illustrates an example series of the `tp_dist` part of the test suite over most of the year 2010 on one particular machine. This test loads a number of popular web pages and calculates the mean of the time it took to completely render them. Important to note here is that the pages are loaded

from the local hard disk, so effects like network latency do not come into play – however, the speed and latency of the hard drive, and potentially other external factors, can still have an effect outside the control of Firefox itself. This will be addressed in Chapter 3.

There are a few interesting observations to be made in this graph. One is the big drop in August, going from about 600 to a bit over 300 and then staying there. This looks like a clear case of a genuine change in performance, most likely due to an optimization in the code. Another observation is about the high variance in the results during the rest of the year. There seems to be no common trend to them, they are "all over the place". We cannot really tell whether those results are due to noise or real code changes. One clear candidate for a code change happens in the middle of June, where the result fits right into the trend that will be established later on in August. But why is it only a single result, as opposed to the later ones? One possible explanation is that the optimization introduced a bug and was therefore removed again until that bug was fixed, which took until August.

So now we have a plausible explanation for one of the results. But that still does not really tell us anything about the rest. Could we apply the same heuristic that lets us explain the big change – seeing it "sticking out" of the general trend – and use it in a more statistically sound way to try to explain the other results? We can – to a certain degree.

The exact details of the best way to do this will be explained in Chapter 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.

2.4 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 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.

As a concrete example, the `tp_dist` test as used in our experiments loads 26 different pages 10 times each. Then the median of the 10 results from each page is calculated, and finally the mean of all the different medians is presented as the final result. This allows us to make use of the *central limit theorem* (Cam, 1986), which states that our results will approximately follow a normal distribution as long as they all come from the same distribution – in our case this means that the source code has to remain unchanged in between runs. But as mentioned earlier we are only concerned with unchanging code anyway so this poses no problem for us. Interesting to note is that Figure 2.1 shows in the so-called "rug" plot on the left that even with changing code the test seems to largely follow a normal distribution, with the exception of the large jump in August which essentially split the distribution into two independent ones.

Normal distributions make it easier to apply various statistical analyses on data, but it is not strictly required in our case. Still, checking for normality of the distribution can give valuable insights about the nature of the variance.

Beginning with this chapter we will be using various statistical techniques to evaluate our results in a statistically valid way. This usually consists of having a *null hypothesis*, which is the "conservative" view that the results are in line with our current theory and do not indicate that the current theory might be wrong. What exactly this means for a specific test will be explained in the relevant sections.

The other part of these techniques is the p-value. This is the probability that the null hypothesis *can not be rejected*. In other words, it is the probability of getting the results we are analysing under the assumption that the null hypothesis is true. If this probability is lower than a previously chosen *significance level* then the results are said to be *statistically significant*. Traditionally a significance level of 0.05 is the most common one for these

kinds of analyses, and that is what we will be using here.

2.5 THE BASE LINE TEST

2.5.1 EXPERIMENTAL SETUP

For this and all the following experiments in this thesis 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 without any special adjustments of our own. Using the same executable guarantees that changes in the performance cannot be caused by code changes and are thus solely attributable to noise. The only adjustments that we made were two techniques used on the official Talos machines[9]:

- Replace the `/dev/random` device, which provides true random numbers, with the pseudo-random number generator `/dev/urandom`.
- Disable CPU frequency scaling and fix the processors at their highest frequency. This prevents variance introduced by switching between the possible frequencies and the case where a processor decides to run at different frequencies during repeated runs of the same test for some reason.

The number 30 for the runs was chosen as a compromise between different requirements. The first was that in order for the central limit theorem to be applicable the common rule of thumb is that at least 30 samples are needed. In addition a higher number of runs would allow us to determine whether the results would settle in some kind of *steady state* where the variance is much lower than between the first few runs. Finally, a practical requirement prevented us from choosing a significantly higher number: since every test run took about one hour to complete

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

on our machine we had to settle on a number that would allow us to reasonably experiment with many different parameters without having to wait unreasonably long for the result. In addition initial tests with 50 runs showed no meaningful difference between the numbers. Thus 30 was chosen as a suitable compromise.

2.5.2 RESULTS

Figure 2.2 shows the results of the `tp_dist` page loading test, and Figure 2.3 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. Looking at the rug plot it seems that the `tp_dist` test does *not* follow a normal distribution, the `a11y` on the other hand looks better. There are two ways to verify these suspicions: *quantile-quantile (Q-Q) plots* and the *Shapiro-Wilk test* (Shapiro and Wilk, 1965).

Figure 2.4 shows the Q-Q plots for our two example tests. They are interpreted roughly in the following way: if the data points closely follow the line the sample is said to follow a normal distribution. The `a11y` test supports that, except for two outliers the points follow the line very well. However, as already suspected, this is not true for the `tp_dist` test – most of the points are quite far away from the line. It is interesting to note, though, that there seem to be two different linear trends in the data points – one in the points near the bottom of the graph and one near the top right, almost as if there are two different influences guiding them.

For a technique that needs less interpretation we can use the Shapiro-Wilk test. It analyses the sample and determines whether the null hypothesis of the distribution being normal can be rejected or not. The resulting p-value for the `a11y` test is 0.135, implying that the normality of the sample cannot be rejected if we use the standard significance level of 0.05. For the `tp_dist` test however, p is < 0.01, so we have the affirmation that the sample is most likely not normal.

Table 2.2 shows a few properties of the results for the complete test suite.

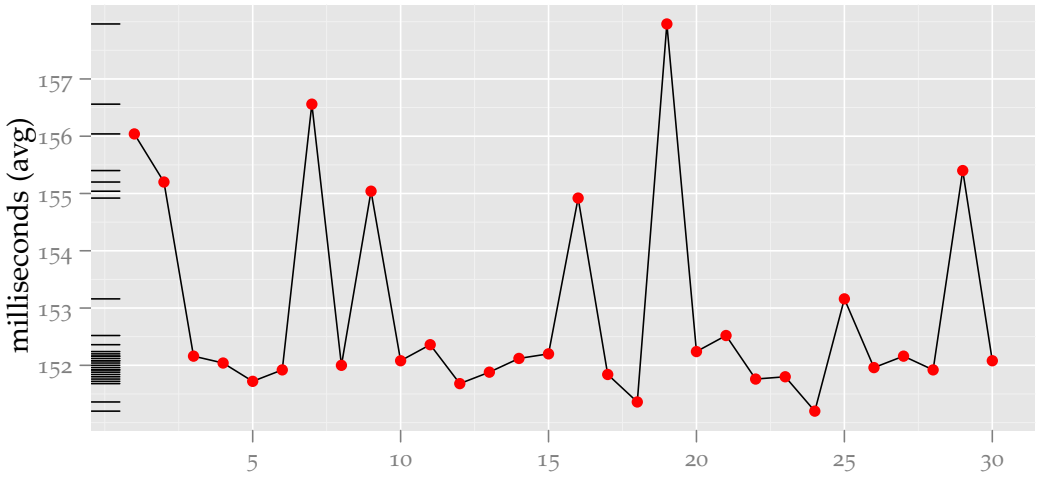


Figure 2.2: `tp_dist` results of 30 runs

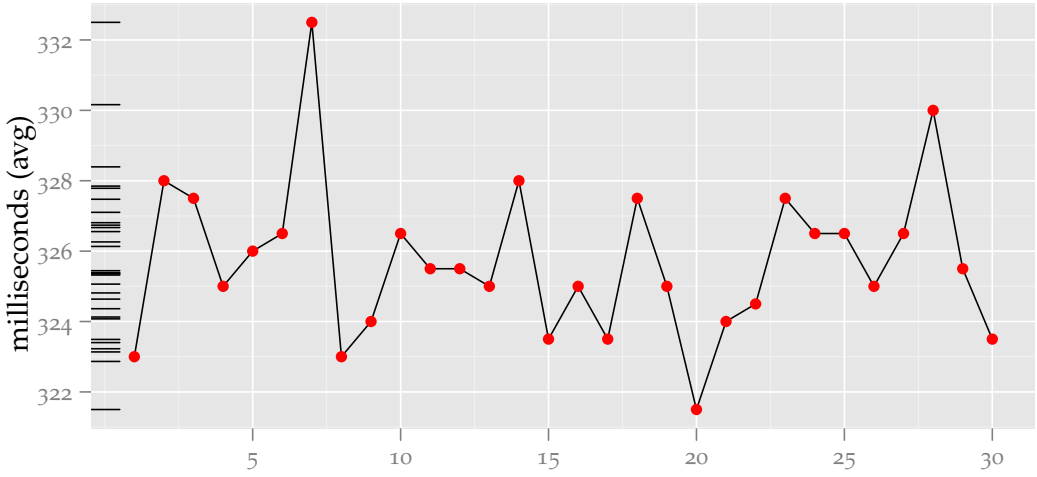
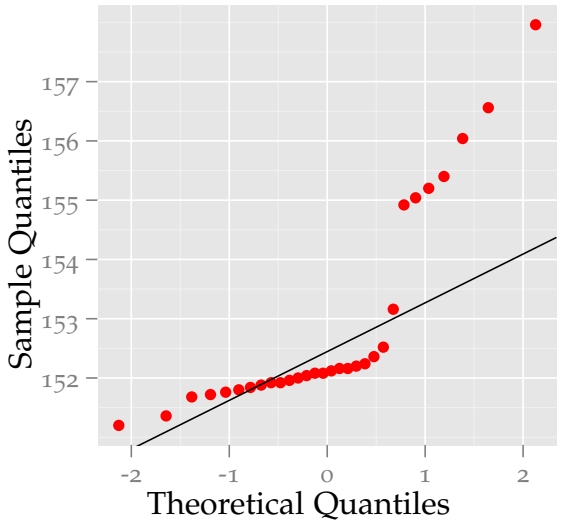
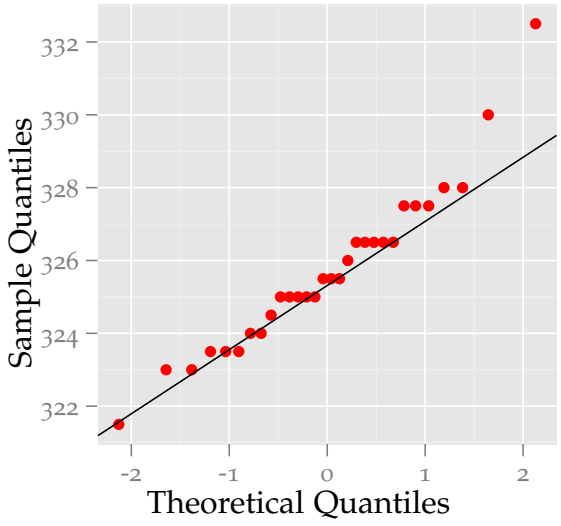


Figure 2.3: `a11y` results of 30 runs



(a) Q-Q plot for `tp_dist`



(b) Q-Q plot fot `a11y`

Figure 2.4: Two example Q-Q plots

TABLE 2.2
Analysis of results

			Max diff (%)		
Test name	StdDev	CoV	Absolute	To mean	p-value
a11y	2.23	0.69	3.38	2.08	0.135
dromaeo_basics	4.41	0.53	2.57	1.62	0.064
dromaeo_css	11.36	0.30	1.39	0.88	0.135
dromaeo_dom	1.02	0.41	1.99	1.14	0.338
dromaeo_jslib	0.53	0.30	1.19	0.60	0.661
dromaeo_sunspider	5.65	0.54	2.09	1.16	0.017
dromaeo_v8	2.02	0.86	3.03	1.77	0.006
tdhtml	0.94	0.33	1.31	0.73	0.156
tgfx	0.80	5.68	25.60	18.88	< 0.001
tp_dist	1.77	1.16	4.42	3.30	< 0.001
tp_dist_shutdown	27.09	5.14	16.51	8.72	0.080
ts	2.27	0.59	2.45	1.66	0.001
ts_shutdown	7.28	2.00	6.88	3.44	0.410
tsspider	0.11	1.15	4.04	2.57	0.014
tsvg	1.43	0.04	0.17	0.10	0.267
tsvg_opacity	0.62	0.74	3.56	2.02	0.055
v8	0.11	1.42	4.31	3.59	< 0.001

As a typical statistical measure we included the standard deviation and the coefficient of variation (CoV), which is simply the standard deviation divided by the absolute value of the mean for easier comparison between different tests. The standard deviation shows us that, indeed, the variance for some of the tests is quite high. The general idea here 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.

Our first approach in this chapter is to simply 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 and Wood (2003). If a new result would increase this value, it would be assumed to not be noise. The result of this analysis can be seen in

Table 2.2. 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. But what if we use a slightly different method? We could measure the difference from the *mean* instead of between the highest and lowest value. Checking our table again again we can see that the values in this case do look better, but they are still too far away from being actually useful.

An additional problem with the two techniques just explained is that they do not account for significant changes in the performance. For example, in a situation similar to that in Figure 2.1 computing the maximum difference or the difference from the global mean would lead to highly unreliable results due to the big, genuine changes in June and August – since most changes, both other genuine ones and those caused by noise should usually be far smaller than that they will remain completely undetected. So our simple approach is clearly not sufficient.

Chapter 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 chapters 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.

2.6 RELATED WORK

This section will present two different approaches that have been used in previous work to reduce nondeterminism in software testing: trying to identify and measure the actual variance and trying to increase the determinism in multi-threaded programs.

2.6.1 VARIANCE MEASUREMENTS

Mytkowicz et al. (2009) investigated what they called *measurement bias*: small changes in an experimental setup that can introduce significant bias in the result. They claimed that many researchers do not pay enough attention to

this bias and investigated its effect on a set of experiments. Specifically they considered two different scenarios: (1) the UNIX environment size and (2) the program link order. They found that both can have a measurable impact on benchmark results, up to 8 % for the environment size and 4 % for the link order. The cause in both cases was attributed to memory layout. The size of the environment influenced the beginning of the stack and thus the alignment of its content, resulting in a layout that was not optimal for the hardware architecture in many cases. Similarly, the link order changed the code layout in the executable which affected hardware buffers and various other hardware aspects like branch prediction. As a partial solution to this problem they proposed using a large benchmark suite and randomizing the experimental setup.

A similar conclusion was reached by Gu et al. (2004). Their original goal was to evaluate different object layouts in the Sable Java VM[10] for its copying garbage collector. However, during their experiments they discovered unexpected differences in performance that could not be explained by their layout changes. This led them to investigate how the changes affected the low-level code execution by using hardware performance counters. They found that even the adding of code that was never executed could lead to shifted code segments in the resulting executables which then has a measurable impact on the instruction cache, the cycle count, and the data cache. These differences were still not well correlated with the performance changes, though.

The presence of variability in multi-threaded workloads both in real and simulated systems was investigated by Alameldeen and Wood (2003). They described two different kinds of variability: time variability, that is different performance characteristics during different phases of a single run, and space variability, the execution of different code paths caused for example by the operating system scheduling threads differently during different runs. They showed that variability can be a problem even in deterministic simulations under certain conditions. In order to quantify their results they

[10] `http://www.sable.mcgill.ca/`

introduced two new metrics: the *wrong conclusion ratio*, the percentage that a wrong conclusion is drawn from an experiment pair, and the *range of variability*, the difference between the maximum and minimum values of a series of runs as a percentage of the mean, which is identical to our absolute maximum difference metric. As a solution to the variability problem they proposed averaging over a number of runs using some statistical techniques to estimate the optimal sample size.

Georges et al. (2007) tried to give the performance analysis of Java programs a statistically sound base since they noted that many published papers lack a rigorous statistical background and instead invent their own methods for analysing results. This situation can lead to incorrect conclusions in extreme cases, especially since managed runtime systems are notoriously difficult to benchmark. They first gave an overview of basic statistical concepts like confidence intervals, the central limit theorem, and the ANOVA test for comparing alternatives. These techniques were then demonstrated on an example that measured the start-up and the steady-state performance of various garbage collector strategies in the Java VM, which also showed that their results occasionally differed from results in other papers that did not use the same rigorous approach.

Kalibera et al. (2005) investigated the dependency of benchmarks on the initial, random state of the system. They claimed that the variation during one run of a benchmark, even if it accounts for things like external disruptions, does not capture the true extent of possible variance and that benchmarks are therefore not the reproducible processes they are usually thought to be. They tested their assumptions on a few benchmarks that produced many samples in one run to be averaged over and ran them several times independently, finding that the between-runs variance was much higher than the within-runs variance. Similar to previously mentioned papers they tested two example causes of such variance: non-determinism in memory allocation and code compilation. Their investigation revealed that there is some correlation between those phenomena and variance but they admitted that listing and eliminating all possible causes would be

impossible. They, too, suggested a benchmarking setup that tries to alleviate the problem as much as possible by running a benchmark several times to be able to use statistical methods that take both kinds of variance into account and so end up at a more reliable average. Note that this setup is similar to ours as the Talos tests already produce a within-tests average that is then used for our between-tests/runs analysis.

A slightly different issue, the effect of "coincidental artifacts" on an evaluation, was investigated by Tsafrir et al. (2007). They defined coincidental artifacts as effects that influenced the outcome of a performance evaluation but were outside the scope of the benchmark, like "unique interactions between the system and the specific trace used". They gave the example of a scheduler evaluation on a specific job workload where changing the workload by only 0.046 % changed the result by 8 %. To alleviate this problem they introduced their methodology of *shaking* the input, that is running the benchmark several times with a different set of noise added to the workload each time[11] and then averaging over the result, and demonstrated it on a scheduling algorithm and a set of different workloads. An important consideration of this technique that they mentioned was that care has to be taken when shaking the workload so that it does not get distorted in a way that will lead to unreliable results, meaning that only less fragile parameters should be modified. Their results showed that even with a relatively small amount of shaking the reliability of their benchmark could be significantly improved, leading to a smoother progression with fewer outliers. Note that this technique is not really applicable for us since many of our tests evaluate concrete functionality instead of a random workload and the required repeated tests would increase our test times too much.

2.6.2 DETERMINISTIC MULTITHREADING

Most of the work concerned with the determinism of multi-threaded programs until recently has only dealt with the problem of *replayability*, that is a technique that records a log of one run and then offers the possibility

[11] This is also known as *fuzzing*.

to replay that run on another machine for debugging purposes. This is not useful for our purposes since we are not concerned with a single run but with a comparison between different runs, especially since those techniques usually do not pay attention to performance characteristics. However, in the last few years there have been some attempts to introduce determinism to a wider range of uses. Since threads are used for a few purposes in Firefox (see Chapter 4 for details) this is worth looking into.

Devietti et al. (2009) presented a way to make threading deterministic using a technique they called *deterministic serialization* of parallel execution. Their technique works by introducing a token that is required for each memory access – or each group of finite accesses they called a *quantum* – and is passed on from thread to thread in a deterministic fashion. Since this method introduces significant performance degradation due to threads having to wait for the token they proposed some hardware changes that would drastically reduce the overhead. In order to support this hypothesis they implemented the changes in a hardware simulator and as a pure software framework for comparison. Their results showed that the hardware version had negligible performance degradation compared to a nondeterministic system and that the software version was at least usable for debugging purposes.

A somewhat similar approach was used by Olszewski et al. (2009). However, their goal was to make thread determinism usable on today's commodity hardware without requiring hardware changes. In order to reduce the performance degradation of this approach they used what they called *weak determinism* which does not apply to every memory access but instead only to lock acquisition. For this they implemented a deterministic subset of the POSIX Thread (pthread) API that utilized hardware performance counters to track the locking behaviour. They then used the SPLASH-2 benchmark suite to evaluate their framework, finding that it only introduced a 16 % overhead on a 4-processor system. Unfortunately, due to their changes to the pthread library it is not possible to run any arbitrary application; only a subset of programs work.

Bergan et al. (2010a) introduced a "compiler and runtime system" based on the LLVM compiler suite that uses a sophisticated mechanism based on an ownership model for memory regions and a deterministic commit protocol for committing changes to shared memory. They showed that their system scales quite well, but it does introduce a performance loss of 1.2×–6×. In addition it is highly dependent on small changes in either the input or the program itself; while it guarantees that the same program run with the same input will always execute in a deterministic fashion there are no such guarantees for even small changes in the program.

A new operating system abstraction called a *Deterministic Process Group* (DPG) was proposed by Bergan et al. (2010b). These DPGs are defined as groups of processes that are executed completely free of *internal* nondeterminism like thread scheduling and were implemented using techniques introduced by Devietti et al. (2009) and Bergan et al. (2010a). In addition they described a type of program called a *shim* that acts as a wrapper around a DPG and that can be used to eliminate external nondeterminism like file access and to implement a record/replay mechanism. Similar to the previous work they introduce some amount of performance loss (around 2.5× on average) and do not guarantee determinism after program changes. The authors also explicitly state that "DPGs guarantee deterministic output, but not deterministic performance".

Cui et al. (2010) tried to address the problem of input dependence for deterministic execution by creating a what they called *stable* system based on schedule memoization. Their idea was that many working schedules can be reused even for different inputs since the internal workings of a program stay the same. In order to accomplish this they developed a system that memoizes working schedules along with their constraints on the input so they can be recalled if new input matches the given constraints of a past schedule. Their implementation also utilizes LLVM and only considers lock synchronization instead of full memory access synchronization similar to Olszewski et al. (2009) for performance reasons. Depending on the use of locks in the programs this can still lead to significant performance overhead,

though. In addition their system requires some changes to the source code of the programs and does not guarantee deterministic performance, only behaviour for the memoized schedules.

Considering all of the constraints of these deterministic multithreading systems we must conclude that they are not really applicable to our situation, at least not yet. While the current systems provide a definite benefit for tasks like debugging the fact that performance determinism is not guaranteed (and indeed probably impossible) due to the way these systems realize their threading guarantees makes them unsuitable for analysing variance. This is because in order to guarantee execution determinism the systems can suspend threads if they are scheduled by the operating system in a different way from the first run, and they then have to wait for the operating system to schedule the thread that is actually supposed to be run at a certain point in time. In this way the threads are executed in exactly the same order every time, but the actual timing can vary wildly depending on the operating system's scheduling decisions. However, it will be interesting to follow the developments in this field of research.

ELIMINATING EXTERNAL FACTORS

3

In this chapter we will deal with eliminating factors that are outside the influence of Firefox itself. For example, since modern operating systems allow multitasking there will usually be several programs running concurrently at any one time – both user-level applications like file managers, word processors and the like, and system-level services that are required for various maintenance tasks. The operating system's job is to manage these programs in a way that is transparent to them, so the programs have only very limited knowledge about *how* exactly their tasks are executed. Thus depending on the other things going on in the system a program will most likely be executed in subtly different ways each time it performs a task, potentially leading to measurable performance differences. A description of the most important of these system-level issues and how to eliminate their interference follows.

3.1 OVERVIEW OF EXTERNAL FACTORS

3.1.1 MULTITASKING

As mentioned above, modern operating systems have many programs running at the same time. At least that is the impression that a user of those systems gets – the reality is significantly more complicated.

Processors are strictly serial systems, that is they can only do one thing at a time. This is clearly at odds with the requirement of running several programs at the same time, i.e. with multitasking. The solution that modern operating systems use is to *give the appearance* of the programs being executed concurrently by switching between them very quickly in a way that is completely transparent to the programs.

The way programs are actually executed is by running them as *processes*. A process is basically a running representation of a program with its own variables, program counter and registers. Tanenbaum (2001) gives the fol-

lowing analogy about the relationship between programs and processes: imagine the program being a cake recipe, the input being the ingredients, and the output being the cake. Then the person baking the cake is the processor, and the process is the whole activity of the person baking the cake using the recipe and the ingredients. Multitasking in this analogy could be explained as the person being interrupted by another person with an urgent task, so that the baking has to be suspended for some time while the other task is being attended to.

Traditionally processes are classified as either CPU-bound or Input/Output (I/O)-bound (see for example Bovet and Cesati (2005)). CPU-bound processes do heavy computations and thus need the CPU as often as possible, I/O-bound processes are waiting for I/O-operations to complete most of the time and therefore have no need of the CPU until then. The *scheduler* of an operating system has the job of weighing the needs of the different processes and schedule them in a way that is both fair to all of the processes and that guarantees a responsive system with a minimum amount of delays. On systems that support preemptive multitasking, which is the norm today, processes can be switched at (almost) any time, making this task much more flexible since the scheduler does not have to wait for a process to give up the CPU voluntarily. Instead each process is assigned a specific time-slice whose length depends on various parameters like the specific scheduler implementation and the process priority, and when this slice runs out the process is switched out for a different runnable process, that is a process that is not waiting for I/O. On multi-processor systems this mechanism is essentially used for each processor independently but with a common pool of processes; see also Section 3.1.2. All of this happens transparently to the programs; to them it looks like they are able to run continuously.

As useful as this mechanism is, it has various unavoidable drawbacks. The most obvious one is probably that the more programs are trying to run at the same time, the less time each of them gets to use in a given time interval, and the more time has to be spent on switching between processes. The fact that many programs are waiting for I/O or other specific events

and only have to use the CPU occasionally unfortunately makes this even worse for our case.

As a simple example let us assume that we want to measure the amount of time that a CPU-intensive application takes to complete a specific task, for instance a complicated calculation. To simplify the scenario we assume that all other processes are currently in a waiting state and do not use the CPU. To do our measurements we use a function that reports the current time or simply look at a watch before and after the calculation, which when subtracted from each other will give us the amount of time our application took. Just to be sure we want to do our calculation again, expecting the same result. But this time, halfway through our calculation, a second process suddenly wakes up – for example a virus scanner that wants to do its daily check, or even just something simple like a network application receiving data from outside that it has to handle. So now our application has to be switched out and will keep getting swapped with the other process until either of them finishes. Due to this the result that we get from our simple time-keeping method will most likely be different from the result of our first run, even though the application did exactly the same thing and, by itself, ran for the same amount of time.

This example illustrates two things: (1) care has to be taken as to what other programs are running during tests, and (2) using "real time" (also called *wall clock time*) is not the best way to measure the performance per time interval of a specific program. Instead, a mechanism that only measures the time the process actually ran is needed.

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. For a more detailed discussion about threads and how they are used in Firefox see Chapter 4.

3.1.3 ADDRESS-SPACE RANDOMIZATION

Buffer overflows are a big issue in all non-managed programming languages. In simplified terms a buffer overflow describes a situation where more data is written into a buffer than fits into it, and the extraneous data then gets written into a consecutive memory region that holds completely different data, thereby destroying it. Apart from just destroying data this is also a threat to the security of a system, since in some cases a carefully crafted deliberate buffer overflow can allow an attacker to execute arbitrary commands through this technique, for example by overwriting the return address on the stack to jump to attacker-chosen, executable memory (Cowan et al., 2000). This is especially dangerous nowadays where most computers are connected to the Internet and thus easily reachable by malicious people.

However, in order for this attack to work the attacker has to know exactly what is where in the address space of the program, since they have to overwrite specific regions with data that will then get called from other regions. Thus many modern operating systems can use (among others) a technique called *address-space randomization* (Shacham et al., 2004). What this essentially does is making the position of the memory allocated by the program unpredictable by randomizing it, thereby preventing the exploitation of the address space layout for this kind of attack.

Unfortunately, for our purposes this normally very useful technique can do more harm than good. For example, 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[1].

In addition the randomization can lead to data structures being aligned differently in memory during different executions of the same program, again introducing variance as observed by Mytkowicz et al. (2009) and Gu et al. (2004).

For our tests we therefore want the memory layout to be as deterministic as possible.

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

[1] See for example Drepper (2007) for more information on this topic.

we worked on the same executables using the same environment and so those effects have not been studied further. In addition hardware effects that could be caused by things like varying temperatures were assumed to be negligible.

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 was terminated. The previous tests were run with an idle desktop, but here we rigorously disabled everything non-vital, including network, to minimize the impact of scheduling effects.
- Address-space randomization was disabled in the kernel.
- The Firefox process was bound to an exclusive CPU. Since we used a dual-core system we restricted all processes to one core and reserved the other one for Firefox so that scheduling effects were reduced even further. It also meant that Firefox would not be swapped between cores by the kernel.
- 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. This prevented problems with slow hard drive access as explained in Section 3.1.4.

Using this setup we ran a test series again and compared the results with our previous results from Section 3.3. 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 3.1. Overall the results show a clear improvement, most of the performance differences have been significantly reduced. For example, the maximum difference for the `a11y` test went down from 3.38 % to 0.77 % and for `tsspider` it went down from 4.04 % to 2.58 %.

In order to give a better visual impression of how the results differ Figures 3.1 and 3.2 show a violin plot (Hintze and Nelson, 1998) 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.

One thing that is immediately obvious from the plots is that there are quite a few differences in effectiveness between the various tests. For example, the already mentioned improvement in the `a11y` test can clearly be seen, but the `dromaeo` tests look all very similar to their unmodified results. In other tests like `tgfx` and `tp_dist` the modifications got rid of all the extreme outliers. One very interesting result is that of the `v8` test. The curious shape and the result table do not really make it obvious, but after the modifications all of the results from the test had the same value – which is exactly what our ideal for all the tests would be. And there is another interesting observation that we can make: our table shows us that two tests, `ts` and `tsvg_opacity`, had a rather drastic *increase* in the max diff metric, but our plot makes it clear that this is due to a few extreme outliers while the rest of the results seem to have gotten better.

Ignoring variance for a moment it is also interesting to see whether our modifications made any difference on the absolute values of the results, that is whether they made the tests actually faster or slower. Figures 3.3 and 3.4 demonstrate that indeed there have been some changes, in some cases even seemingly significant ones, for example in the `tgfx` and `tsspider` tests. Interestingly enough some of the `dromaeo` tests seem to suffer a slight degradation of performance, though.

TABLE 3.1
Results after all external optimizations

	StdDev		CoV		Max diff (%) Absolute		Max diff (%) To mean		
Test name	nomod	cumul	nomod	cumul	nomod	cumul	nomod	cumul	Levene p-value
a11y	2.23	0.54	0.69	0.17	3.38	0.77	2.08	0.46	< 0.001
dromaeo_basics	4.41	2.39	0.53	0.29	2.57	1.40	1.62	1.01	0.028
dromaeo_css	11.36	7.95	0.30	0.21	1.39	0.89	0.88	0.46	0.314
dromaeo_dom	1.02	1.00	0.41	0.40	1.99	1.40	1.14	0.74	0.562
dromaeo_jslib	0.53	0.44	0.30	0.25	1.19	1.16	0.60	0.79	0.280
dromaeo_sunspider	5.65	3.77	0.54	0.36	2.09	1.33	1.16	0.74	0.086
dromaeo_v8	2.02	1.20	0.86	0.52	3.03	1.56	1.77	0.81	0.075
tdhtml	0.94	0.30	0.33	0.10	1.31	0.50	0.73	0.39	< 0.001
tgfx	0.80	0.14	5.68	1.37	25.60	5.63	18.88	2.93	< 0.001
tp_dist	1.77	0.19	1.16	0.14	4.42	0.62	3.30	0.35	0.002
tp_dist_shutdown	27.09	8.59	5.14	1.75	16.51	8.55	8.72	5.41	< 0.001
ts	2.27	2.46	0.59	0.74	2.45	3.68	1.66	3.26	0.282
ts_shutdown	7.28	3.75	2.00	1.19	6.88	4.86	3.44	2.89	< 0.001
tsspider	0.11	0.05	1.15	0.64	4.04	2.58	2.57	1.34	< 0.001
tsvg	1.43	0.68	0.04	0.02	0.17	0.09	0.10	0.05	0.006
tsvg_opacity	0.62	1.11	0.74	1.35	3.56	7.87	2.02	6.82	0.639
v8	0.11	0.00	1.42	0.00	4.31	0.00	3.59	0.00	0.008

nomod: unmodified setup; cumul: cumulative modifications
New results with a statistically significant difference are shaded grey.

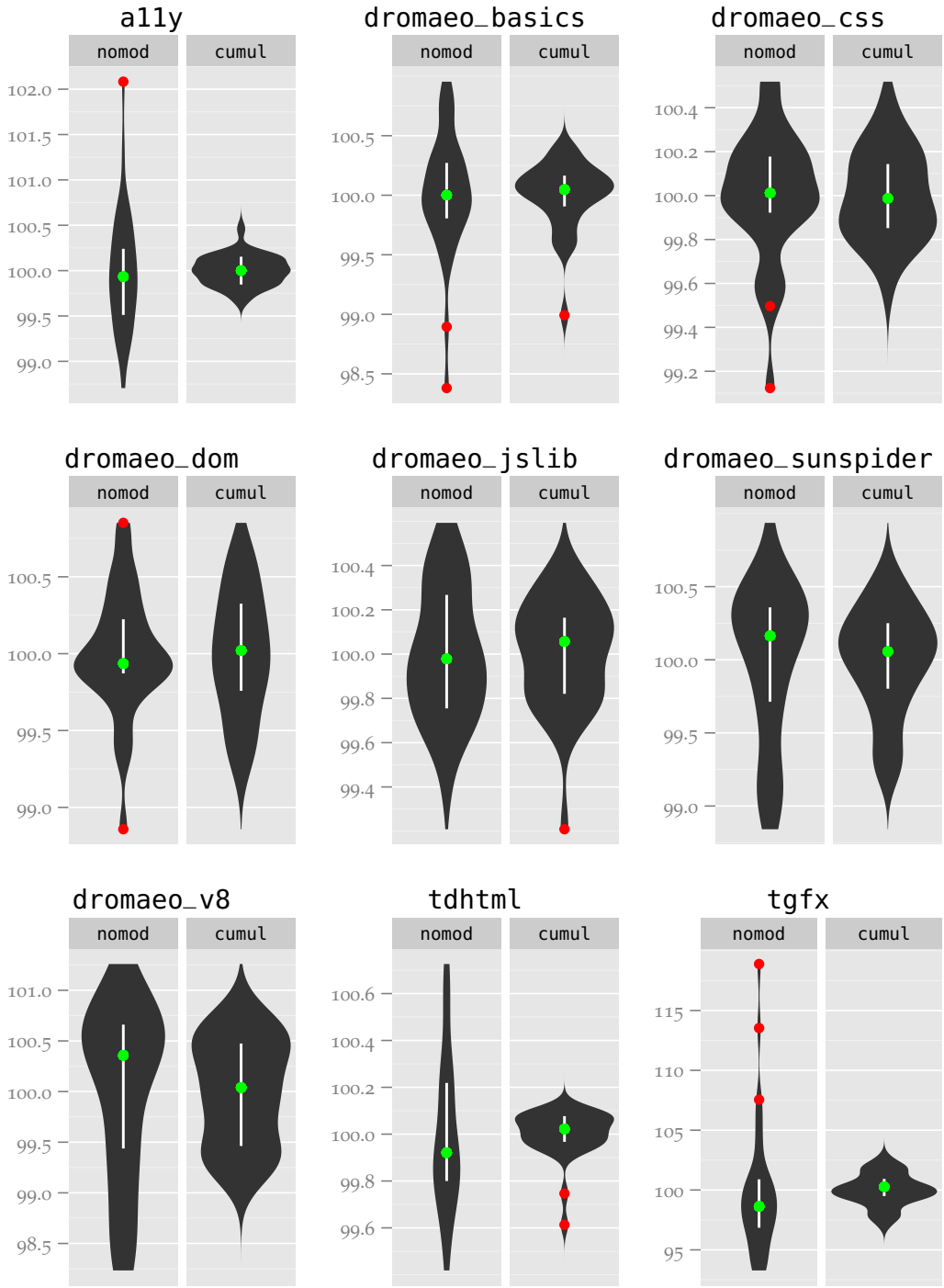


Figure 3.1: The first half of the tests after external optimizations, displayed as the percentage of their mean

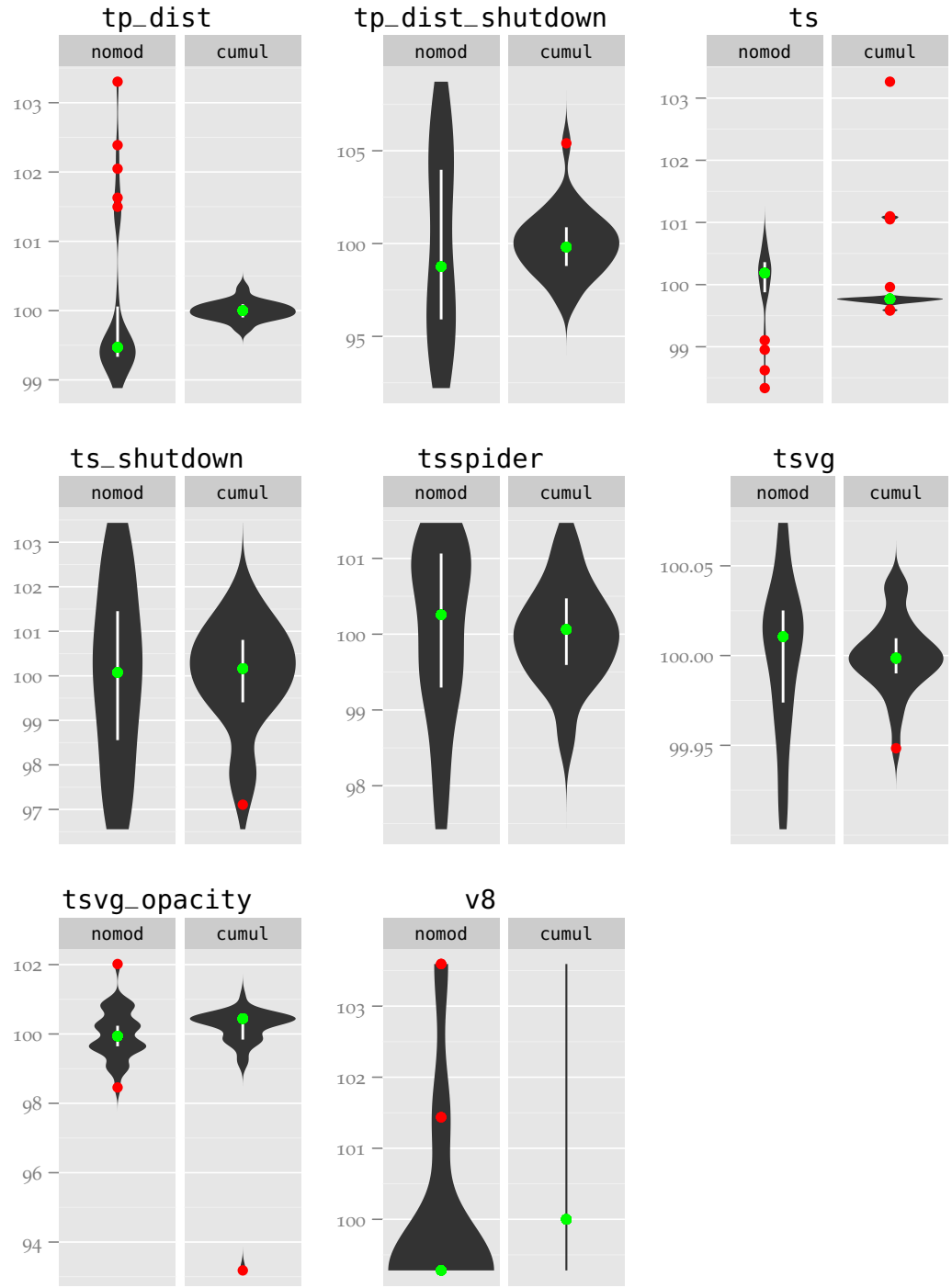


Figure 3.2: The second half of the tests after external optimizations, displayed as the percentage of their mean

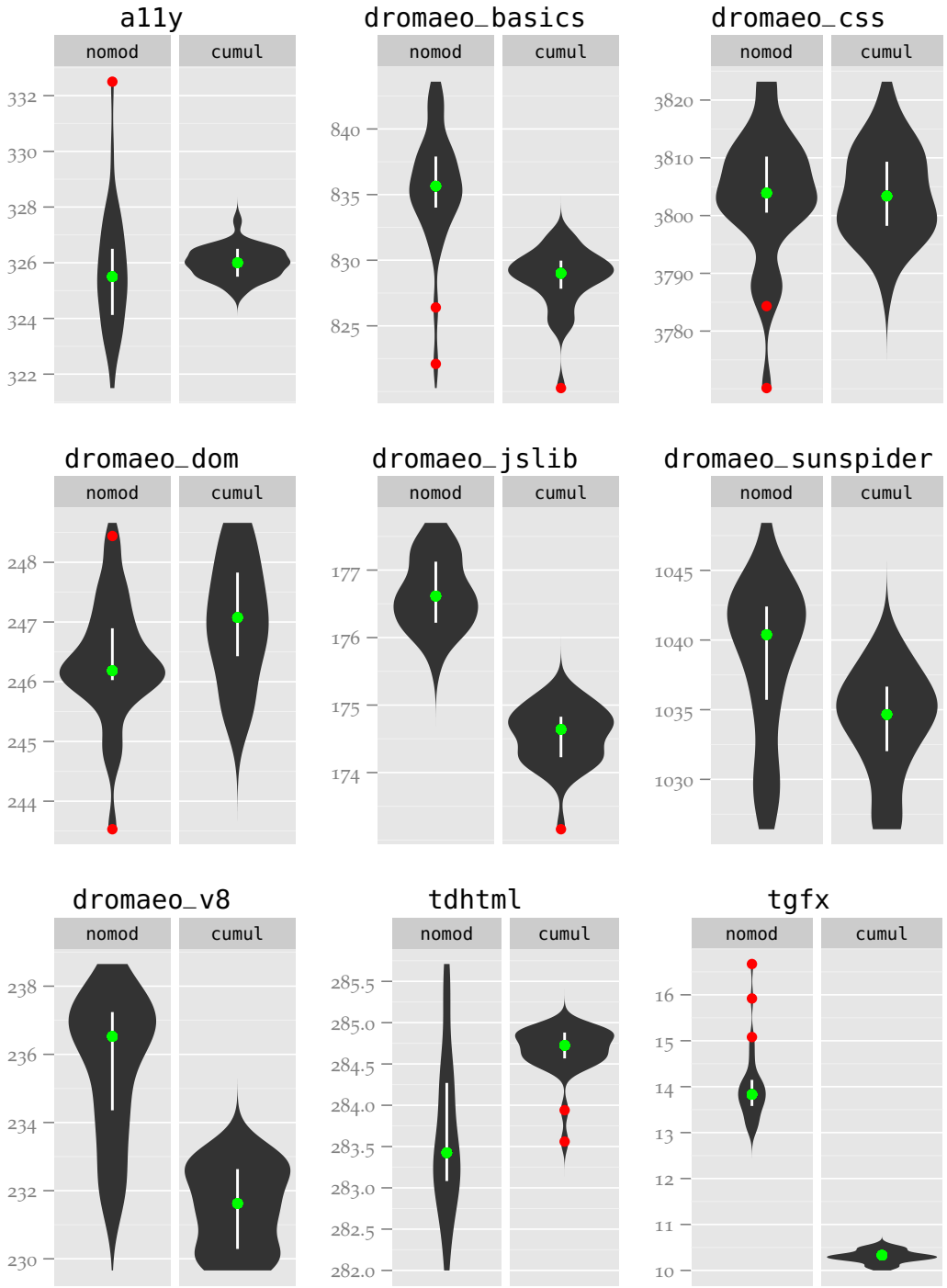


Figure 3.3: The first half of the tests after external optimizations, absolute result values

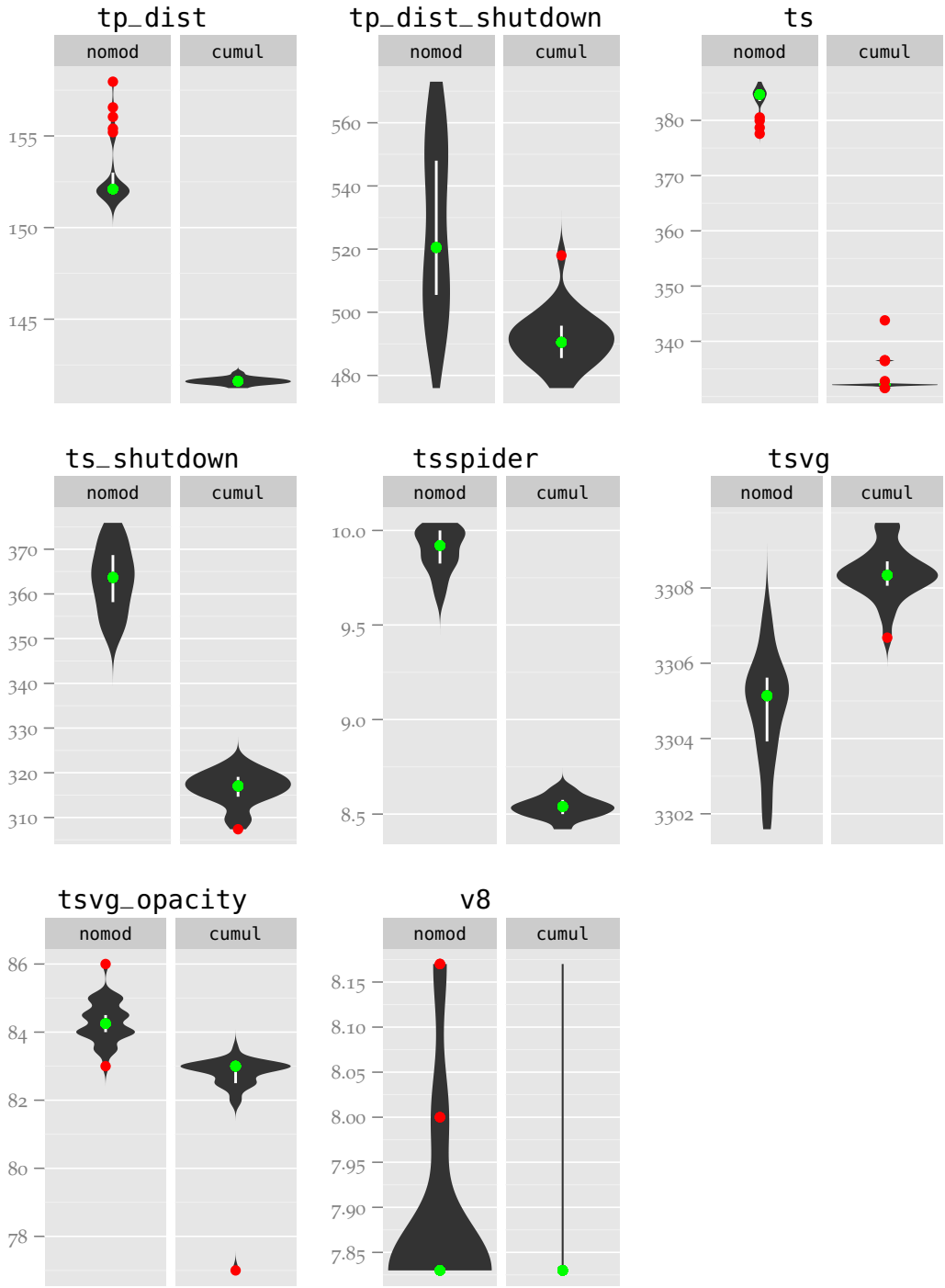


Figure 3.4: The second half of the tests after external optimizations, absolute result values

3.3.1 THE LEVENE TEST

Now we have a nice visual representation of the differences between our two setups. But looks can be deceiving – can we really be sure that the differences we see are actually *statistically significant*, that is in more technical terms whether the two samples from our tests come from *different distributions*? This is where we can make use of the *Levene test for the equality of variances* (Levene, 1960; Brown and 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 our initial analysis (see Sections 2.4 and 2.5.2) shows that not all of the tests necessarily follow a normal distribution the test will still be valid.

Table 3.1 shows the resulting p-value after applying the Levene test to all of our test results. Using the standard significance level of 0.05 again the results confirm our initial observations: Almost all of the tests have a very significant difference, except for most of the `dromaeo` tests and the `ts` (startup) 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 3.2 shows the results of comparing the isolated parameters to the

TABLE 3.2
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

Statistically significant values are shaded grey.

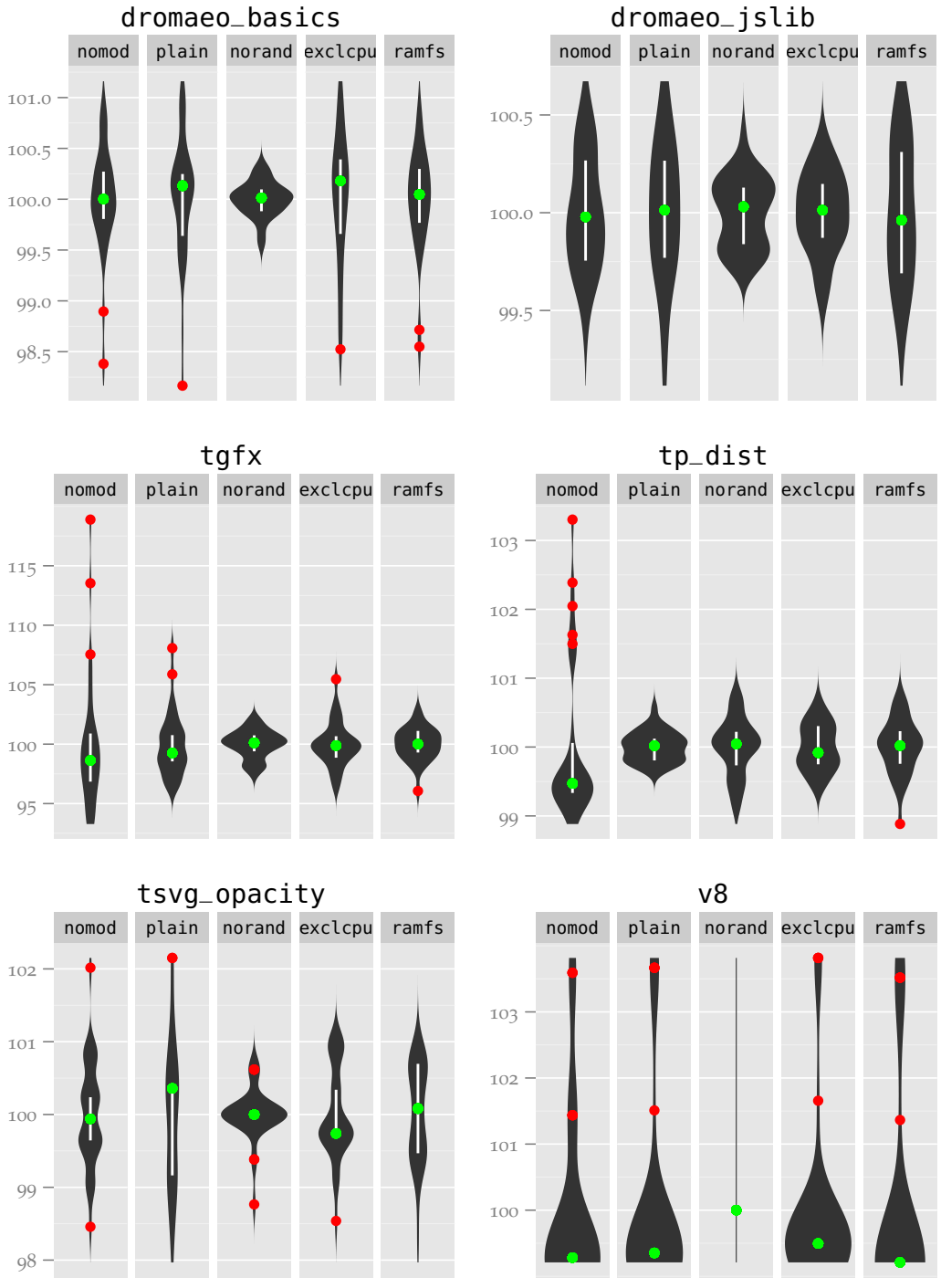


Figure 3.5: Some of the results from isolated modifications

unmodified version using the Levene test, and Figure 3.5 a few interesting examples of the distributions. 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 means 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 especially the `tp_dist` case all of the modifications have an influence on the outliers, especially in the latter test.

These finding about the memory randomization mirror the results of the papers from Section 2.6.1 in that the memory layout is a major contributing factor to variance due to aspects like alignment and prefetching, even if it cannot explain all of it.

In order to test what factors exactly were responsible on a lower level we had planned on using *hardware performance counters*, similar to Gu et al. (2004). Unfortunately both performance counter libraries that are available for Linux, Rabbit[2] and PCL[3], have not been updated in years and are not compatible with current Kernels or even current processors. Recent Kernel versions have support for a new hardware performance counter framework, but so far there are only stand-alone tools available that can make use of it. This was not useful in our case since we are only interested in the data from the period when the actual tests run inside of the browser, not from the whole program lifetime. With the stand-alone tools restricting the data gathering to this period would not have been possible.

One thing to note is that even with disabled memory randomization there can still be variance between *different* versions of a program if there

[2] `http://www.scl.ameslab.gov/Projects/Rabbit/`
[3] `http://berrendorf.inf.h-brs.de/PCL/PCL.html`

TABLE 3.3
Levene p-values for comparing the cumulative modifications with isolated ones

Test	plain	norand	isolcpu	ramfs
a11y	< 0.001	< 0.001	< 0.001	< 0.001
dromaeo_basics	0.017	0.156	0.001	0.027
dromaeo_css	0.020	0.545	0.174	0.027
dromaeo_dom	0.420	0.006	0.462	0.603
dromaeo_jslib	0.050	0.264	0.523	0.025
dromaeo_sunspider	0.213	0.490	0.055	0.111
dromaeo_v8	< 0.001	0.343	0.048	0.194
tdhtml	< 0.001	< 0.001	0.002	< 0.001
tgfx	0.008	0.905	0.064	0.302
tp_dist	0.012	< 0.001	< 0.001	< 0.001
tp_dist_shutdown	< 0.001	< 0.001	< 0.001	< 0.001
ts	0.014	0.113	0.860	0.0625
ts_shutdown	0.134	0.049	0.663	< 0.001
tsspider	0.012	1.000	0.348	0.3739
tsvg	0.003	0.077	< 0.001	0.002
tsvg_opacity	0.195	0.202	0.939	0.750
v8	0.023	NaN	0.046	0.011

Statistically significant values are shaded grey.

are slight differences in the environment or other areas as Mytkowicz et al. (2009) observed. The effect of this would be similar as if an unchanged program were to be run with enabled randomization. The only way to guard against that would be to run each version of the program multiple times with enabled randomization and then take an average of the results. Note that this is not the same thing as the "internal" test repetitions that are already being done as part of the tests as those are all run within the same instance of a program and thus are not affected as much by the randomization.

So was our `norand` modification the only one that actually resulted in a significant change? Unfortunately, no. Table 3.2 shows that all of the modifications have at least some significant differences, in the case of `exclcpu` even in tests that have no significant difference for `norand`. Table 3.3 shows

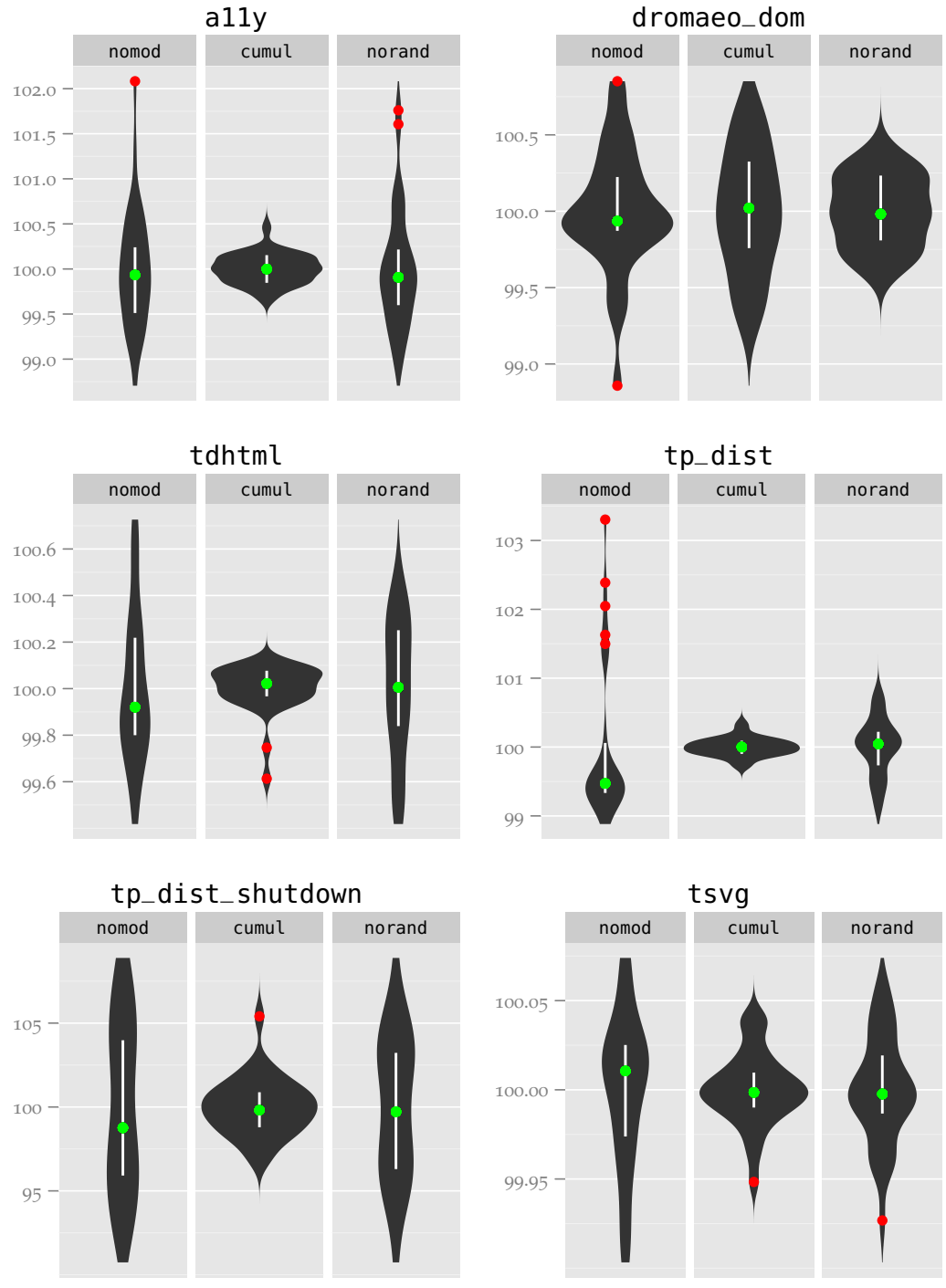


Figure 3.6: Comparison of the cumulative modifications with only `norand`

that there are still several significant differences between the isolated `norand` setup and the cumulative one. The fact that the number of differences is smaller with the `norand` setup than with the others indicates that it is the largest contributor, though, even if it is not responsible for all of them. Figure 3.6 shows a few comparisons between the unmodified setup, the cumulative modifications and the isolated `norand` one.

The complete plots for all of the tests are available in Appendix B.

3.5 SUGGESTIONS

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 variance is caused by the external factors investigated in this chapter. This leads to the following suggestions, taken from the way our experiments were set up:

1. Address space randomization should be disabled. Since the test machines should not be directly accessible through the internet anyway this should pose no additional security risk. As mentioned above this had the most significant effect on the variance, so if only one of the changes could get implemented this should be it.

2. Test machines should run only the most essential processes while testing. As a graphical application Firefox needs at least an X server and a terminal that the test suite can be run from, but apart from that only some system services should be needed. Not needed are things like graphical login managers, servers and `cron`-like scheduling programs unless those services are necessary to interact with the test machines or the test suite.

3. In case the machines possess more than one CPU (core), the processes should be segregated into the Firefox process and all the other processes on the separate CPUs to minimize scheduling interference.

4. Tests should always be run from a RAM disk. Since both the test suite and the result logs are relatively small this should pose no problem as far as RAM size is concerned. The Firefox binary along with its libraries is a bit larger, but still not really enough to create a real problem. Note that a RAM disk type should be used that will never be swapped out to disk (on Linux the file system type `ramfs` is suitable for this).

Even with the significant improvements from this chapter 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.

CPU TIME, THREADS & EVENTS

4

After dealing with external influences in the last chapter 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.

4.1 THE XPCOM FRAMEWORK

The xpcom (Cross Platform Component Object Model) framework is a component object model similar to corba[1] (and is in fact partly derived from it), which has the goal of abstracting away many implementation details to improve cross-platform compatibility. It essentially allows to specify interfaces in a special Interface Description Language (idl) that can then be implemented and used by a variety of languages. This is for example used to allow JavaScript to call C++ methods and for the events that are used for the internal work (described in more detail below). Classes that implement one of those interfaces are called *components*.

4.2 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. Therefore it would make sense to only measure the time that our program is actually running: the cpu time. This will get rid of both the time during which other processes are running and of the time needed for context switches. Since we are running Firefox exclusively on one processor the former should not be much of an issue except when Firefox and some other

[1] `http://www.corba.org/`

process are trying to use the same shared resource, but the latter will be if there is a different number of context switches between threads.

4.2.1 EXPERIMENTAL SETUP

Our setup consisted of two parts: a custom XPCOM component that could report the CPU time using a system call and a modification to the Talos framework that would use this component instead of the wall-clock time.

The XPCOM component Our component was essentially a wrapper around the system call `clock_gettime()`. This function can report the values of several timers; here we are specifically interested in the one named CLOCK_PROCESS_CPUTIME_ID, which reports the time the current process has been running so far in nanoseconds.

The Talos modification The Talos framework normally records the current time before starting to load a page and after the loading has finished using the JavaScript `Date.now()` function which reports the number of milliseconds since the start of the UNIX epoch (1970-01-01 00:00:00). Our modification made it use our XPCOM wrapper instead. Since only the difference between the two time points is of interest the different reference points did not matter for us.

Unfortunately only a few tests make direct use of the time that the Talos framework gathers in this way, 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. However, the results should still give an indication of whether the difference in time-keeping leads to significant changes in principle or not.

4.2.2 RESULTS

Table 4.1 shows the results from a test series compared to the externally optimized results from the previous chapter. The statistically significant differences are highlighted in grey again. We can see that from the four

TABLE 4.1
Results after the CPU time modification, compared to the externally optimized results from Chapter 3

	StdDev		CoV		Max diff (%) Absolute		Max diff (%) To mean		
Test name	cumul	cputime	cumul	cputime	cumul	cputime	cumul	cputime	Levene p-value
a11y	0.54	0.81	0.17	0.25	0.77	0.92	0.46	0.56	0.106
dromaeo_basics	2.39	2.05	0.29	0.24	1.40	1.01	1.01	0.56	0.949
dromaeo_css	7.95	8.52	0.21	0.22	0.89	0.91	0.46	0.48	0.887
dromaeo_dom	1.00	0.83	0.40	0.33	1.40	1.47	0.74	0.74	0.217
dromaeo_jslib	0.44	0.31	0.25	0.18	1.16	0.79	0.79	0.44	0.111
dromaeo_sunspider	3.77	3.17	0.36	0.31	1.33	1.29	0.74	0.71	0.384
dromaeo_v8	1.20	1.62	0.52	0.70	1.56	1.96	0.81	0.99	0.009
tdhtml	0.30	0.35	0.10	0.12	0.50	0.50	0.39	0.28	0.172
tgfx	0.14	0.14	1.37	1.28	5.63	5.42	2.93	2.88	0.834
tp_dist	0.19	0.27	0.14	0.18	0.62	0.75	0.35	0.39	0.197
tp_dist_shutdown	8.59	8.40	1.75	1.70	8.55	7.07	5.41	4.10	0.747
ts	2.46	1.84	0.74	0.55	3.68	2.71	3.26	2.27	0.769
ts_shutdown	3.75	4.39	1.19	1.37	4.86	6.36	2.89	3.32	0.479
tsspider	0.05	0.09	0.64	1.02	2.58	4.55	1.34	2.30	0.016
tsvg	0.68	0.90	0.02	0.03	0.09	0.10	0.05	0.06	0.163
tsvg_opacity	1.11	1.51	1.35	1.88	7.87	7.46	6.82	3.81	0.005
v8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	NaN

New results with a statistically significant difference are shaded grey.

tests that should be affected by our changes only one, `tsvg_opacity`, does have a significant difference, and the variance actually seems to have gotten worse.

Figure 4.1 again displays the results visually. We can clearly see that the variance in the `tsvg_opacity` test got much worse, except for an outlier in the previous results. The other results do not look much different, but they still show that there is clearly no improvement in any of them. 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.

The complete plots are available in Appendix B.3.

4.3 INTRODUCTION TO THREADS & EVENTS

The main job of a web browser is undeniably to display web pages, and do so in an efficient way. Putting it like that makes the task sound reasonably easy, but, unfortunately, things tend to be more complicated than they look at first. In the case of web browsers in general, and Firefox in particular, there are more things to consider than just the loading of a single web page. For example, the user interface (UI) should still respond to user actions like trying to open a menu, even if a web page is still loading at the same time. In other cases a user might have several pages open at the same time, some of which haven't finished loading or are continually running some JavaScript code, and the user then wants to interact with another page or open a new tab. All these scenarios require that several tasks need to be able to run in parallel, at least from a user perspective, not unlike how multitasking operating systems work.

On the application side there are essentially two approaches to this problem: use multiple threads or split the tasks into small, interruptible units that can be executed out of order.

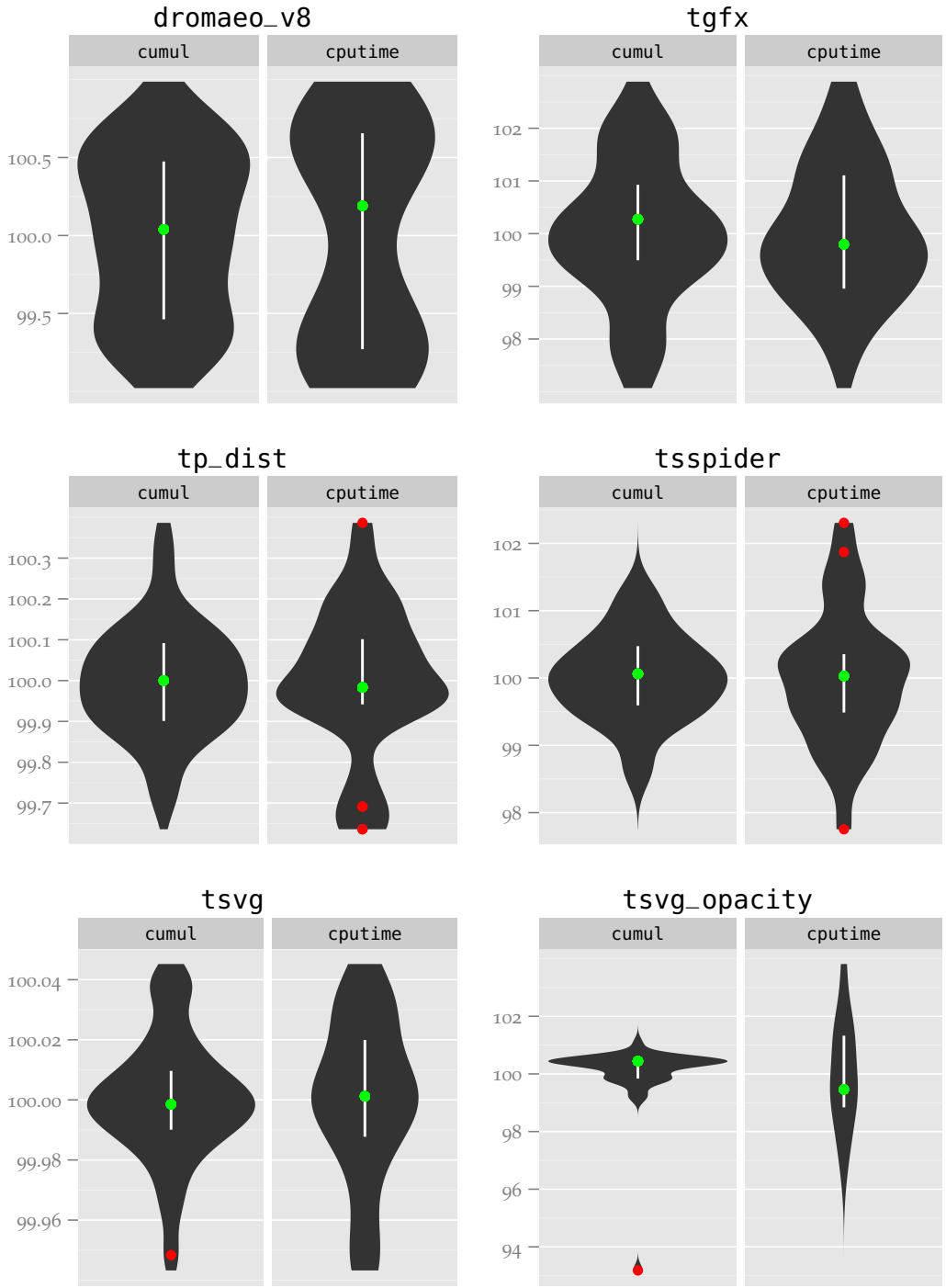


Figure 4.1: Comparison of the external modifications with the CPU time modifications

4.3.1 THREADS

Threads are a popular solution in these cases. They allow asynchronous execution and hand the responsibility of scheduling them off to the operating system. In addition, they can be put onto different processors (or processor cores), potentially increasing performance if a lot of expensive, non-communication-heavy processing is required. However, they also have serious drawbacks. Since threads share memory there must be a way to regulate concurrent access to prevent race conditions. This usually involves mechanisms such as locks, semaphores and/or monitors (Tanenbaum, 2001). Unfortunately these mechanisms are both hard to get right, with the potential of deadlocks and other hard to find bugs, and do not scale well to more than a few cores. In addition there is usually a significant amount of communication necessary between a web page and the UI, so separating that into different threads would be difficult and potentially even decrease performance.

4.3.2 EVENTS

For these reasons Mozilla for the most part went with the second approach: dividing work up into small units and executing them on just one thread. These "work units", called *events*, can range from very small, like a simple message-passing equivalent that just sets one variable (example: `nsThreadShutdownAckEvent`, which simply acknowledges that the shutdown event has been processed), to rather complex (`nsPreloadURIs`, which pre-loads pages linked to from the current page in order to decrease the time needed to load the page if a user clicks on the link later on).

Each newly created event will get dispatched to an *event queue* where it will then get picked up to get executed. That way events that are used for rendering a web page and user interface events can be freely mixed and allow for a responsive UI even when the browser is in the process of loading a page, without the difficulties of threads. Another advantage of this model is that it allows for *incremental page load*, meaning that a web page is displayed incrementally as soon as an element has finished rendering

and then *reflowed* (meaning, from a user perspective, that the layout gets adjusted) once other elements have been added. This way a user does not have to wait for the complete page to load before they can see anything.

4.3.3 THREADS AGAIN: THE THREAD POOLS

Despite the event model described in Section 4.3.2, threads *are* used in Firefox's XPCOM framework for a few things, most notably asynchronous operations like I/O and statement execution in the SQLite databases that are used for bookmarks and the history. These threads typically get started early on during startup and exist throughout the entire lifetime of Firefox.

Additionally Firefox uses the concept of a *thread pool*. This is a pool of anonymous threads that exists purely to execute occasional events in an asynchronous manner without having to keep a specific thread alive for them all the time. These thread pools (there can be more than one that are used from different parts of the code) work in the following way:

1. An event gets put into the pool's event queue.
2. If there are no idle threads and the current number of thread pool threads is smaller than the maximum number allowed, a new thread gets created.
3. If a new thread has been created, it gets added to the thread pool thread list.
4. All thread pool threads are notified via a monitor so they can check the event queue.
5. One of the threads picks up the event and executes it.
6. If a thread has been idle for longer than a specified timeout, or if there are more idle threads than allowed, that thread gets shut down.

This setup allows for the easy handling of asynchronous operations without the thread posting the event having to worry about the details.

However, for our purposes it is less than ideal. The rather quick shutting down of threads (due to a low default timeout) can lead to a situation where an event that gets posted to a thread pool may in one case arrive just before a thread is supposed to get shut down and will thus reuse this thread. In another case, say in a different run of the same test, that event may – due to tiny scheduling differences in the operating systems – arrive at the thread pool shortly after the thread has been shut down, requiring for a new thread to be created. This thread creation (and destruction), while not as expensive as process creation, still incurs a cost that could lead to measurable variance in the test suite. Initial tests showed that there were indeed a different number of threads being created by the thread pools, so this issue was certainly worth investigating.

4.4 INVESTIGATING THREAD POOL VARIANCE

4.4.1 EXPERIMENTAL SETUP

In order to analyse the impact of the threading issues described above we modified the Firefox source code to only ever create one thread per thread pool, and increased the timeout to a value that guarantees that the thread will be kept alive throughout the whole lifetime of the process. This had the potential to reduce the absolute performance of some of the tests, since now the order of events mattered more and unrelated events would have to wait for each other. However, since we are only interested in the variance, this was an acceptable risk. We then ran a test series again using the same setup as explained in Chapter 3.

4.4.2 RESULTS

Table 4.2 shows the results from our thread pool modification experiment. Unfortunately we can immediately see that only two of the tests have a significant difference, and again the variance has actually gotten worse instead of having improved as hoped.

The plots in Figure 4.2 illustrate this. In both tests the density has moved

Table 4.2
Results after the thread pool modification, compared with the CPU time modification from Section 4.2.2

	StdDev		CoV		Max diff (%) Absolute		Max diff (%) To mean		
Test name	cputime	tp1	cputime	tp1	cputime	tp1	cputime	tp1	Levene p-value
a11y	0.81	0.63	0.25	0.19	0.92	0.93	0.56	0.53	0.418
dromaeo_basics	2.05	2.06	0.24	0.24	1.01	0.88	0.56	0.49	0.856
dromaeo_css	8.52	8.83	0.22	0.23	0.91	0.92	0.48	0.51	0.625
dromaeo_dom	0.83	1.24	0.33	0.50	1.47	1.64	0.74	0.89	0.002
dromaeo_jslib	0.31	0.45	0.18	0.26	0.79	1.21	0.44	0.74	0.092
dromaeo_sunspider	3.17	2.37	0.31	0.23	1.29	0.91	0.71	0.62	0.250
dromaeo_v8	1.62	1.40	0.70	0.60	1.96	1.88	0.99	1.21	0.123
tdhtml	0.35	0.35	0.12	0.12	0.50	0.53	0.28	0.27	0.926
tgfx	0.14	0.18	1.28	1.69	5.42	6.28	2.88	3.15	0.026
tp_dist	0.27	0.28	0.18	0.19	0.75	0.89	0.39	0.53	0.977
tp_dist_shutdown	8.40	13.24	1.70	2.68	7.07	14.36	4.10	10.21	0.649
ts	1.84	1.30	0.55	0.39	2.71	1.86	2.27	1.52	0.994
ts_shutdown	4.39	3.78	1.37	1.17	6.36	4.73	3.32	2.38	0.595
tsspider	0.09	0.09	1.02	1.01	4.55	4.33	2.30	2.72	0.895
tsvg	0.90	1.16	0.03	0.04	0.10	0.14	0.06	0.07	0.077
tsvg_opacity	1.51	1.48	1.88	1.81	7.46	7.99	3.81	4.51	0.789
v8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	NaN

New results with a statistically significant difference are shaded grey.

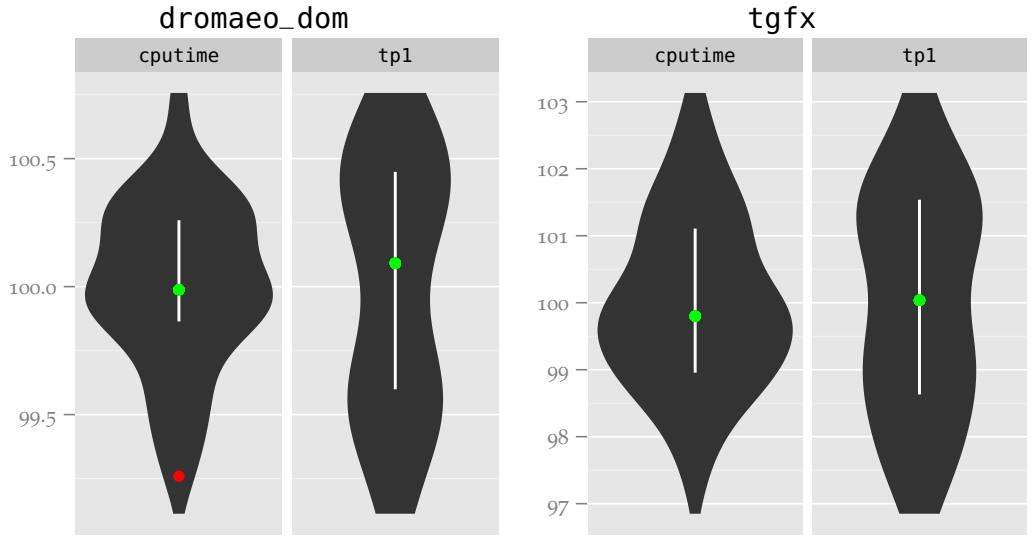


Figure 4.2: Comparison of the CPU time modifications with the thread pool modification

from roughly around the mean to two "bulges" further away, and the interquartile range of the `dromaeo_dom` test is about twice the size as the old one. This is clear evidence that the variance we are looking for is not caused by thread activity surrounding the thread pool, and the slight increase in variance might be caused by a performance degradation due to the reduced number of threads.

Again, the complete plots are available in Appendix B.4.

4.5 EVENT VARIANCE

As mentioned above, events are the main mechanism by which work is done in Firefox. This leads to an interesting question: is the same work, for example a test in our test suite, always done using the exact same events, or can it be done in different ways? And if yes, could this be the cause for the variance we are seeing? For this we have to take a look at what events are executed during a test and check whether there is any correlation between the most important event properties and the variance. In concrete terms we

are going to look at two properties specifically: the number of events, and their order of dispatch.

4.5.1 EXPERIMENTAL SETUP

Events are classes that inherit from the XPCOM interface `nsIRunnable`, which declares the single method `Run()`. This poses a problem for us: we do not have a way to identify the different classes of events, since there is no public method to inspect them and run-time type information (RTTI) is not used in Firefox. We therefore need a different way of identifying them, ideally without having to modify every single event class. The way we solved that problem in our experiments is to generate backtrace information at the time when an event is dispatched to an event queue, showing us exactly where an event comes from, which is even more information than what a normal class identification would have given us.

Since we are only interested in the events that are used during the actual tests, we also again used our custom XPCOM component and a modification to the Talos framework to print out a special message at the moments when the test starts and when it finishes so that we can separate the events we are interested in from the others.

Using these modifications we again ran a test series, with the only difference that we used only 5 distinct runs. This was due to the size of the generated log files and the time it took to run our analysis scripts afterwards.

4.5.2 RESULTS: NUMBER OF EVENTS

Figure 4.3 shows an example of what the result of an event number analysis of our log files looks like. Each line represents a single event, with the string at the beginning being a hash of the complete backtrace and the numbers signifying the number of times this event occurred during each of the five runs. An exclamation mark is printed after the hash if the number of events differs between the runs, and at the end the sum of all the event numbers is printed.

```
1  e19fd78b2439bcbb55d5 !    11      11      11      12      11
2  91b1112a65d131c0a537        3       3       3       3       3
3  5aee85cd567f628853de        1       1       1       1       1
4  20162c32c9092813e7b0        3       3       3       3       3
5  535510a64010a2e38bf7        1       1       1       1       1
6  becd6cc1818bf0ff8d14        6       6       6       6       6
7  c0df172143e27468f0b7 !      0      35       0       0      35
8  a0b44c64541919647ae0        6       6       6       6       6
9  a93dc981b861c0cc9821 !      0       0       0       1       0
10 3dd2cc7672da14088339       12      12      12      12      12
11    .
12    .
13    .
14
15 Sum:                    2075    2109    2076    2095    2107
```

Figure 4.3: Simplified example of an event number log after analysis

Using this information we can indeed see that there is variance 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, like for example the one in line 7 in Figure 4.3, 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.

The interesting question is now whether this event variance is in any way related to the variance we are seeing in the test results. For this we need to do a *correlation analysis*. We used the *Pearson product-moment correlation coefficient* (Rodgers and Nicewander, 1988), a well-established technique to measure the correlation between two variables. In this first analysis the two variables are straightforward: the number of events for each run and the corresponding test results, and the null hypothesis is that there is no correlation between the variables.

The results of the correlation analysis are shown in Table 4.3. The coeffi-

TABLE 4.3
Correlation analysis for the total number of events

Test name	Coefficient	Pearson p-value
a11y	0.19	0.763
dromaeo_basics	−0.05	0.933
dromaeo_css	0.30	0.623
dromaeo_dom	0.16	0.793
dromaeo_jslib	0.36	0.554
dromaeo_sunspider	0.76	0.135
dromaeo_v8	0.41	0.492
tdhtml	−0.16	0.800
tgfx	0.95	0.012
tp_dist	0.97	0.033
tsspider	−0.18	0.824
tsvg	0.20	0.796
tsvg_opacity	−0.76	0.236
v8	—	—

Statistically significant values are shaded grey.

cient indicates the way in which the variables are correlated to each other: a positive value means that as x (the number of events) increases y (the result of the test) increases as well, with 1 indicating a perfect line through all of the points. A negative value means that y decreases as x increases, again with -1 indicating a perfect line through the points. Values near zero mean that there is no correlation between the variables. As before we also calculated the statistical significance of the analysis.

As we can see in the table only for two of the tests is there a correlation between the test results and the number of events. The other tests are quite far away from any statistical significance, indicating that in general the number of events is unrelated to the test result, even though it does vary.

4.5.3 RESULTS: ORDER OF EVENTS

The second analysis we used was on the *order* of events. We were interested in determining whether for example a large number of differences in the

```
1          949d991aa93da7c011ae   949d991aa93da7c011ae
2          949d991aa93da7c011ae   949d991aa93da7c011ae
3                                 > fa11c0338f6f80ea22b2
4                                 > 5d6d1725bf28309d1969
5                                 > 52339ce63644fd59cb4b
6          211460b9bb9aa2d25270   211460b9bb9aa2d25270
7          7aa5349ab6cdb364f723   7aa5349ab6cdb364f723
8          7aa5349ab6cdb364f723   7aa5349ab6cdb364f723
9          5d6d1725bf28309d1969 <
10         52339ce63644fd59cb4b <
11         525a50883625fca6e9eb   525a50883625fca6e9eb
12         54b2cc584af6cd076f74   54b2cc584af6cd076f74
13         a036d8999956f9249846   a036d8999956f9249846
14         d91ebb7d5101277b7177   d91ebb7d5101277b7177
15         d91ebb7d5101277b7177   d91ebb7d5101277b7177
16         7aa5349ab6cdb364f723   7aa5349ab6cdb364f723
17         fa11c0338f6f80ea22b2 <
18         aeafb3e54beb9751f54f   aeafb3e54beb9751f54f
19         e63578d0f3549b8de07a   e63578d0f3549b8de07a
```

Figure 4.4: Simplified example of an event order log after analysis

event order also resulted in a big difference between the respective test results, indicating that some event orderings are more favourable than others. For this we took all of the combinations of the runs in our test series and computed the difference in the order of events, similar to a standard `diff` algorithm, and the difference between the test results, and again ran a correlation analysis on these two variables.

Figure 4.4 shows an example of what such an event order diff between two runs looks like, again with the string representing the hash of the specific event. We can see that indeed some events appear out of order, even though most of them are in the same order in both runs. Also, some of the out-of-order events seem to depend on others, like the events on line 4 and 5 which show up at a different place in both runs but in the same order relative to each other. Other events seem not to be dependent on others; the event on line 3 appears before the just mentioned two events in the second

TABLE 4.4
Correlation analysis for the order of events

Test name	Coefficient	Pearson p-value
a11y	−0.07	0.858
dromaeo_basics	0.01	0.978
dromaeo_css	−0.19	0.597
dromaeo_dom	0.06	0.860
dromaeo_jslib	−0.02	0.955
dromaeo_sunspider	0.58	0.079
dromaeo_v8	0.22	0.543
tdhtml	−0.09	0.813
tgfx	0.14	0.699
tp_dist	0.98	< 0.001
tsspider	0.08	0.887
tsvg	0.44	0.386
tsvg_opacity	0.71	0.113
v8	—	—

Statistically significant values are shaded grey.

run but a while after them in the first.

Looking at what kinds of events routinely occur out of order, we found that many of them share a common theme: they are dependent on external or at least asynchronous factors. The following list gives a few examples:

- `nsInputStreamReadyEvent` is responsible for asynchronous I/O.
- `mozilla::storage::AsyncExecuteStatements` utilizes a separate thread to execute statements in the databases that are used for storing bookmarks and other information.
- `nsTimerImpl::PostTimerEvent()` provides access to various hardware timers and is thus dependent on when those timers fire which is out of the control of the Firefox process.

The results of the correlation analysis are shown in Table 4.4. Here we only have one test with a statistically significant difference – the `tp_dist`

test, which was also one of the only two significant ones in the event number analysis. Since this test is by far the most long-running one due to the number of pages it loads we suspect that the length of the test has an impact on how well the events and the test results are correlated – possibly the connection is drowned out by unrelated factors in the other, shorter tests.

Unfortunately, these results mean that – except possibly for the `tp_dist` test – there is no direct correlation between event properties and the test results, so it seems like the events are not directly responsible for the variance we see in the test results.

FORECASTING

5

As the previous chapters have shown, it is not reasonably possible to eliminate all potential variance in our performance tests. This still leaves us with our original problem, though: how do we determine whether a new test result signifies a genuine change in performance or is just noise. If we cannot reduce the noise itself, is there maybe a way to determine the type of a new value based on the previous ones, that is figure out whether it fits into the current trend? There is, to a certain extent.

Note that there are a few differences between the series we used in the previous chapters and the ones we will be looking at here: in this chapter we will use data from the official Mozilla test servers that form a *time series* (meaning they are based on different points in time with actual changes in between, something that our own data intentionally did not have) instead of our self-generated data in order to have a mixture of noise and real performance changes to test our models on.

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 while the new code is in place.

Phrasing it like that indicates 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 and of the 5 runs starting from it, that is create two *moving averages*.

2. Use a *t*-test to determine whether the difference between the means is statistically significant.

Like with our question of how many runs to use in a test series as described in Section 2.5.1, there is an inherent trade-off involved in deciding how many results should be used for the means calculations. In the case of the so-called *back-window*, that is the window that goes back from the current result, a too big one would mean that larger, genuine performance changes would distort the mean in a way that it no longer represents the most recent performance that we are trying to compare our new results to, and a window that is too small would put too much emphasis on short-term noise. The number 30 that the Mozilla developers chose seems to be a reasonable compromise between these conflicting requirements.

For the *fore-window* the requirements are slightly different: we still have the problem of putting too much emphasis on noise if we choose a small window, but more importantly we want to find a regression as soon as possible so the code changes that are responsible for it can be reversed without too much trouble. In addition short performance spikes could go unnoticed if they get "lost" in a long series of normal results. Again, the value of 5 should work reasonably well in this case.

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.

Now that we have our two windows, how do we determine whether the difference between their means is actually significant enough, that is whether it can be attributed to genuine performance changes? This is where the hard statistics comes in. Determining the significance of a difference in means is a well-established field, and the method that is appropriate in our case is the so-called *t-test*. A t-test is essentially a special case of an ANOVA analysis for finding the difference in means between two or more groups, as the t-test only works with exactly two groups – which is what our two windows are – and one factor of interest, that is the test result in our case. To be more specific we use a variation of the t-test for cases with independent samples (i.e. an *unpaired* test), unequal sample sizes and potentially unequal variance called *Welch's t-test*. The test statistic t is computed in the following way:

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

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[1]. However, this degree of freedom has to be computed

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

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 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.

An obvious 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:

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

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[2].

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 and Chatfield (1990) developed a technique for that using the assumption that the underly-

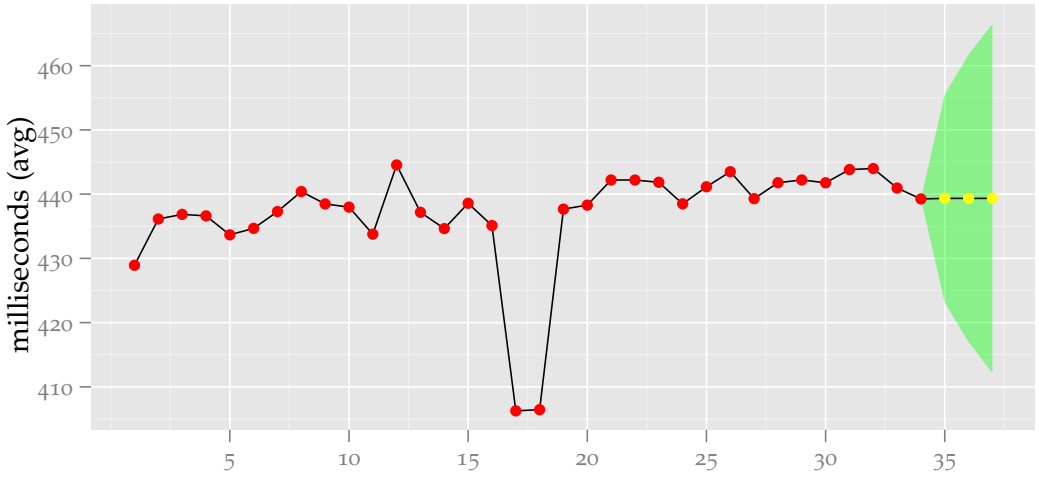
[2]See for example `http://stat.ethz.ch/R-manual/R-patched/library/stats/html/HoltWinters.html`.

ing statistical model of exponential smoothing is the ARIMA (autoregressive integrated moving average) model, calling the intervals *prediction intervals*.

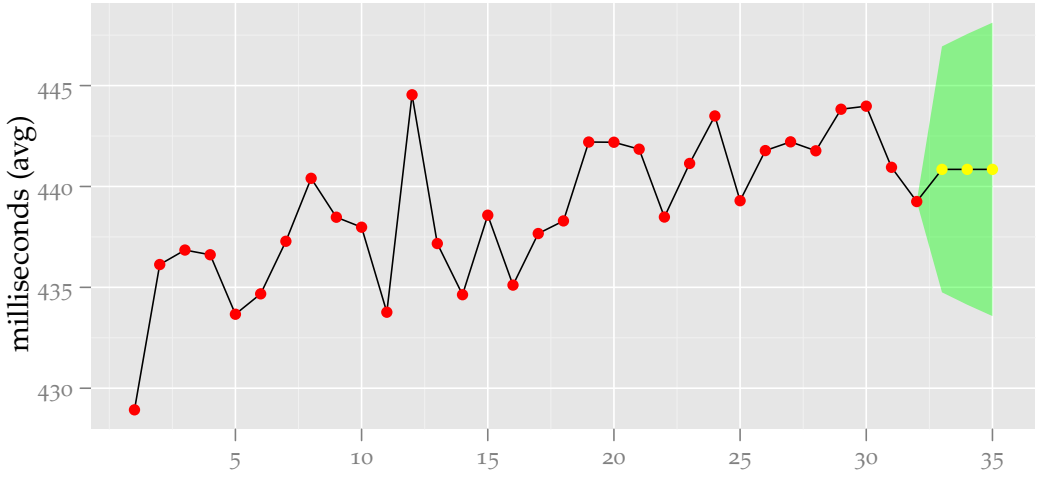
Figure 5.1a shows an example from the `tp_dist` test with official test server data and the 95 % prediction interval for the next three values. We used three here to make the interval easier to identify, but in practice only one would be needed.

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 5.1b 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 – even several of the values from the first third would now lie outside of them, demonstrating that they do not have much influence any more. 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.

An important thing to note is that the official test results form an *irregular* time series, that is the values were taken at irregular intervals in time – usually when a new version was committed to the main repository, which is of course very much random. However, prediction with exponential smoothing only works on *regular* time series, where all the distances between the values are the same. We argue that in our case we can ignore this distinction and interpret our irregular time series as a regular one. This is possible as our irregular series has a fundamental difference from common ones: usually the values from an irregular series are a kind of snapshot that are taken at certain times, but change is happening at all times whether a snapshot gets taken at that time or not. But in our case the values that we have are the *only* changes that occur, so the actual time that has passed in between the values is irrelevant. There is only one catch with this theory: since the test instances are distributed over a whole range of machines, it



(a) Jump included



(b) Jump removed

Figure 5.1: Prediction intervals for three values

is possible, and even rather likely, that between two values *on one machine* there are other values on other machines. However, since potential changes would then be detected on those other machines earlier, this catch can still be safely ignored. This applies equally to the current *t*-test method and would therefore not introduce any additional issues anyway.

I addition to this simple exponential smoothing two extensions have been developed to handle more complex cases. *Double exponential smoothing* is one such method that is better able to deal with trends in the data. Global trends do not really exist in our performance tests, though, so utilizing this approach would yield no immediate benefit and would introduce the need to find a way to determine the optimal *trend smoothing factor* similar to the smoothing factor we are already using. However, future work might look at a way to use this technique on short-term trends during work periods that focus on optimizations and other similar situations.

A further extension, *triple exponential smoothing*, sometimes called the *Holt-Winters* method (Winters, 1960; Goodwin, 2010), was created to deal with seasonal trends, for example cases where buying habits change predictably depending on the month. Such trends do not occur at all in our performance tests though and this method has therefore not been investigated further.

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 5.2 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,

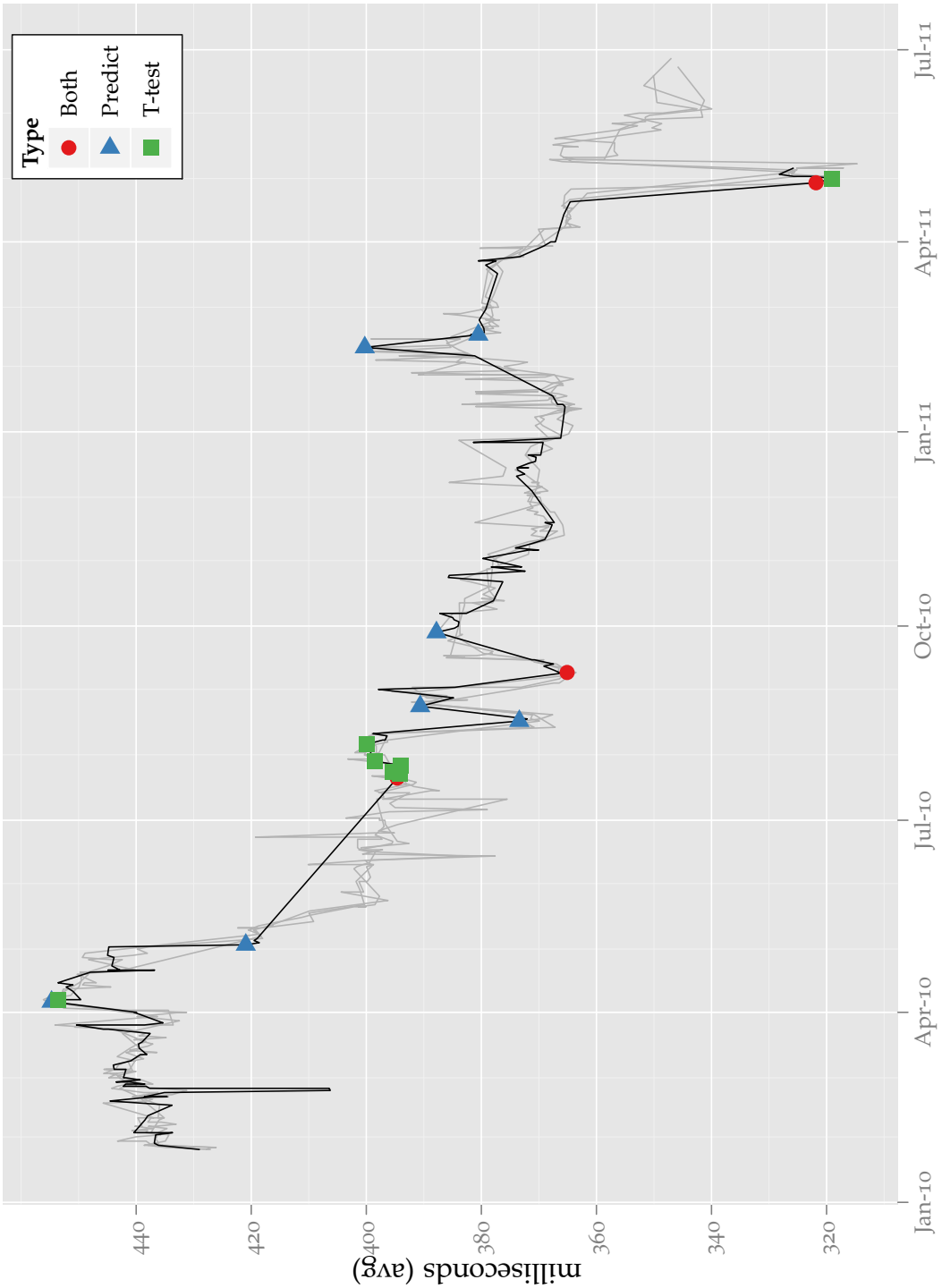


Figure 5.2: Comparison of the two analysis methods

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 even already exist in widespread 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.

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

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

CONCLUSIONS

6

This thesis 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. We evaluated three different categories of approaches with varying degrees of success to achieve these goals.

In Chapter 3 we looked at external influences like concurrently running programs, memory randomization and hard drive access. We found that all of them contributed to the variance to some degree, with the memory randomization surprisingly being the most influential one. This indicates that issues like memory alignment, physical layout as with NUMA architectures and prefetching have more influence on program performance than might be expected. Unfortunately we were not able to trace these assumptions at such a low level, but they are consistent with work done by others (see Chapter 2.6.1). We then suggested a way to minimize these influences in official tests run by Mozilla. While the advances we achieved with our modifications were significant, they did not reduce the variance to our ideal level, though.

Chapter 4 dealt with the internal workings of Firefox. Here we were focussing on three major aspects: the time the process actually runs on the CPU, the threads that are constantly created and destroyed by the thread pool and the event mechanism that is used in Firefox's XPCOM framework to do its work. Regarding the process time we discovered that there was no measurable improvement achieved by our modifications, and that in fact some of the tests had a slightly higher variance than before. Changing the thread pool implementation to only create one thread and keep it alive for the lifetime of the program had a similar result: a slight worsening in variance for some of the tests and no improvements in the others, indicating that these threading issues are not significantly responsible for the variance

we are seeing. Lastly we measured two event properties: the number of events that are used during a test run and the order in which they are used. We discovered that both of them had a certain amount of variance, but a correlation analysis concluded that this variance was only correlated to the test result variance in two respectively one cases. Interestingly the only test for which both properties were correlated is also the one with the longest test run time by far, suggesting that the length of the test is responsible for this. Possibly there are some influences that overshadow the correlation in shorter tests but get marginalized once the test length exceeds a certain threshold. This could be a promising starting point for future work.

Finally, in Chapter 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 variance and how it relates to the performance test variance, in addition to other possible sources of internal variance, should be a promising topic for future work.

There is one important thing to note about performance testing in general. The work done by Mytkowicz et al. (2009) and Kalibera et al. (2005) and us in Chapter 3 shows that even small changes in the environment can

have a measurable impact on the test results, and it is nearly impossible to eliminate all potential changes. For example, running a test with memory randomization disabled can obviously only give results that result from the specific memory layout that happens to be chosen for the current run of the program, but it cannot tell us anything about how favourable this layout is in regard to the hardware issues we mentioned. So we could get a favourable layout in one run, and an unfavourable one in the next run (for example due to shifted code as investigated by Gu et al. (2004)), and the effect would be that the results differ from each other even without any genuine performance changes. The only solution to this would be to take several samples, i.e. tests, while only changing this specific parameter – in this case memory layout, which could be achieved by enabling randomization again – and average over the results. Unfortunately this is not really feasible in many continuous integration scenarios because of the additional resources it would require. But it is something to keep in mind when evaluating test setups and techniques.

6.1 FUTURE WORK

Looking at our results there are still various variance factors that we have not found yet. It would be valuable to know whether there are other external factors that can be reduced or whether they are part of the above mentioned ones that can only be approximately solved through averaging. Additionally the internal variance in the events is worth investigating further, especially the question of why it almost only correlates with the longest-running test.

Appendices

SCRIPTS

This is the shell script we used to prepare for the external optimization tests. As mentioned in the text memory randomization and CPU isolation mechanisms were enabled directly in the kernel.

```
# official optimizations
rm /dev/random
mknod /dev/random c 1 9
echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
echo performance > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor

if [[ "$1" == "-o" ]]; then
    exit
fi

# our optimizations start here
stop gdm
stop ssh
stop avahi-daemon
/etc/init.d/networking stop
pkill dhclient
stop network-manager
pkill modem-manager
pkill wpa_supplicant
stop cron
stop atd
/etc/init.d/cups stop
pkill pulseaudio
pkill irqbalance

mount /ramfs
chown test:test /ramfs
```

COMPLETE PLOTS

B

B.1 ISOLATED MODIFICATIONS

Explanation of the abbreviations:

- `nomod`: No modifications except for the official ones (see Section 2.5.1).
- `plain`: All non-essential processes terminated.
- `norand`: Memory randomization disabled.
- `exclcpu`: Firefox runs exclusively on one processor with the rest of the processes running on the other one.
- `ramfs`: Firefox and all of the test data and logs reside on a RAM disk (no hard disk access).

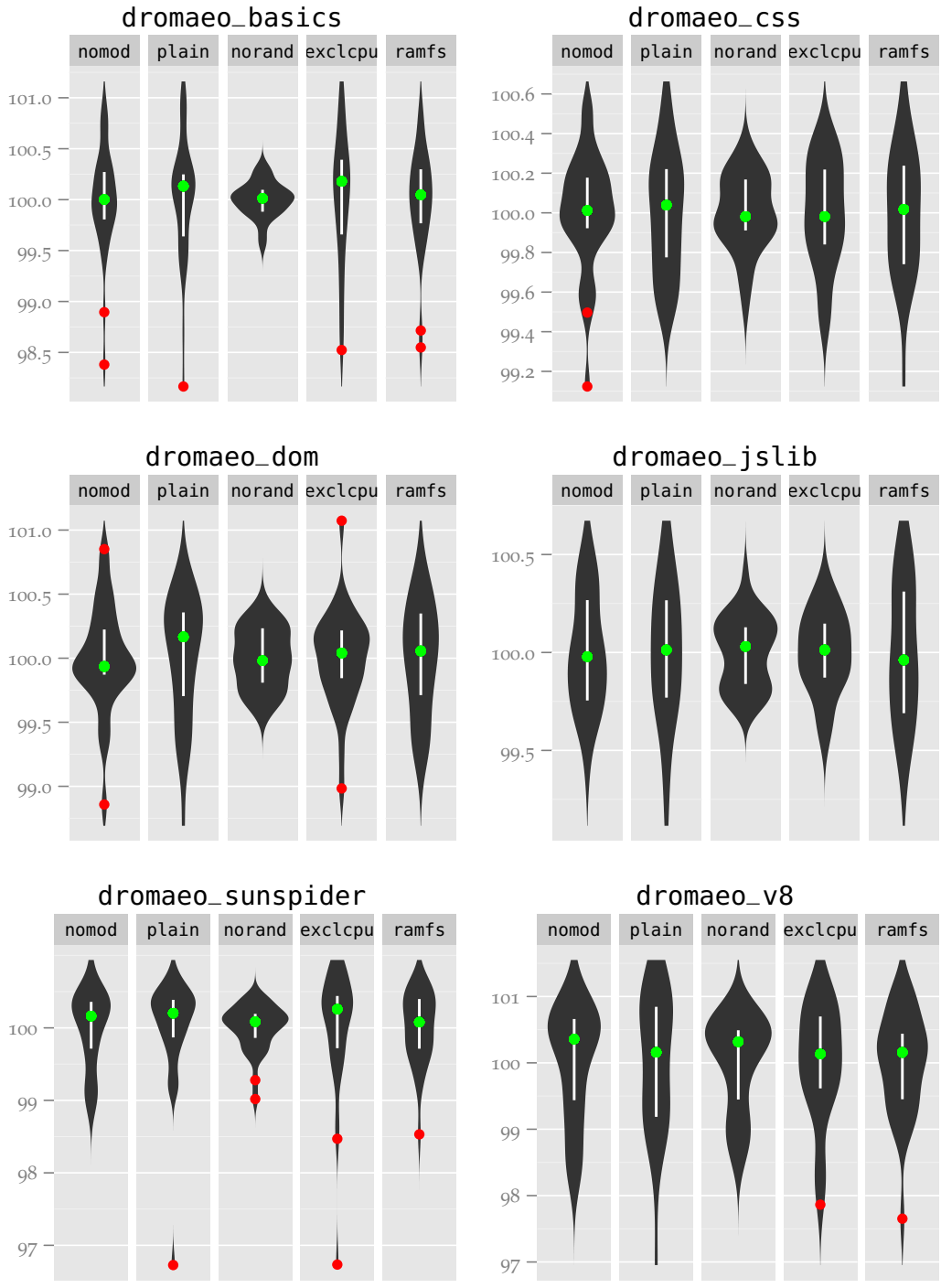


Figure B.1: Isolated modifications, percentage of mean, part 1

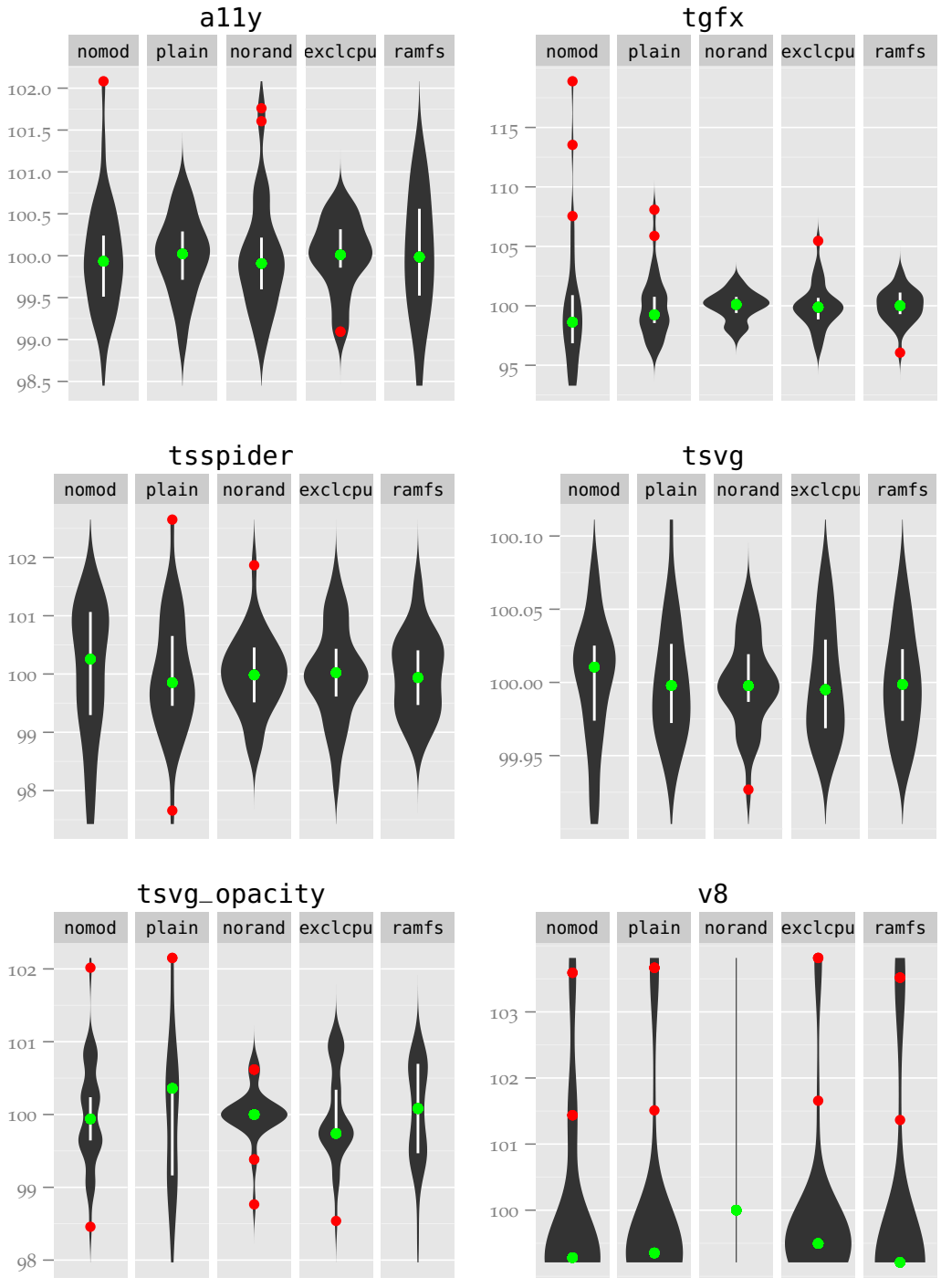


Figure B.2: Isolated modifications, percentage of mean, part 2

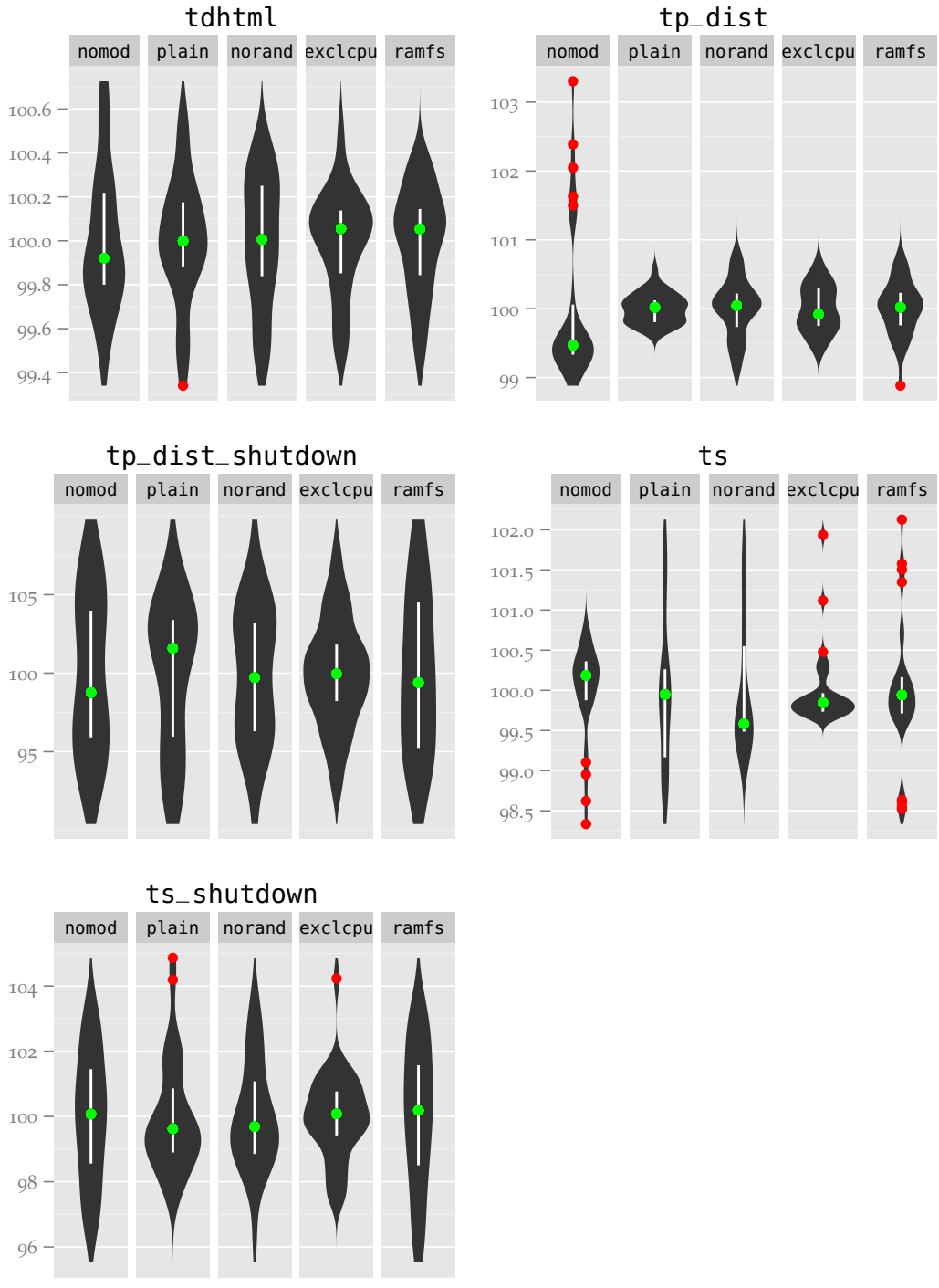


Figure B.3: Isolated modifications, percentage of mean, part 3

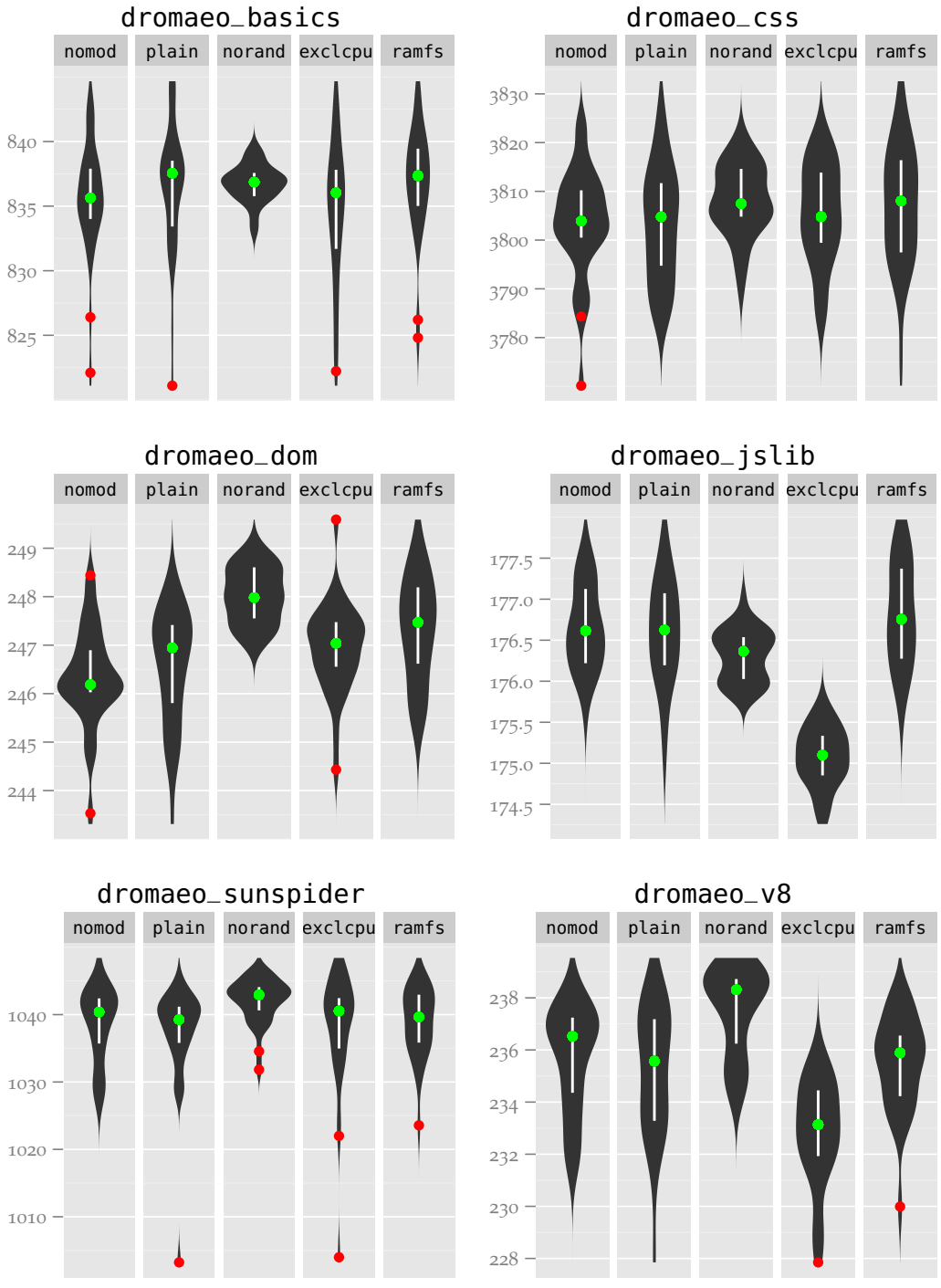


Figure B.4: Isolated modifications, absolute values, part 1

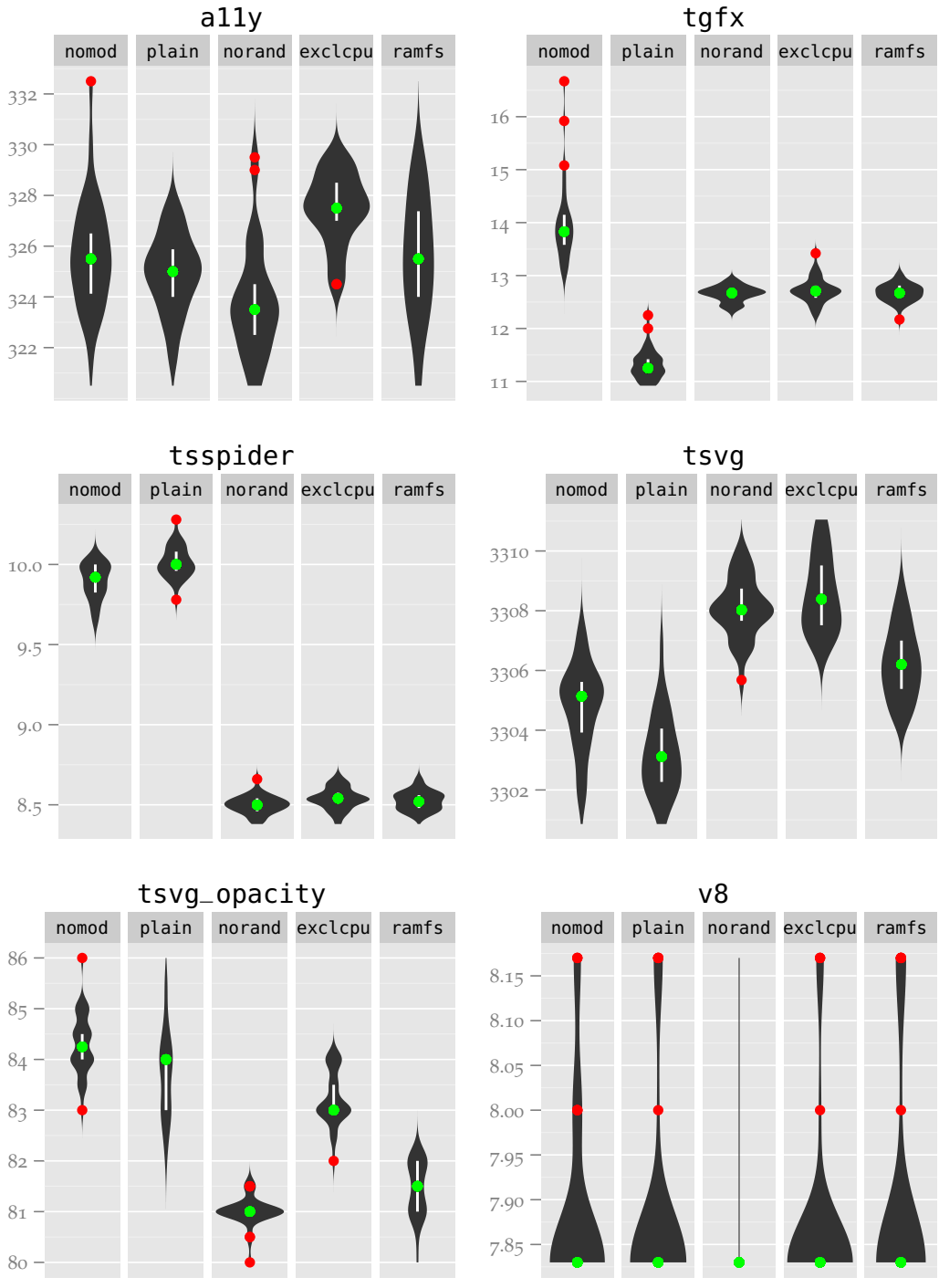


Figure B.5: Isolated modifications, absolute values, part 2

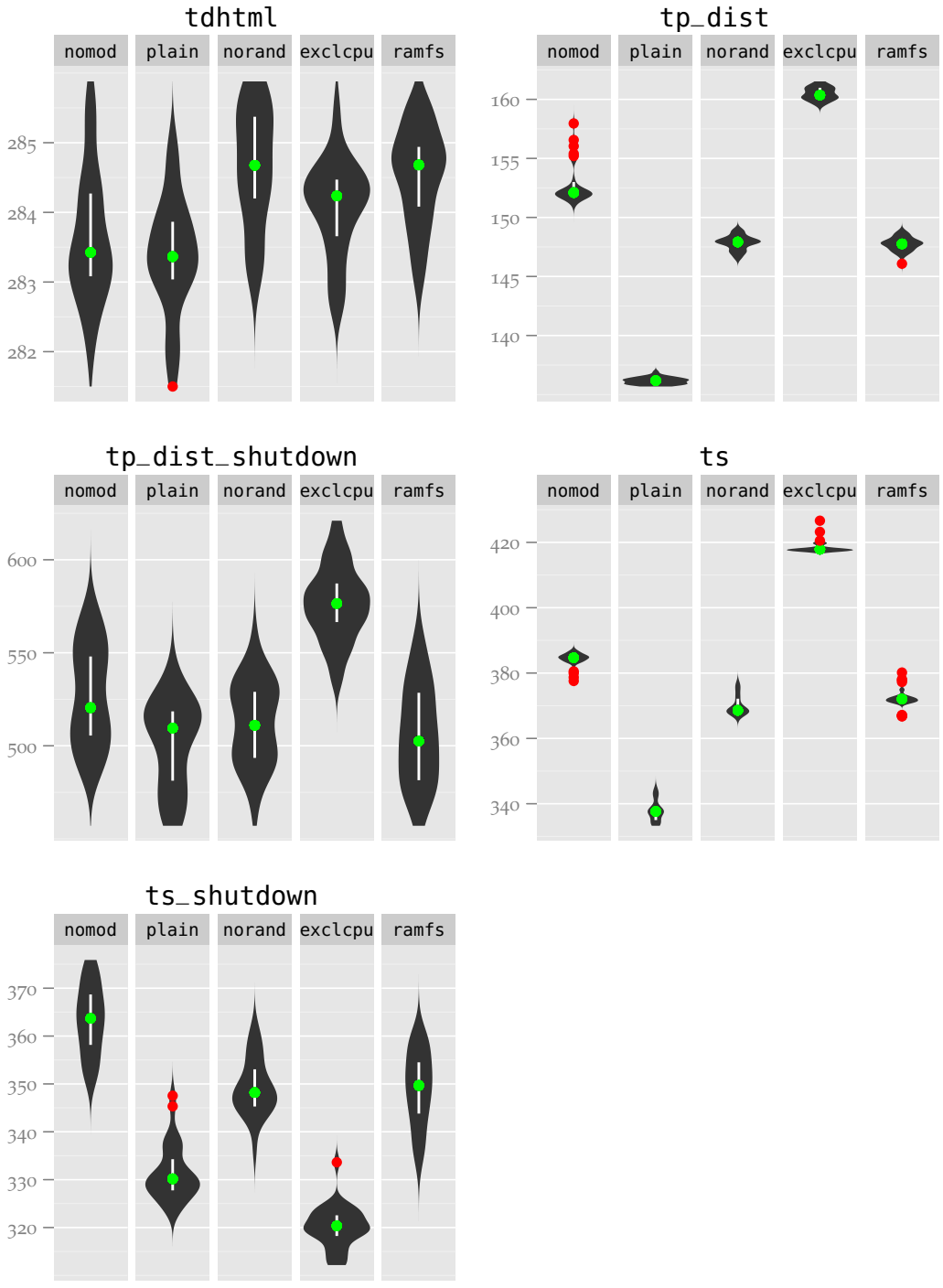


Figure B.6: Isolated modifications, absolute values, part 3

B.2 MEMORY RANDOMIZATION COMPARISONS

Explanation of the abbreviations:

- `nomod`: No modifications except for the official ones (see Section 2.5.1).
- `cumul`: All of the changes from Chapter 3.
- `norand`: Memory randomization disabled.

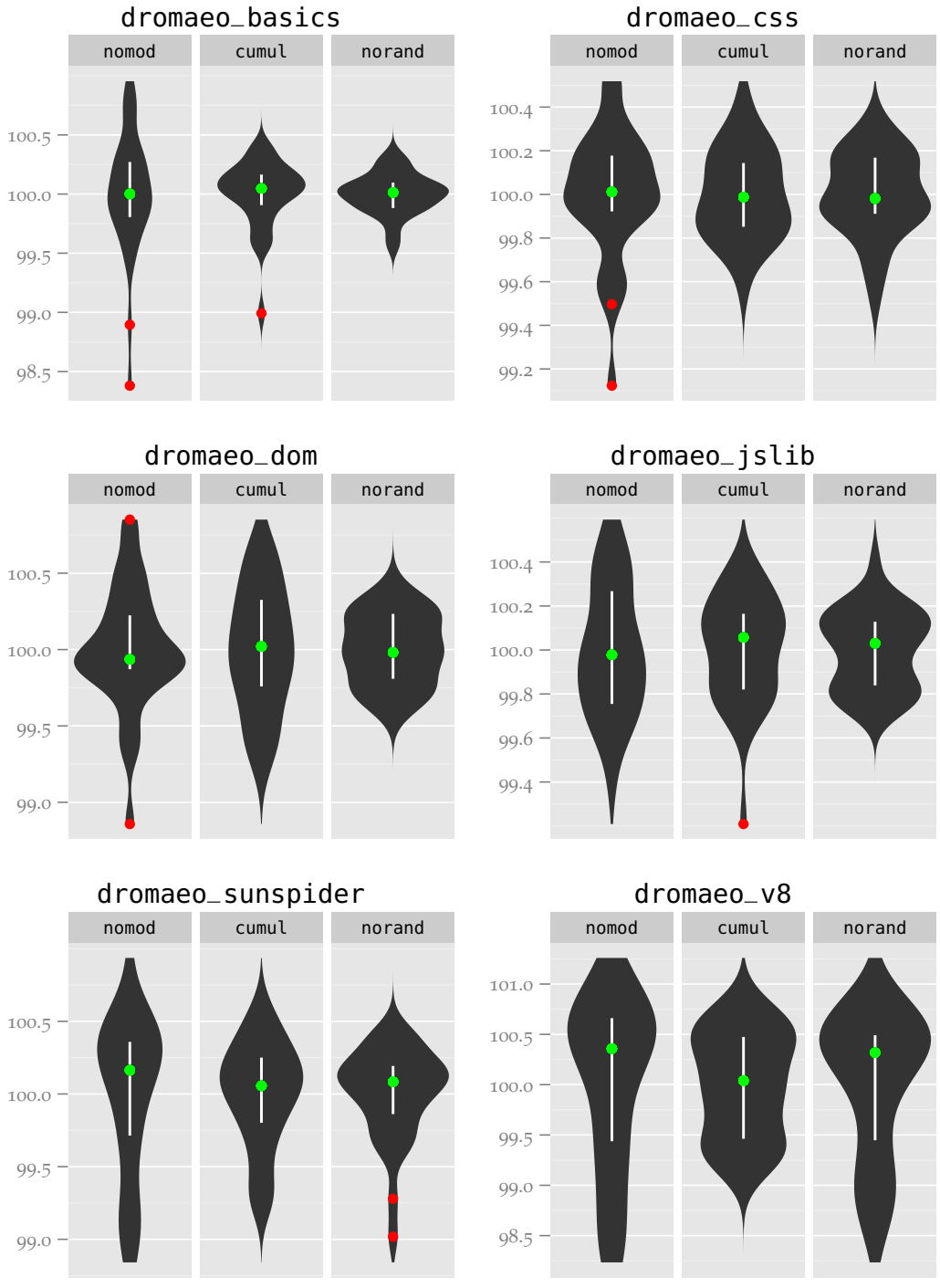


Figure B.7: `norand` comparisons, percentage of mean, part 1

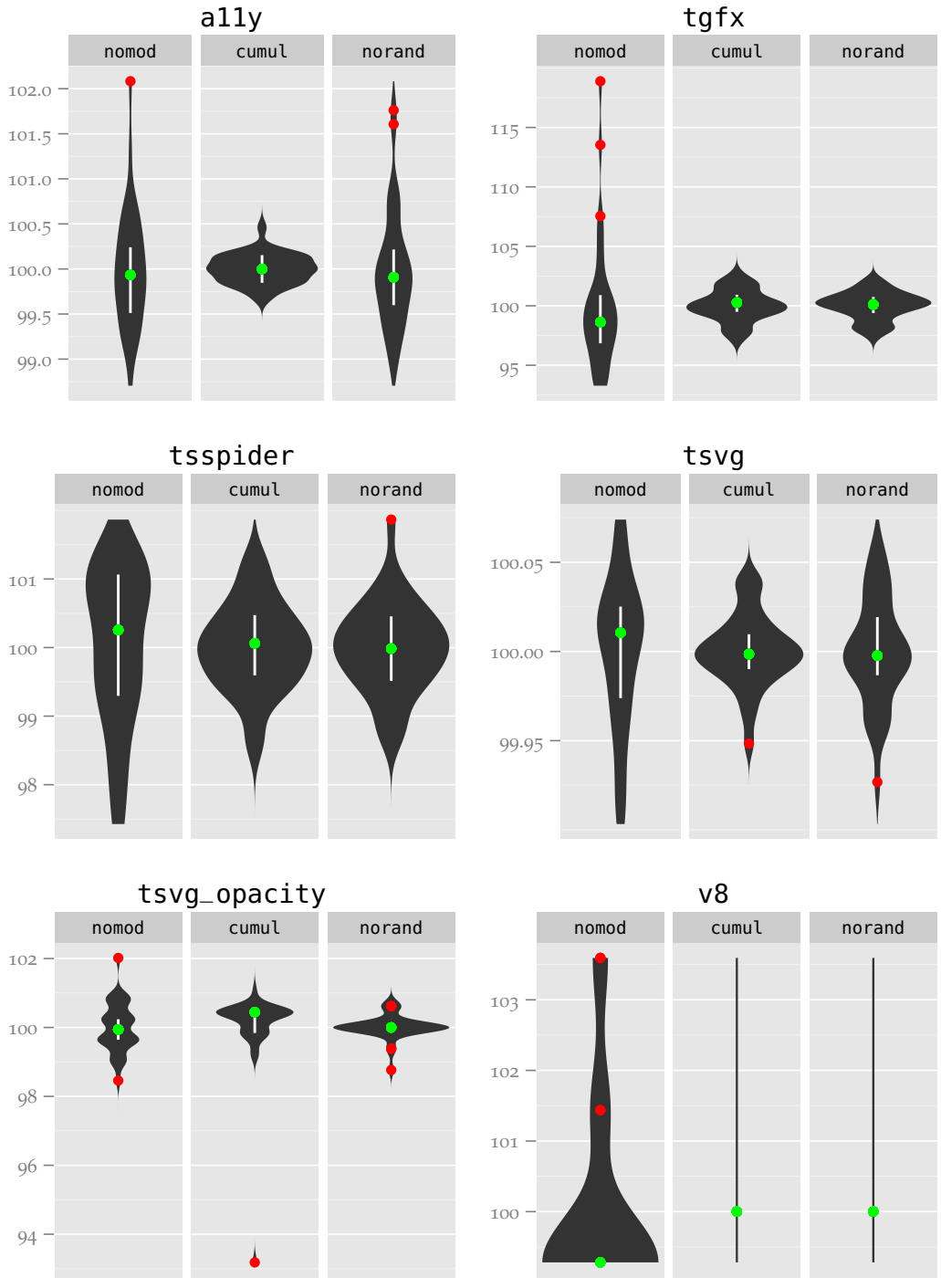


Figure B.8: `norand` comparisons, percentage of mean, part 2

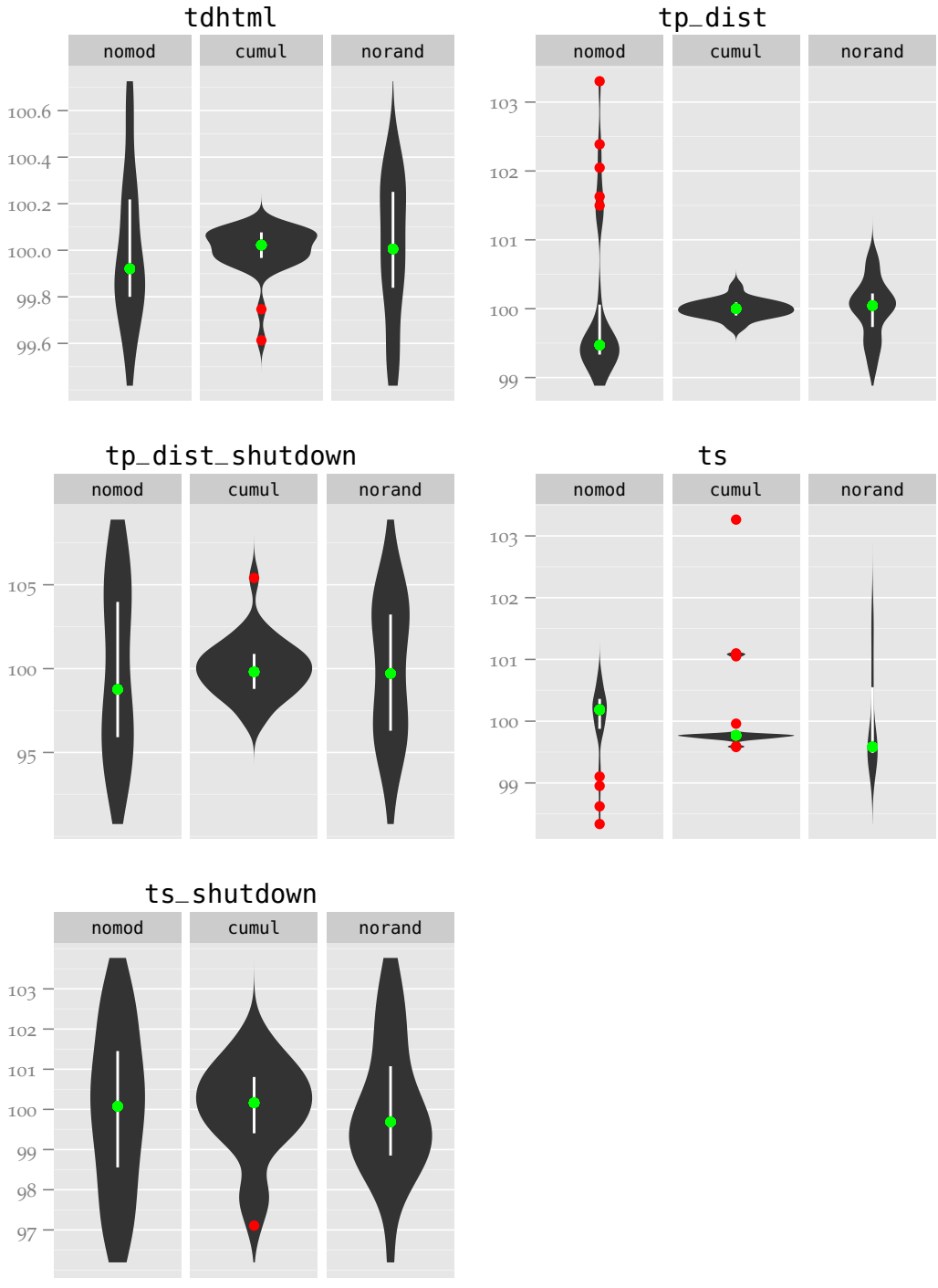


Figure B.9: `norand` comparisons, percentage of mean, part 3

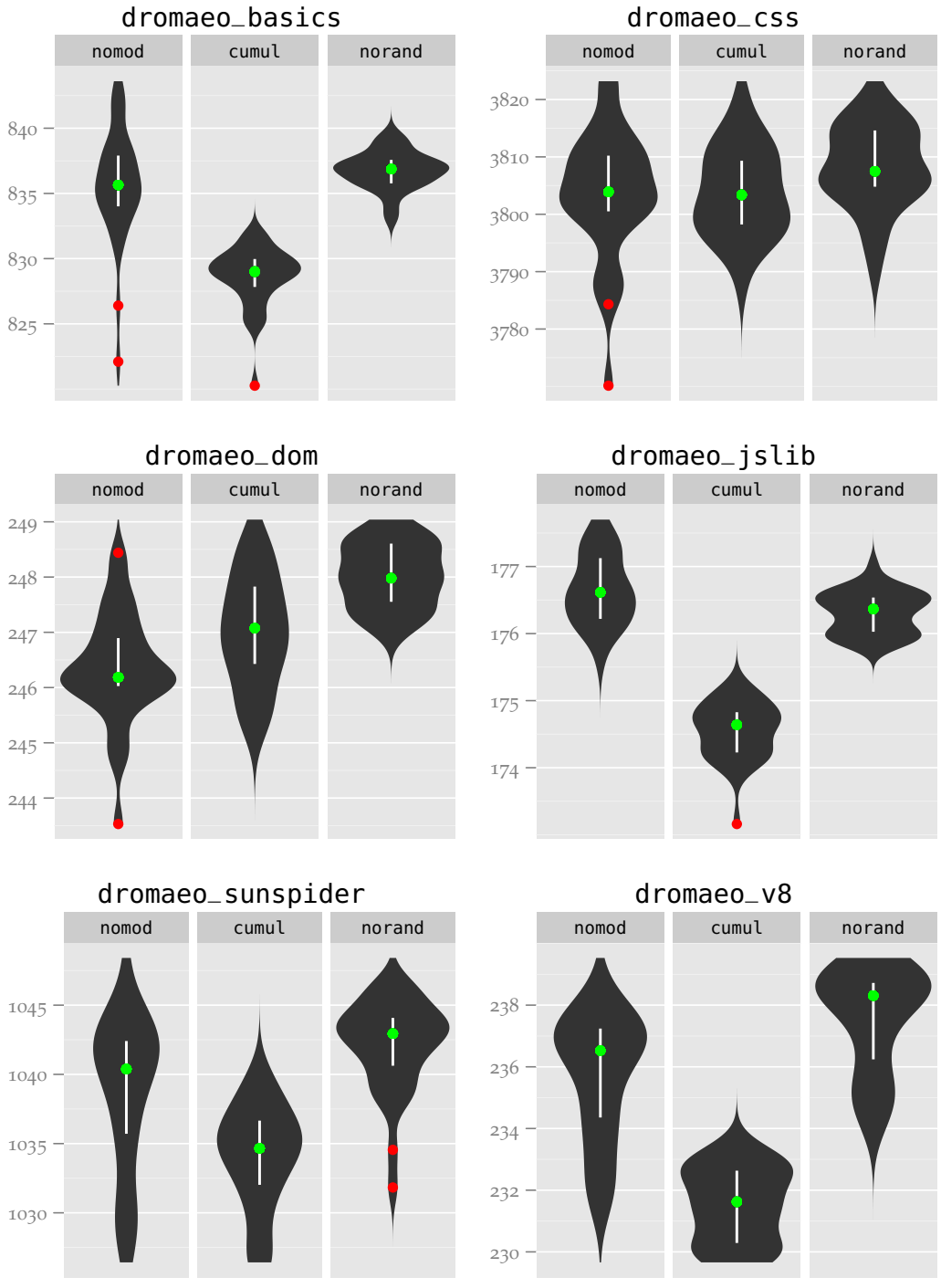


Figure B.10: `norand` comparisons, absolute values, part 1

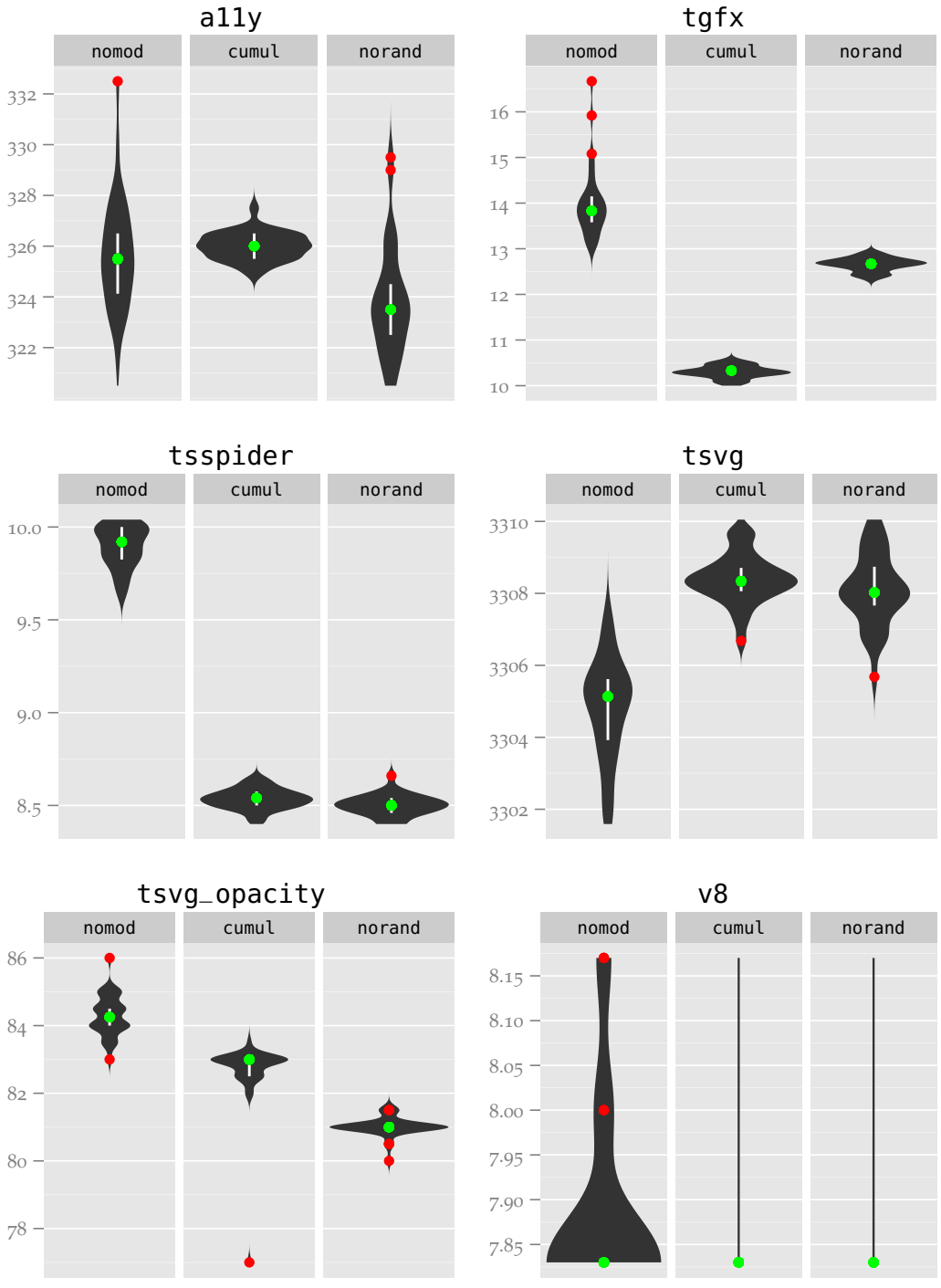


Figure B.11: `norand` comparisons, absolute values, part 2

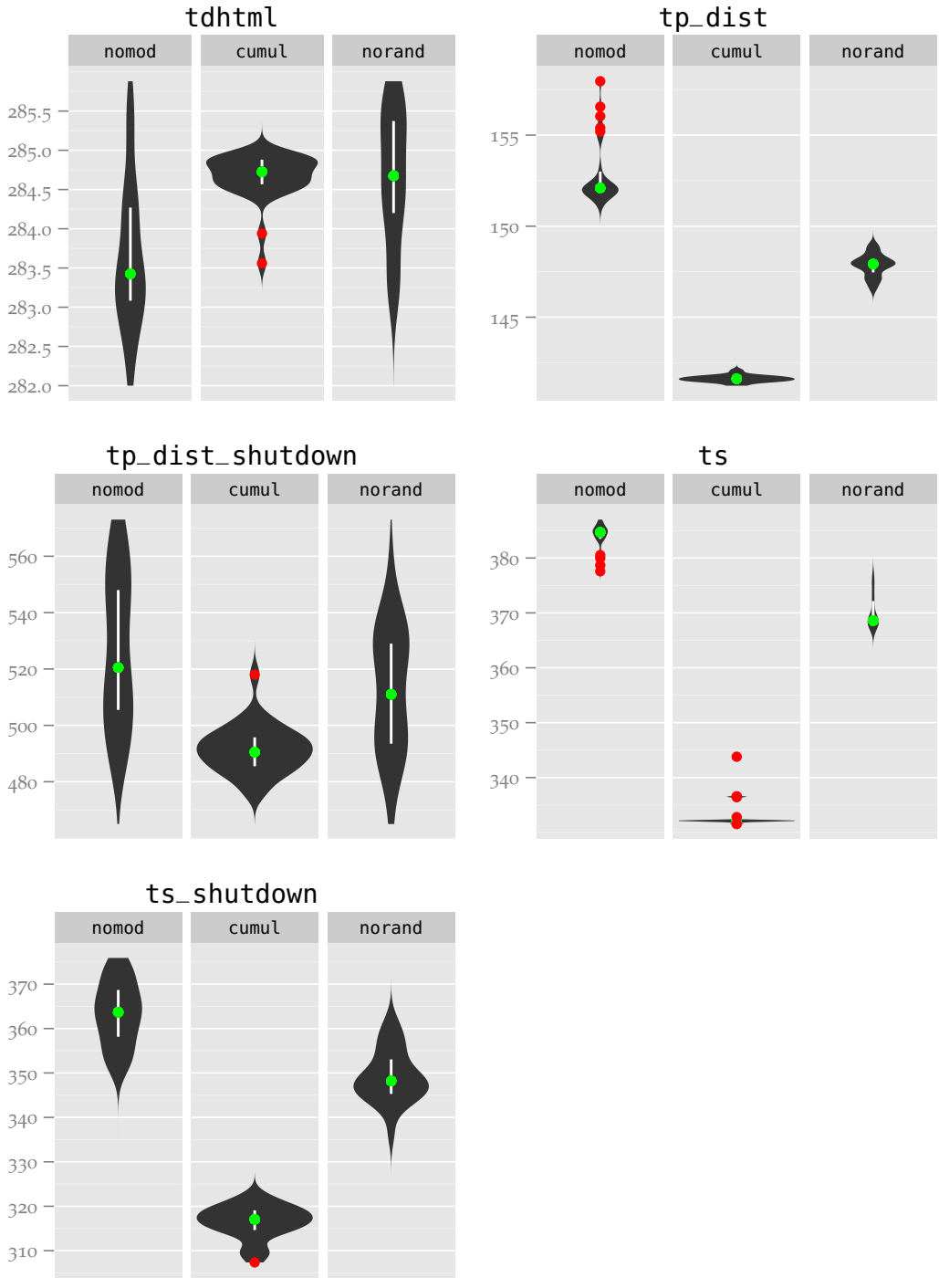


Figure B.12: `norand` comparisons, absolute values, part 3

B.3 CPU TIME MODIFICATION

Explanation of the abbreviations:

- `cumul`: All of the changes from Chapter 3.
- `cputime`: The CPU time changes from Section 4.2.1.

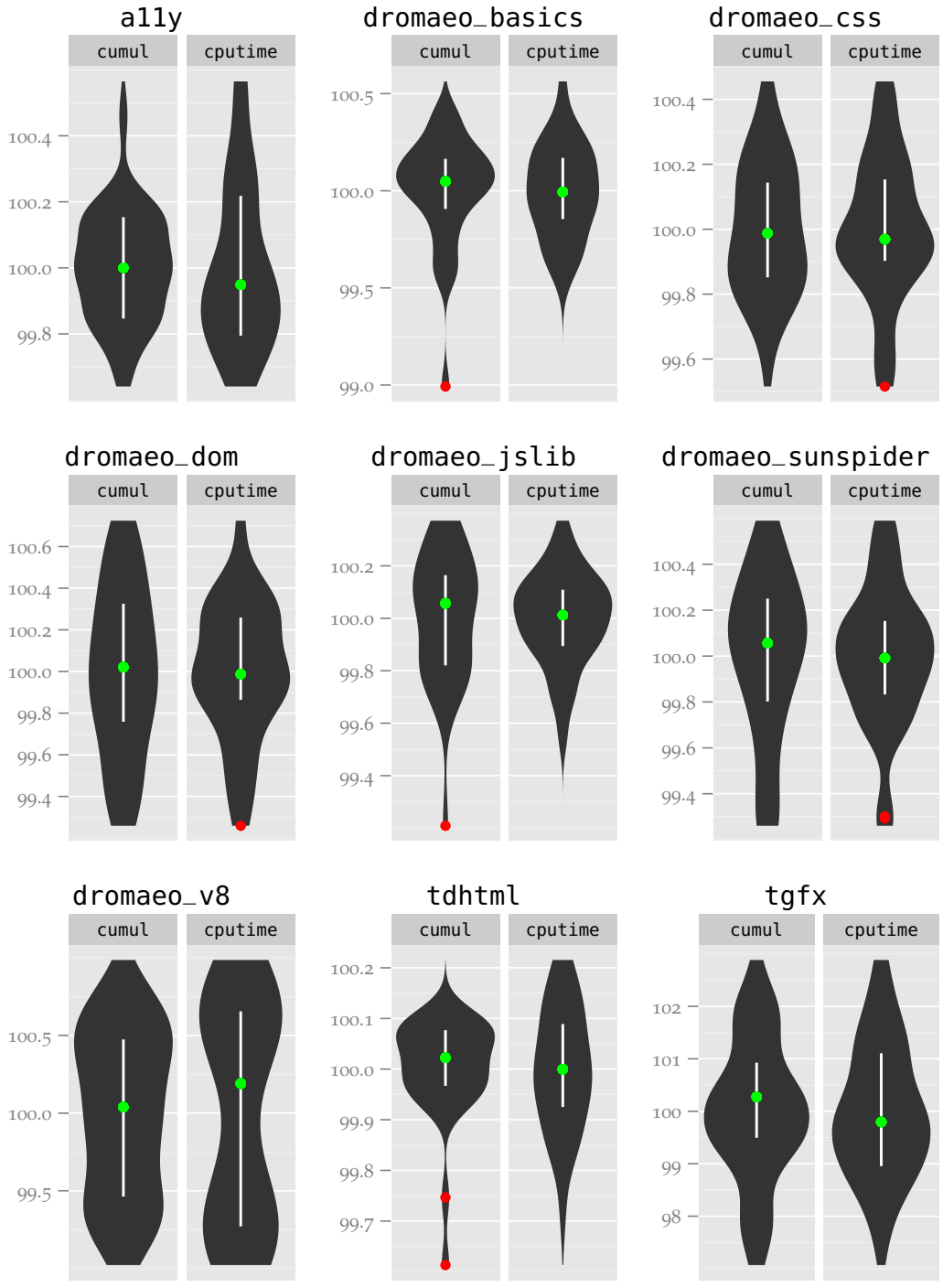


Figure B.13: CPU time modification, percentage of mean, part 1

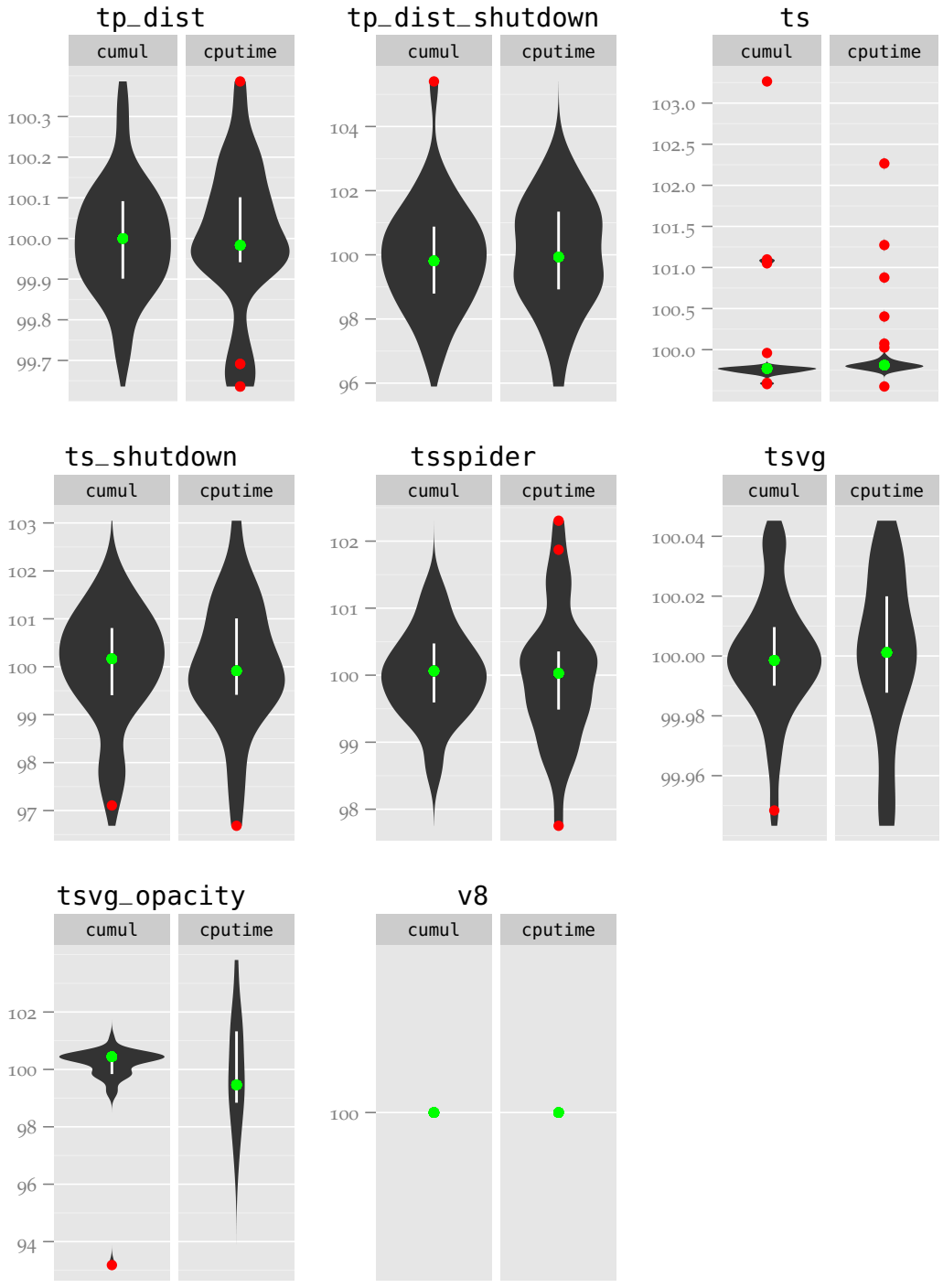


Figure B.14: CPU time modification, percentage of mean, part 2

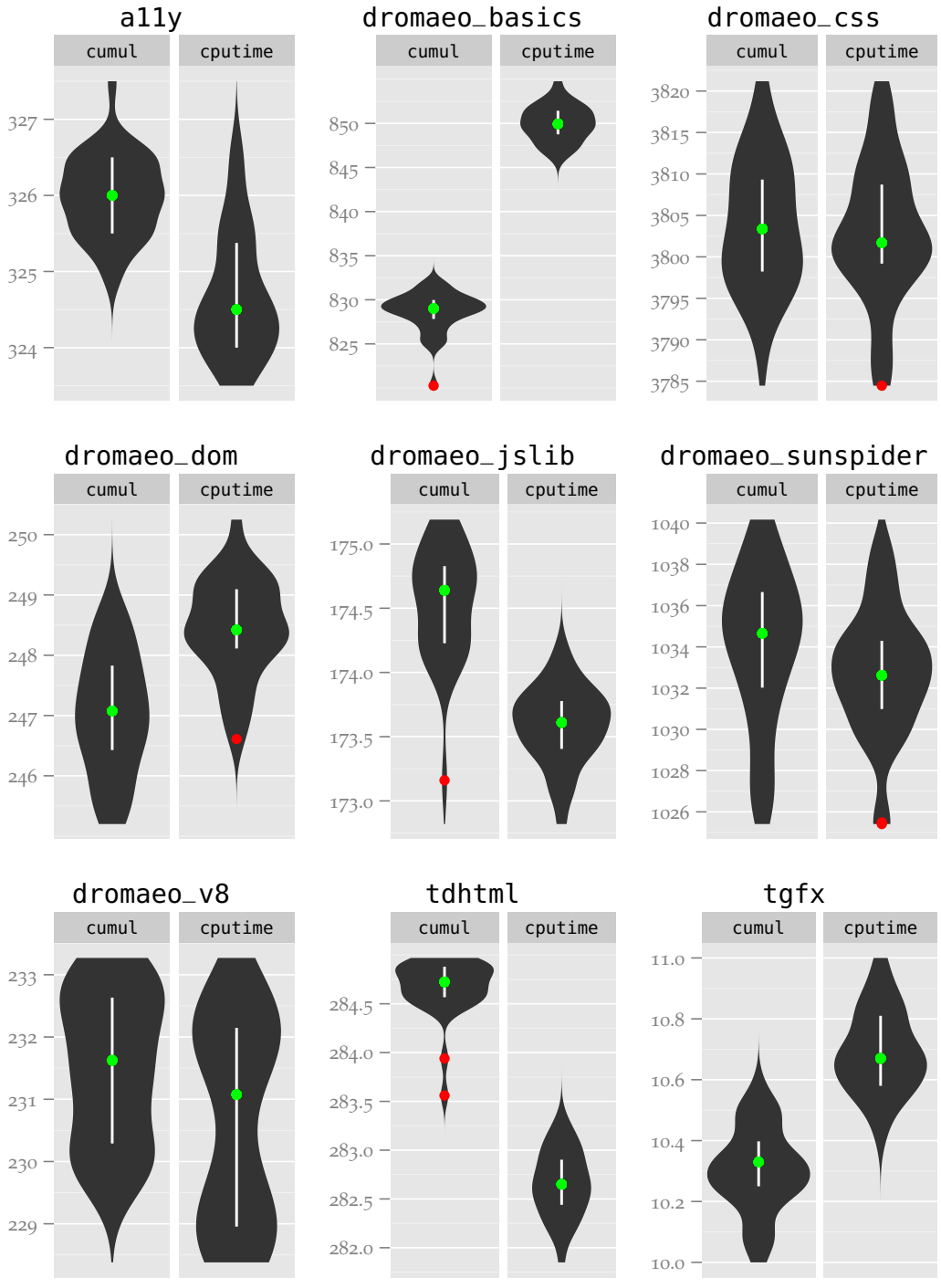


Figure B.15: CPU time modification, absolute values, part 1

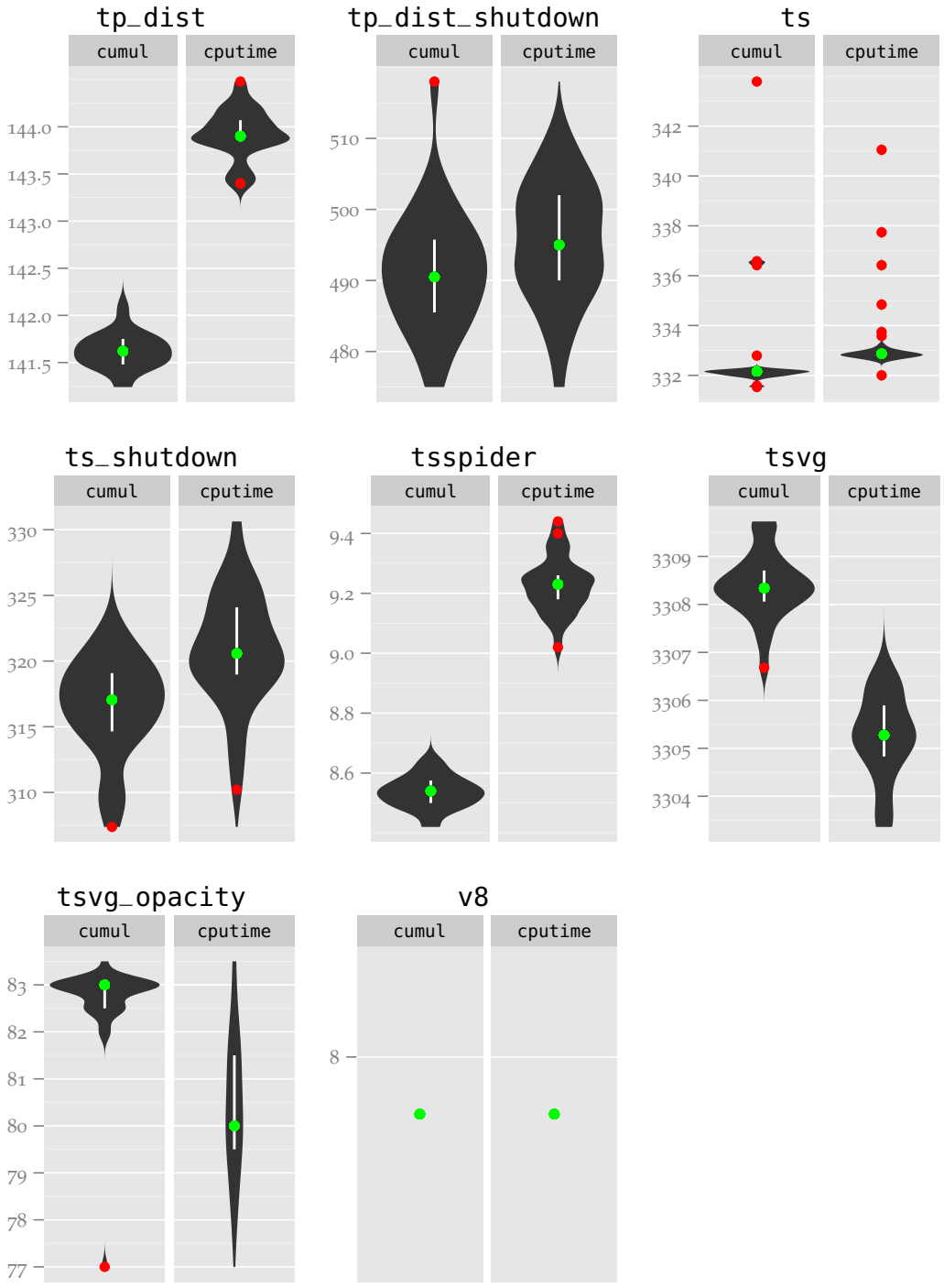


Figure B.16: CPU time modification, absolute values, part 2

B.4 THREAD POOL MODIFICATION

Explanation of the abbreviations:

- `cputime`: The CPU time changes from Section 4.2.1.
- `tp1`: The thread pool modifications from Section 4.4.1.

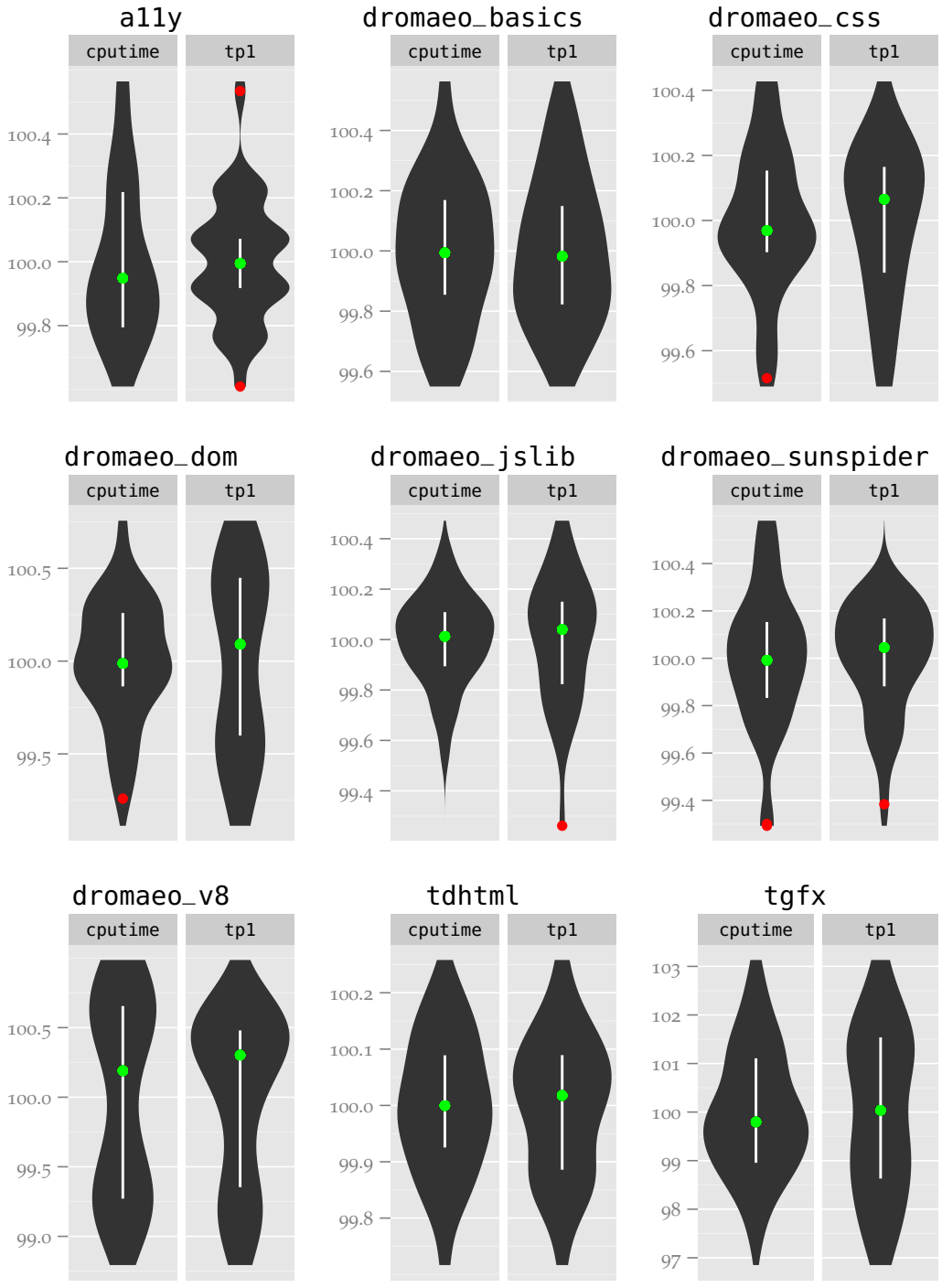


Figure B.17: Thread pool modification, percentage of mean, part 1

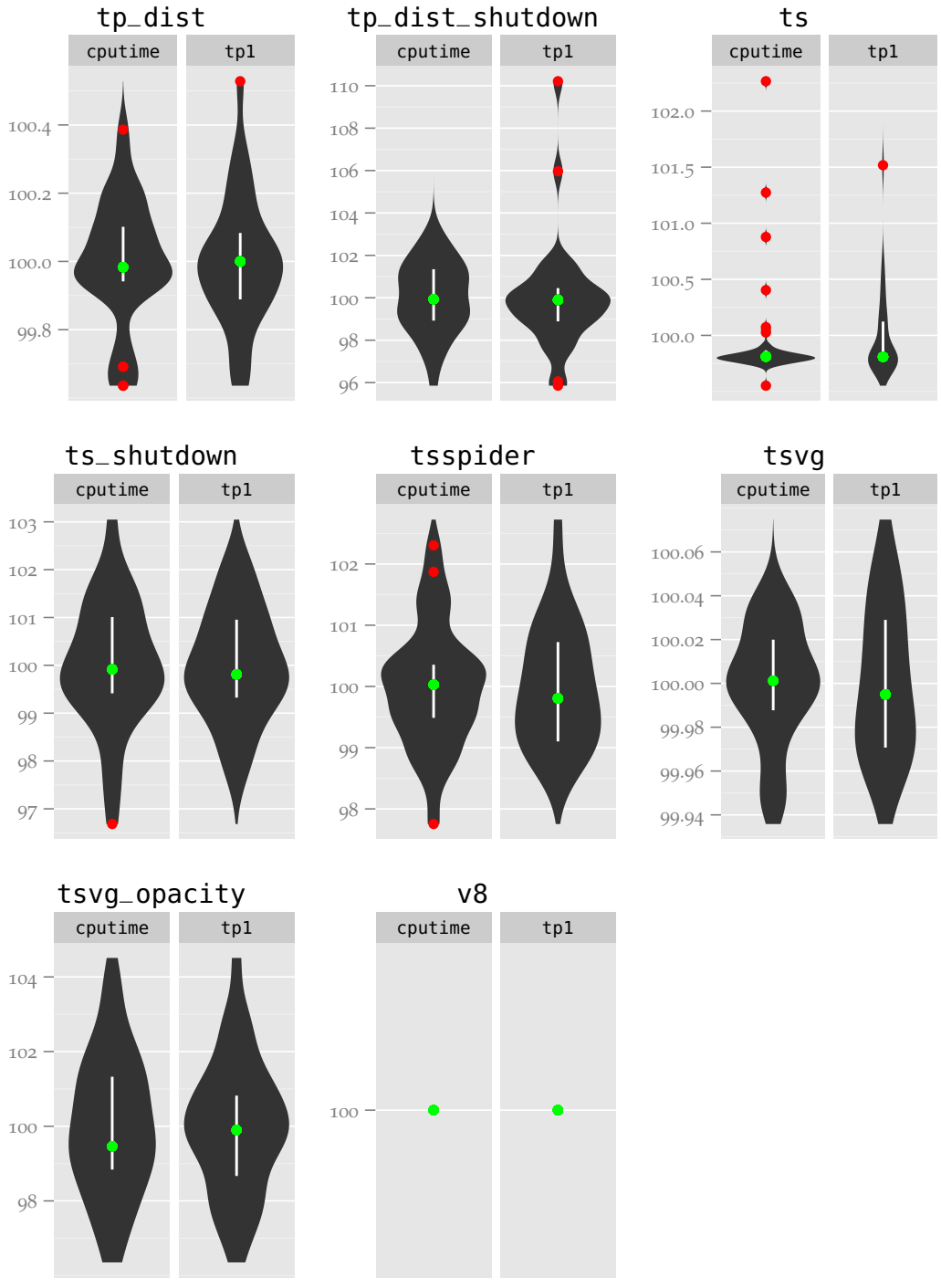


Figure B.18: Thread pool modification, percentage of mean, part 2

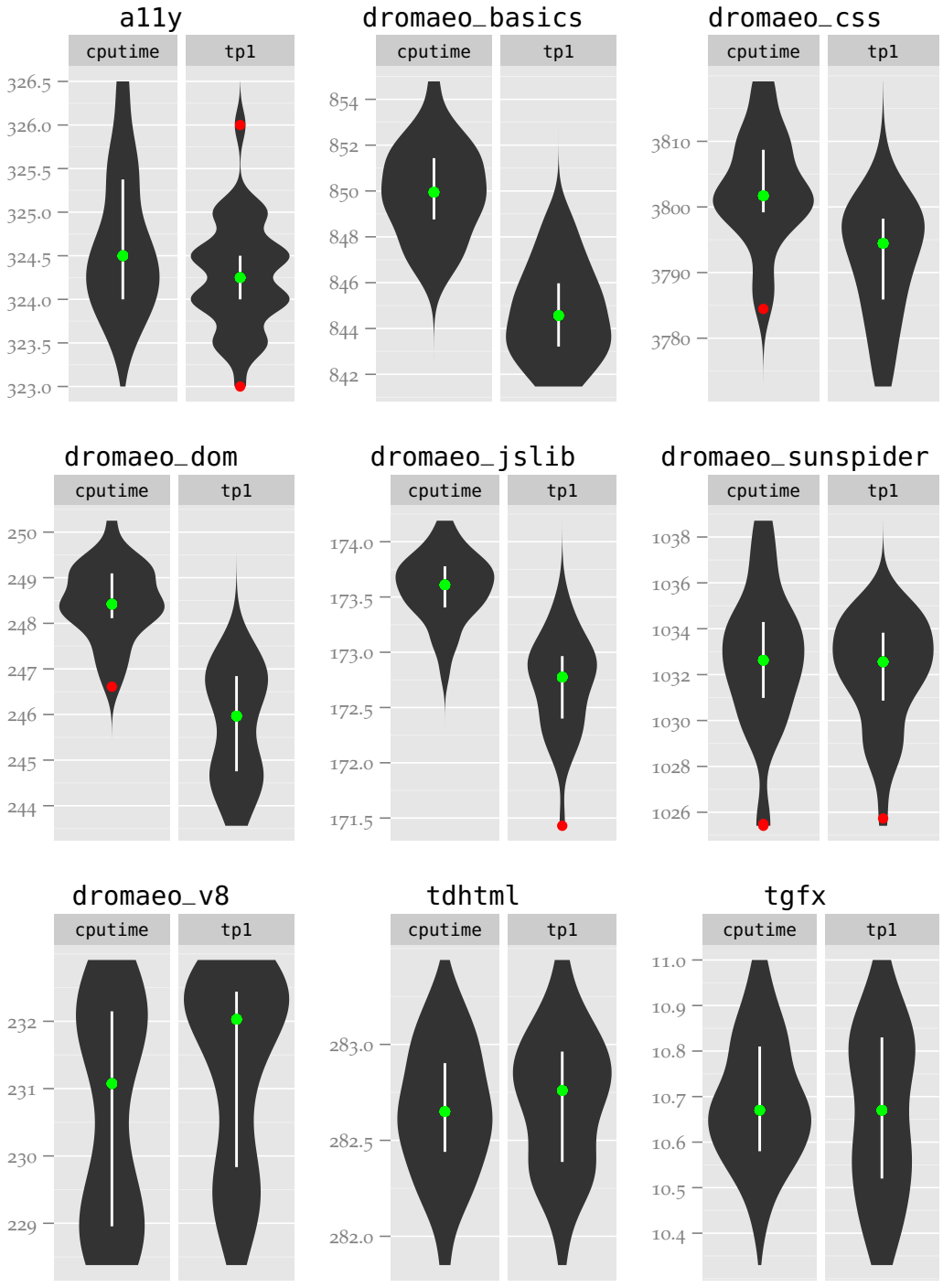


Figure B.19: Thread pool modification, absolute values, part 1

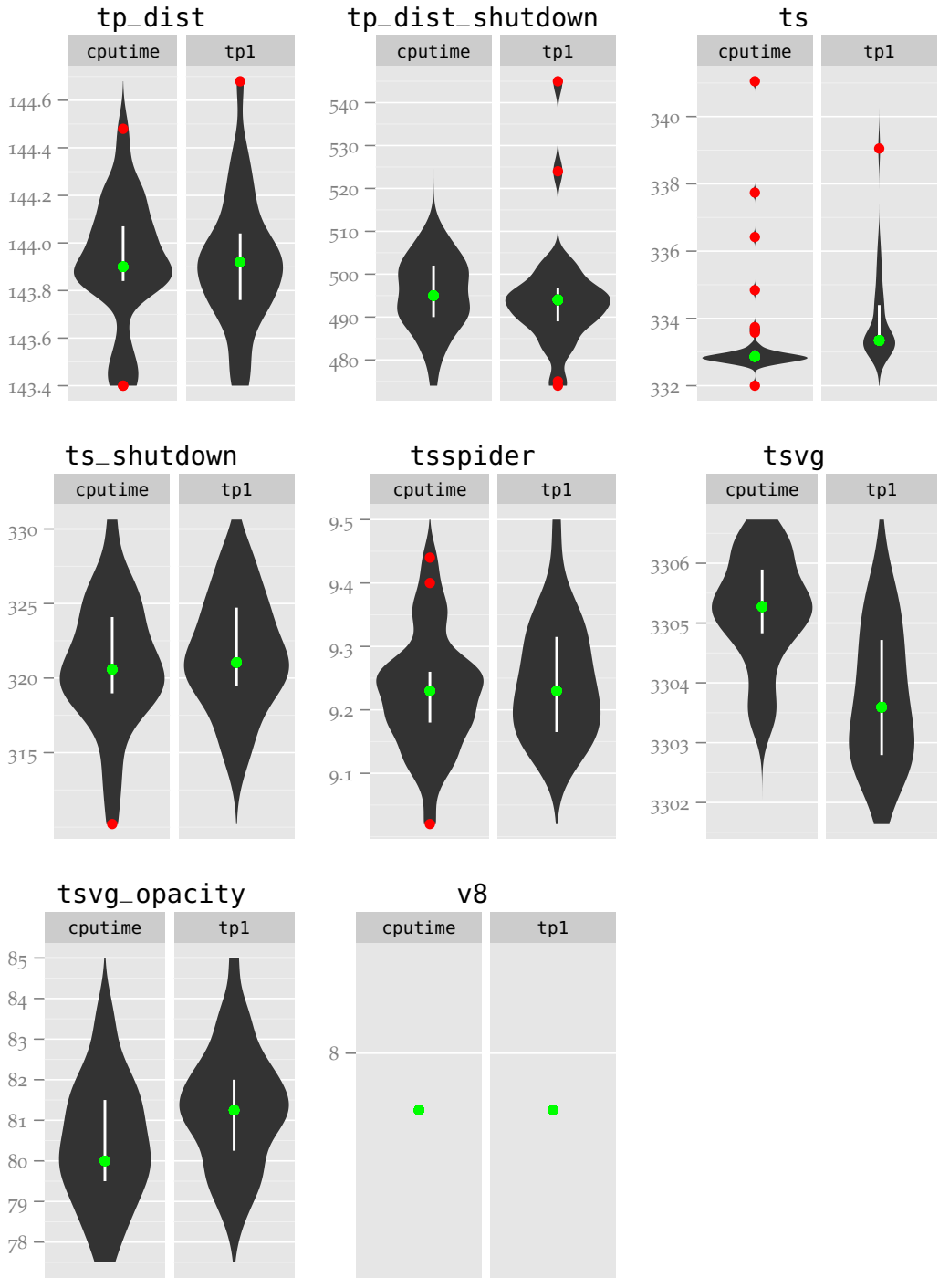


Figure B.20: Thread pool modification, absolute values, part 2

BIBLIOGRAPHY

Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 7–18, 2003.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 53–64, Pittsburgh, Pennsylvania, USA, 2010a. ACM.

Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. *9th OSDI*, 2010b.

Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 3 edition, November 2005.

Morton B. Brown and Alan B. Forsythe. Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69(346):364–367, June 1974.

L. Le Cam. The central limit theorem around 1935. *Statistical Science*, 1(1): 78–91, February 1986.

Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition,*, volume 2, page 1119, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

Heming Cui, Jingyue Wu, Chia-che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. *9th OSDI*, 2010.

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09*, page 85, Washington, DC, USA, 2009.

Ulrich Drepper. What every programmer should know about memory. `http://people.redhat.com/drepper/cpumemory.pdf`, 2007.

Martin Fowler. Continuous integration. `http://www.martinfowler.com/articles/continuousIntegration.html`, May 2006.

Armen Zambrano Gasparnian. Release engineering at mozilla through buildbot. `http://fosslc.org/drupal/content/release-engineering-mozilla-through-buildbot`, October 2010.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, page 57, Montreal, Quebec, Canada, 2007.

Paul Goodwin. The Holt-Winters approach to exponential smoothing: 50 years old and going strong. *FORESIGHT*, page 4, 2010.

Google Inc. V8 JavaScript engine - benchmarks. `http://code.google.com/apis/v8/benchmarks.html`, 2008.

Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. *In Component and Middleware Performance Workshop, OOPSLA*, 2004.

Jerry L. Hintze and Ray D. Nelson. Violin plots: A box Plot-Density trace synergism. *The American Statistician*, 52(2):181–184, May 1998.

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

Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems*, pages 853—862, 2005.

Howard Levene. Robust tests for equality of variances. In Ingram Olkin, Sudhish G. Ghurye, Wassily Hoeffding, William G. Madow, and Henry B. Mann, editors, *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, pages 278–292. Stanford University Press, 1960.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, Washington, DC, USA, 2009. ACM.

Robert O'Callahan. Private communication, 2010.

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09*, page 97, Washington, DC, USA, 2009.

Joseph Lee Rodgers and W. Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, February 1988.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, page 298–307, Washington DC, USA, 2004. ACM.

S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (Complete samples). *Biometrika*, 52(3/4):591–611, December 1965.

Maciej Stachowiak. Announcing SunSpider 0.9. http://www.webkit.org/blog/152/announcing-sunspider-09/, December 2007.

Anamaria Stoica. Mozilla's build system. https://anamariamoz.wordpress.com/2010/11/08/mozillas-build-system/, November 2010.

Andrew S. Tanenbaum. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2 edition, March 2001.

Dan Tsafrir, Keren Ouaknine, and Dror G. Feitelson. 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*, pages 231–237, 2007.

Peter R. Winters. Forecasting sales by exponentially weighted moving averages. *Management Science*, 6(3):324–342, 1960.

Mohammed Yar and Chris Chatfield. Prediction intervals for the Holt-Winters forecasting procedure. *International Journal of Forecasting*, 6(1): 127–137, 1990.