NIST Technical Note 1855

Screening for factors affecting application performance in profiling measurements

David Flater

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.TN.1855

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Technical Note 1855

Screening for factors affecting application performance in profiling measurements

David Flater
Software and Systems Division
Information Technology Laboratory

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.TN.1855

October 2014

U.S. Department of Commerce
Penny Pritzker, Secretary of Commerce

National Institute of Standards and Technology
Willie E. May, Acting Under Secretary of Commerce for Standards and Technology and Acting Director

National Institute of Standards and Technology Technical Note 1855
Natl. Inst. Stand. Technol. Tech. Note 1855, 31 pages (October 2014)
This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.TN.1855
CODEN: NTNOEF

Screening for factors affecting application performance in profiling measurements

David Flater

October 2014

Abstract

This report provides an example application of screening techniques in experimental computer science, including validation and selection of metrics and measures, the screening experiment itself, and supporting statistical methods. Together with related references, it is intended to encourage other computer scientists to make more use of established statistical methods in performance evaluations.

The described screening experiment sought to identify which of three factors (environment, frame pointers, and profiler options) affected the performance of the application under test. The results showed that they all had a measurable impact, but the shapes and ranges of the distributions of measurement results would likely make their impacts immaterial in experiments using smaller samples. The validation phase of this experiment yielded evidence that the ill-behaved distributions are attributable to actual performance variations rather than measurement error.

1 Introduction

The mission of the Software Performance project is to strengthen the scientific foundations of software performance measurement ("metrology for information technology"). Its goals are:

- Application: Replace unreliable common practices with rigorously-studied methods grounded in design of experiments.
- Research: Solve measurement challenges created by the evolution and increased complexity of commodity hardware.
- Transfer: Feed improved methods back into the community of practice.

Previous work on this project has included a case study of performance degradation attributable to one form of software hardening [1], a validation of profiling tools[1] for C and C++ applications under 64-bit Linux [2], and a treatise on estimation of uncertainty in application profiles [3]. A test suite for validating application profiling tools, a test driver for Android, an R package for finding confidence intervals, and a Ruby "gem" for managing experimental data are available as associated software products [4, 5, 6].

The research plan brings us now to the topic of screening for factors affecting application performance in profiling measurements. "Screening" is a specialized type of experimental design. The primary purpose of a screening design is to separate significant factors from insignificant ones [7].

We conducted a screening experiment with the goal of identifying which of three factors (environment, frame pointers,[2] and profiler options) affects the performance of an application being profiled. However, as with any experiment, it was important to validate the infrastructure before proceeding to full-scale data collection. Thus, there were two phases: validation and selection of metrics and measures, followed by the main experiment. This report provides details on both phases.

[1] Profiling tools are the instruments used to measure software performance at the function and application levels.
[2] Frame pointers are an optional feature/protocol of applications that is used by some profiling options.

The remainder of this report is assembled as follows. Section 2 introduces terms used throughout the report. Section 3 covers validation and selection of metrics and measures. Section 4 covers the screening experiment itself. Finally, Section 5 ends with conclusions and future work.

2 Terms

The terms *factor* and *independent variable* are synonymous. They refer to a property of the system under test that is deliberately changed within the scope of the experiment. In some communities these are called *input variables*.

The different values to which a factor is set in an experiment are called *levels* of that factor. Usually, the levels of factors are mapped onto the same numerical scale to simplify discussion and data management. In this report we use the scale of 0,1 for the "low" and "high" values of 2-levelled factors and integers 0–N for factors with more than 2 levels.

A *treatment* is a specific combination of factor levels. If there are two factors, each of which has two levels, then there are four possible treatments.

A *quantity* is an objective property (*e.g.*, of a software artifact or program execution) that has a magnitude that can be expressed as a numeric value with a reference (*e.g.*, an SI unit) that defines what the numeric value indicates. When results are quantitative in nature but not necessarily expressing the magnitude of an objective property, the weaker term *metric* is used.

A *dependent variable* is a quantity or metric that is used to determine the effects of different treatments. In some communities these are called *response variables* or *output variables*.

The term *measures* is used variously to refer to the instruments, methods, and units that are candidates for use in obtaining and stating the values of quantities and metrics.

Finally, a *controlled variable* is any property that could in principle change, but that has been deliberately kept constant within the scope of the experiment.

Any terms that remain unclear should be interpreted in light of the International Vocabulary of Metrology (VIM) [8], the design of experiments literature, the statistics literature, and the terms' ordinary dictionary definitions.

3 Validation and selection of metrics and measures

3.1 Environment

The hardware used throughout the experiment was a Dell Precision T5400 with dual quad-core Xeon X5450 CPUs and 4 GiB of DDR2-667D ECC RAM.

Testing was conducted under two versions of 64-bit Slackware with replacement kernels. The two environments, hereafter known as os 0 and os 1, are identified in Table 1.

All data were collected while running in single-user mode to reduce interference. The kernel's HZ value was set to 1000 because the resolution of CPU time results from bash's `time` command is limited by this configurable. The value 0 was written to `/proc/sys/kernel/perf_cpu_time_max_percent` on every boot to disable a new (since 3.11) kernel feature that interfered with data collection.

SpeedStep was disabled in the BIOS configuration so that the CPUs would operate at a fixed frequency.

C and C++ test programs were built using GCC version 4.6.4 at optimization level –O2. GCC 4.6.4 was used instead of a current version because GCC 4.7 introduced a feature that invalidated the assumptions of frame pointer profiling in the 64-bit mode of the x86 architecture (x86_64) [9] and this will become important

Factor	Description	Level 0	Level 1
os	Environment	Slackware 14.0 Kernel 3.12.6	Slackware 14.1 Kernel 3.14.3

Table 1: Identification of two operating environments.

in the screening experiment. The test suite originated in [2] was used to regression-test and validate this latest release in the 4.6 series.

3.2 Workloads

The programs used to create CPU-intensive workloads were:

- A single-threaded C++ program, *aliquot*, that calculates aliquot sequences [10] using a simple but inefficient algorithm.
- A Ruby script, *aliquot-par.rb*, that distributes the work of the simple algorithm over multiple CPU cores.
- An invocation of the core utility program *dd* that reads data from /dev/urandom, a computationally expensive pseudorandom number source that is implemented in kernel space.
- A single-threaded C program, *10sec-user*, that executes a busy-waiting loop polling the system function `gettimeofday` until the timer readings indicate that at least 10 seconds have elapsed. `gettimeofday` is implemented in user space [11].

Workloads 0 through 4 as used in the experiment were as follows:

0. Run *aliquot* to calculate the 154-term aliquot sequence beginning with 10^7.
1. Id. except with the `nosmp` kernel parameter and `dd if=/dev/urandom of=/dev/null` running continuously as a competing load.
2. Run *aliquot-par.rb* to verify that $2^{30} \cdot \left(2^{31} - 1\right)$ is a "perfect number" (an aliquot cycle of length 1) using all 8 CPU cores.
3. Run *dd* to copy 5 blocks of size 10 MiB from /dev/urandom to /dev/null.
4. Run *10sec-user*.

3.3 Ground truth

Even if all of the measures tested proved to be consistent with one another, there would remain the question of whether every possible mechanism on the computer ultimately traced to a clock which could, for example, run 10 % fast or slow without drawing attention.

The purpose of *10sec-user* is to realize a quantity close to 10 seconds of elapsed time and incidentally somewhat close to 10 seconds of CPU time (if there is no competing load) to serve as a reference or *étalon* for the measures being tested. Regardless of variations in the speed at which the polling loop is executed, the program terminates when the clock indicates that 10 s have elapsed.

Still, it remained necessary to validate that *10sec-user* realizes the desired quantity. As a primitive validation, the elapsed execution time of *10sec-user* was measured 10 times using a digital wristwatch and a script that simply printed "Start" and "Stop" before and after the program ran. Given the uncertainty contributed by human reaction time and gained proficiency in activating the buttons on the wristwatch, the consistency of the results was impressive:

10.07 9.99 9.94 10.00 10.01 10.00 10.01 10.01 10.01 9.99

Although such a small sample proves nothing with regard to outliers, it does provide some independent if imprecise validation that *10sec-user* realizes a quantity of 10 seconds elapsed time. Unfortunately, being entirely a function of internal state, CPU time is not independently observable.

3.4 Metrics

Two (or essentially three, as described below) different instruments were employed to collect a variety of related metrics. Bash is the default command-line shell used by Linux. Perf [12] is a high-powered profiling toolset that is included in the Linux kernel source tree.

For the purposes of this discussion, the names of the metrics have been prefixed by "bash" or "perf" to identify the source.

3.4.1 bash time

Metrics: bash-elapsed, bash-user, bash-sys.

The bash shell has a built-in time command that provides one measure of elapsed time and two measures of CPU time. The two measures of CPU time indicate "user" time and "system" time respectively. Results are reported to a maximum numerical precision of 0.001 s; however, the resolution of the CPU time metrics is limited by the kernel's compiled-in HZ configurable, which we set to be 1000 Hz.

3.4.2 perf stat

Metrics: perf-elapsed, perf-cpu-cycles, perf-cpu-clock, perf-task-clock.

`perf stat` provides elapsed time to 9 decimal places as well as the final values of a user-configurable subset of counters. The supported counters include hardware counters implemented by the CPU and chipset, software counters implemented by the kernel, and tracepoint statistics. For this experiment we chose the counters cpu-cycles, cpu-clock and task-clock.

perf-cpu-cycles is a count of CPU cycles that traces to a hardware counter. If the CPU frequency is fixed (as ours was), the CPU cycle counts can be converted to seconds by dividing by the CPU frequency.

perf-cpu-clock and perf-task-clock are both software counters with results provided in milliseconds.

The practical differences among these counters being unclear, it was a secondary goal of the validation to find which ones would be most appropriate for subsequent uses.

3.4.3 perf record

Metrics: perf-samples.

While `perf stat` reads counters to provide totals for an entire program execution, `perf record` uses such counters to drive sampling-based profiling. These distinct functions cannot simultaneously be applied to a given program execution unless one nests the perf invocations.

Although `perf record` is normally used for function-level profiling, the reports generated include a sample count total that can be used as a measure of the whole program. Without nested measurements, it is the only such measure that is available after a `perf record` profiling run.

The sample counts can be converted to seconds by dividing by the sampling frequency. However, `perf record` entails periodic interruption of the workload, so measurements of time that are obtained this way may differ from results that are obtained without such interruptions (*e.g.*, by `perf stat`).

3.5 Data collection for metric validation

1000 samples were collected for each combination of metric, workload, and os. Since different tools provide different metrics, it was necessary to run the program three times to obtain one sample for each of the metrics without nesting the tool invocations: once for each of bash, `perf stat`, and `perf record`.

perf-cpu-cycles was converted to seconds by dividing by the CPU frequency, 2.9925 GHz. The `perf record` sample counts were converted to seconds by dividing by the sampling frequency, 10^6 cycles = 2992.5 Hz.

3.6 Results and analysis

Figure 1 shows the means of the validation data for the different metrics with 95 % confidence intervals. Confidence intervals were determined using the bias-corrected and accelerated (BC_a) bootstrap method [13, 14]. The numbers of bootstrap replications were determined using the adaptive method described in [15, §7.9.4] to achieve estimated bootstrap simulation precisions that are below the screen resolutions of the plots. The two or three plots pertaining to a given workload are drawn to the same vertical scale albeit with different locations on the Y axis; each workload, however, is scaled independently due to the significant range differences among them.

Figure 2 and Figure 3 visualize the entirety of the collected data with hybrid violin-scatter plots. From these figures one can see that the tails of the distributions are sparsely populated but quite long. Direct examination of the time series shows that the outliers are not always solitary but sometimes cluster around low-frequency secondary modes such as in the example of Figure 4. Distributions like this were previously observed in [3, §4].

3.7 Face validity

The data for the first three workloads include high outliers for most of the metrics. These outliers are typical of what we have seen previously in performance measurements [1, 3] and are not readily dismissed as invalid results.

On the other hand, the data for Workload 2 in os=1 include extreme low outliers for perf-cpu-cycles, perf-cpu-clock, perf-task-clock, and perf-samples that are implausible. The perf-elapsed results from affected invocations of `perf stat` were close to the mean. We speculate that the parallel execution in Workload 2 occasionally triggered some kind of concurrency-related fault in the perf subsystem that prevented most of the time from being accounted for. The absence of this anomaly in os=0 is consistent with the theory that a fault was introduced between one kernel version and the other.

The fact that the elapsed time measures for Workload 2 are less than the CPU time measures is not an anomaly; it simply means that the parallel workload averaged a CPU utilization greater than one by using more than one CPU core.

3.8 Analysis

Our findings for the validation phase are as follows:

1. bash-elapsed and perf-elapsed differ under some workloads but are equally usable measures of elapsed time.

2. perf-cpu-cycles, perf-cpu-clock, perf-task-clock, and perf-samples all reflect the total CPU time used by the application. This experiment failed to illuminate any material differences that exist among the first three. The impact of the different measurement method used for perf-samples is apparent in its divergence from the others.

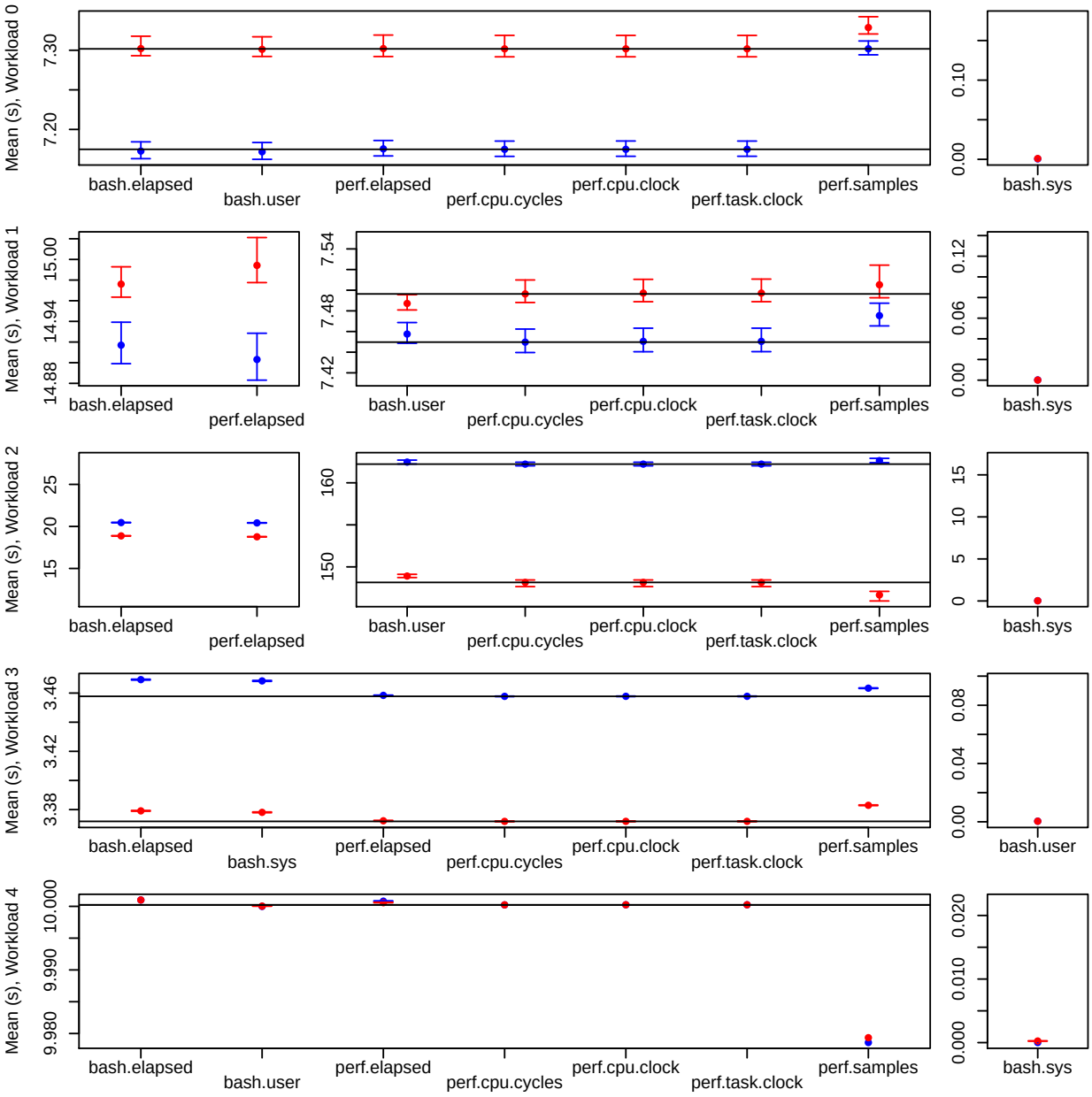

Figure 1: Means of 1000 samples for each combination of metric, workload, and os (os 0=blue, os 1=red) with 95 % BC_a confidence intervals. A horizontal line is drawn at the perf-cpu-cycles mean to show the close agreement of perf-cpu-cycles, perf-cpu-clock, and perf-task-clock. Intervals are not drawn when the range is vanishingly small.

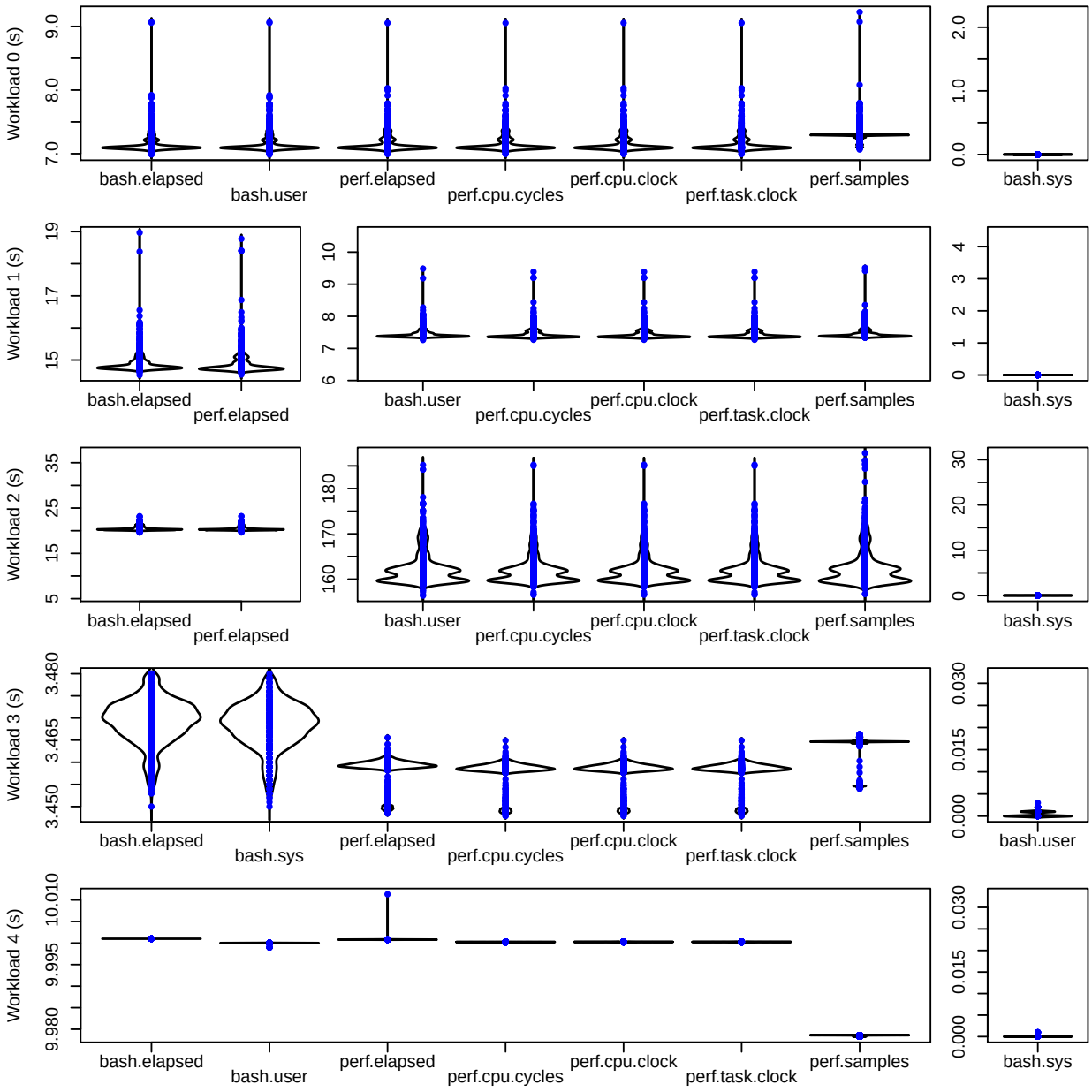

Figure 2: Hybrid violin-scatter plot of 1000 samples for each combination of metric and workload, os=0.

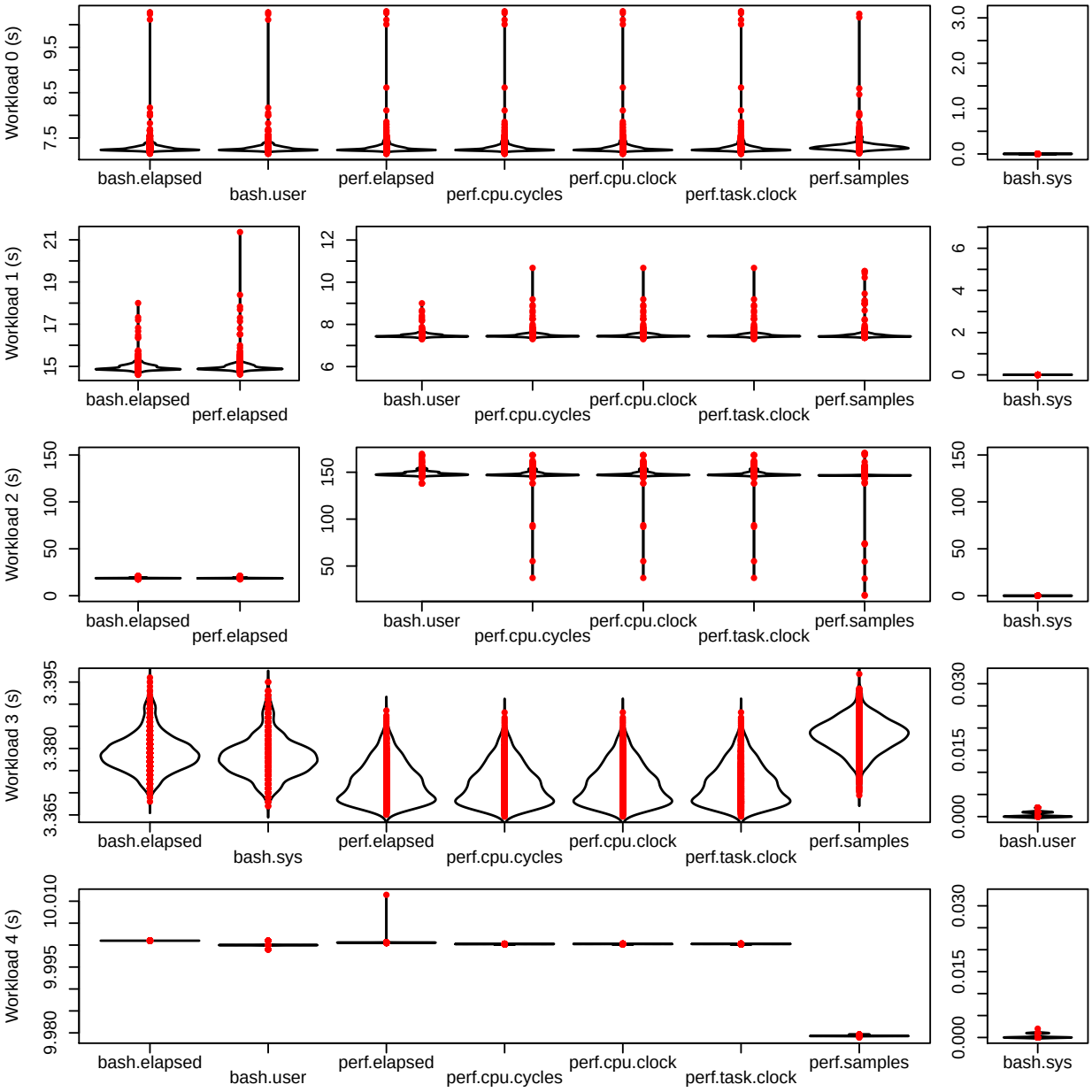

Figure 3: Hybrid violin-scatter plot of 1000 samples for each combination of metric and workload, os=1. The range of the Workload 2 results is not a typo.

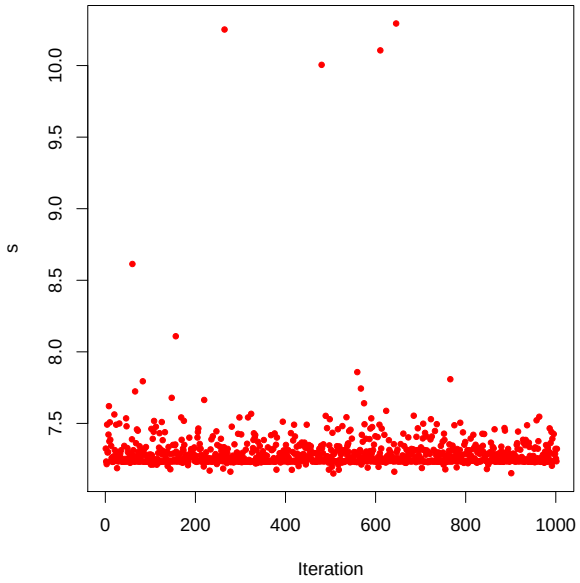

Figure 4: Scatter plot of time series for os=1, Workload 0, metric perf-cpu-cycles.

3. bash-user and bash-sys reflect CPU time that was used while in user space and kernel space respectively. The sum of these measures is comparable to the previous set of total CPU time measures. N.B., perf events can be tweaked to count user space or kernel space only [16]; that capability was not tested in this experiment.

4. For os=1, perf-cpu-cycles, perf-cpu-clock, perf-task-clock and perf-samples are unreliable for parallel Ruby programs, though the failures are rare enough that filtering or averaging over a large number of measurements suffices to mitigate them.

5. Consistent with our previous findings [1, 3], outliers occur at such a frequency and at such distance from the main cluster as to cast doubt on any conclusions drawn from small samples. Not even the simplest elapsed time metrics or CPU time totals are immune from excursions.

6. The significant reduction in variability that appeared in Workload 4 with *10sec-user* dynamically adjusting itself to terminate after the same amount of elapsed time is evidence that the plausible outliers previously observed represent actual variations in the execution performance of fixed workloads instead of artifacts of the measurements. The lone outliers of perf-elapsed ≈ 10.01 s (which is hardly a large excursion from the mean in absolute terms) occurred as the first measurement of the series in each os, possibly reflecting one or more one-time overheads such as loading the executable binary into the file system cache. (`perf stat` happened to be the first of the three instruments to run.) Omitting that one value reduces the standard deviation of the samples by an order of magnitude.

An alternate explanation for the reduction in variability that appeared in Workload 4 is that the variability resulted from fluctuations in the underlying clock used for measurement and the dynamic adjustments made by *10sec-user* merely pegged the workload to the same fluctuating clock. With SpeedStep being disabled and the variability appearing in elapsed time and CPU time measures alike, this alternative seems less plausible.

4 Screening experiment

We now proceed to the main experiment. Its primary goal was to identify factors that affect the performance of an application being profiled. Profiling options being one of the factors of interest, it also enabled a comparison of the relative impacts of two different methods of profiling in perf.

One method collects call chains using frame pointers, a legacy x86 run-time feature/protocol that is by default not done on x86_64 but that can be enabled with a compiler switch. This method requires the application under test to implement a frame pointer stacking protocol at run time. The other method collects call chains using static call frame information that recent compilers will include in the DWARF debugging data [17] that are typically included in the executable. This method requires no run-time actions by the application at all.

Prima facie, it would appear that the DWARF method is better. Profiling using frame pointers is known to produce spurious call chains some fraction of the time [2, 18]. This inaccuracy was exacerbated on x86_64 by a change in GCC 4.7 that invalidated the assumptions of frame pointer profiling [9]; hence the reason that GCC version 4.6.4 was used for this experiment as noted in Section 3.1. Profiling using DWARF is immune to these call chain anomalies. Furthermore, being more likely to work with the as-distributed executables of a software package (particularly on x86_64), DWARF profiling eliminates some validity questions that arise when the software as tested is not the same as the software as deployed. Nevertheless, some users prefer to use frame pointer profiling on the grounds that DWARF profiling incurs too much overhead [19].

The population about which the results of this experiment were expected to provide insight is the population of CPU-intensive, non-parallelized programs. For our representative process to profile we chose the aliquot workload known as Workload 0 in Section 3.

4.1 Experimental design

In this experiment there were constraints on the feasible combinations of factor levels (a failure of orthogonality) and on the feasible testing sequences. However, it was practically feasible to cover all meaningful combinations of factors, so a straightforward "full" design covering all meaningful treatments was used. N.B., in general, if there are not conflicting constraints on the combinations of factor levels or the testing sequence, a two-level fractional factorial experimental design [20, 21, 22, 23] is more efficient for screening.

4.2 Independent variables

Table 2 shows the factors and levels for the experiment.

For level 0 of the fp factor, the application was the same compiled C++ *aliquot* binary used in Section 3. For level 1, it was compiled with –fno-omit-frame-pointer. Note that frame pointers were added only to the workload binary, not to the kernel or shared libraries.

4.3 Controlled variables

Any aspect that is not mentioned in Table 2 was held constant in the same configuration that was used in Section 3.

4.4 Dependent variables

Given the focus of this experiment, the results in Section 3 did not indicate a strong preference for any of the tested metrics over any other. On the other hand, we were interested in the overhead of call chain profiling, which is mutually exclusive with `perf stat`. This left us with the metrics bash-elapsed, bash-user, bash-sys, and perf-samples to use for dependent variables.

Table 3 lists the 12 resulting treatments and identifies the non-orthogonalities that exist:

1. There can be no data for perf-samples when prf=0. The main effects of os and fp are still balanced, and for prf we are simply missing level 0 for one of the dependent variables. That is assuming that we are not trying to compare the dependent variables against one another, but only to make comparisons among treatments for each dependent variable.

Factor	Description	Level 0	Level 1	Level 2
os	Environment	Slackware 14.0 Kernel 3.12.6	Slackware 14.1 Kernel 3.14.3	
fp	App built with frame pointers	No	Yes	
prf	Perf profiling	No	`--call-graph dwarf`	`--call-graph fp`

Table 2: Independent variables for the screening experiment.

Treatment			Dependent variables		
os	fp	prf	bash-*	perf-samples	Use case
0	0	0	OK	Unobtainable	Valid
0	0	1	OK	OK	Valid
0	0	2	OK	OK	Dysfunctional
0	1	0	OK	Unobtainable	Valid
0	1	1	OK	OK	Valid
0	1	2	OK	OK	Valid
1	0	0	OK	Unobtainable	Valid
1	0	1	OK	OK	Valid
1	0	2	OK	OK	Dysfunctional
1	1	0	OK	Unobtainable	Valid
1	1	1	OK	OK	Valid
1	1	2	OK	OK	Valid

Table 3: Treatments.

2. The treatments in which fp=0 and prf=2 represent a dysfunctional use case in which one is profiling using frame pointers but the application was built without them. We did not make use of the resulting broken call chains in this experiment, and an argument can be made that the measurements of elapsed and CPU time that we did use are still valid in a technical sense. Nevertheless, if it is a use case that ought not be followed in practice, the resulting data are impugned as being unrepresentative of practice. To exclude these treatments unbalances the experiment with respect to two of the three factors. With the design that we used, we have as much flexibility as possible to analyze results both including and excluding the impugned data.

Since we no longer needed to make comparisons among the dependent variables but were rather more interested in the overhead contributed by perf, the invocations of perf were nested within the bash time commands. The bash measurements therefore reflect the combined time of the application and the perf infrastructure, albeit with the possibility that some perf overhead operations may have run concurrently on a different CPU, while perf-samples reflects only the time of the application.

The perf invocations had the following form, specifying sampling with a period of 10^6 CPU cycles and sending the recorded profiling data to a ramdisk mounted on /tmp:

```
perf record -q -e cpu-cycles -c 1000000 -o /tmp/perf_data --call-graph ...
```

The perf-samples metric was then extracted from the report produced by:

```
perf report -i /tmp/perf_data -s dso -g none
```

The collected call chains themselves were not relevant to the experiment.

4.5 Ordering

Switching between the two levels of os requires a reboot, so the data collection was performed in two batches, os=1 followed by os=0. Within each batch, each "iteration" consisted of six executions of the test application—one per treatment, executed in a random order. This ordering helped control for any

macroscopic time dependency (drift) that might have occurred at the system level as well as any local ordering effects that might exist among the treatments.

The fact that the treatments were not run in a completely randomized order technically gives this experiment design a split-plot structure [24]. The next section discusses a side experiment that was run to determine if the run order had any practical impact on the data from the screening experiment.

4.5.1 Boot dependency

"Boot dependency" refers to the theoretical possibility that software performance could have a component that varies randomly from one boot of the computer to the next but remains constant for the period of uptime corresponding to a given boot. The existence of such an effect would be highly inconvenient for our experimental design since it would be prohibitively cumbersome to break the screening experiment over many reboots to assess such an effect with reasonable precision.

To check for boot dependency, we ran a mini-experiment of 20 reboots. We used a nonrandom ordering that balanced the number of occurrences of each os level without simply alternating:

Day 1: 1 0 1 0 0 1 1 0 0 0
Day 2: 1 1 1 0 0 1 1 0 1 0

Each "block" (reboot) collected 101 samples for each of the bash metrics. The workload was the same compiled C++ *aliquot* binary used in Section 3.

The results are plotted in Figure 5. Although the results were affected by the os levels, no boot dependency is evident; the distributions of results appear not to have changed materially from one boot to the next. As a result, the data from the full screening experiment were analyzed as though they came from a completely randomized experiment design.

N.B., the configuration of shared libraries used in this mini-experiment was slightly different than that used in the validation and the main experiment. While that could affect the distributions, the test for a boot effect was concerned only with the consistency of results across multiple boots.

4.6 Results

The following results are based on 5000 iterations (as defined in Section 4.5), producing 5000 values for each dependent variable for each treatment except for perf-samples when prf=0.

4.6.1 Distributions

Figure 6 visualizes all of the data from the screening experiment in hybrid violin-scatter plots. Many of the distributions are again long-tailed and non-Gaussian.

4.6.2 Main effects and interactions

As was noted in the design, there is a conflict between balancing the data and excluding dysfunctional configurations from the treatments. Figure 12 through Figure 27 in Appendix A visualize main effects and 2-factor interactions for the complete 12-treatment experiment as well as three abridged data sets that make different tradeoffs.

The first abridgment, "Minus os=* fp=0 prf=2," excludes the two treatments for the dysfunctional configuration in which callchain profiling using frame pointers is being attempted on an application that was compiled without frame pointers. This makes the experiment unbalanced with respect to fp and prf.

The second abridgment, "Minus os=* fp=* prf=2," restores balance at the cost of losing all data for callchain profiling using frame pointers. 8 treatments remain.

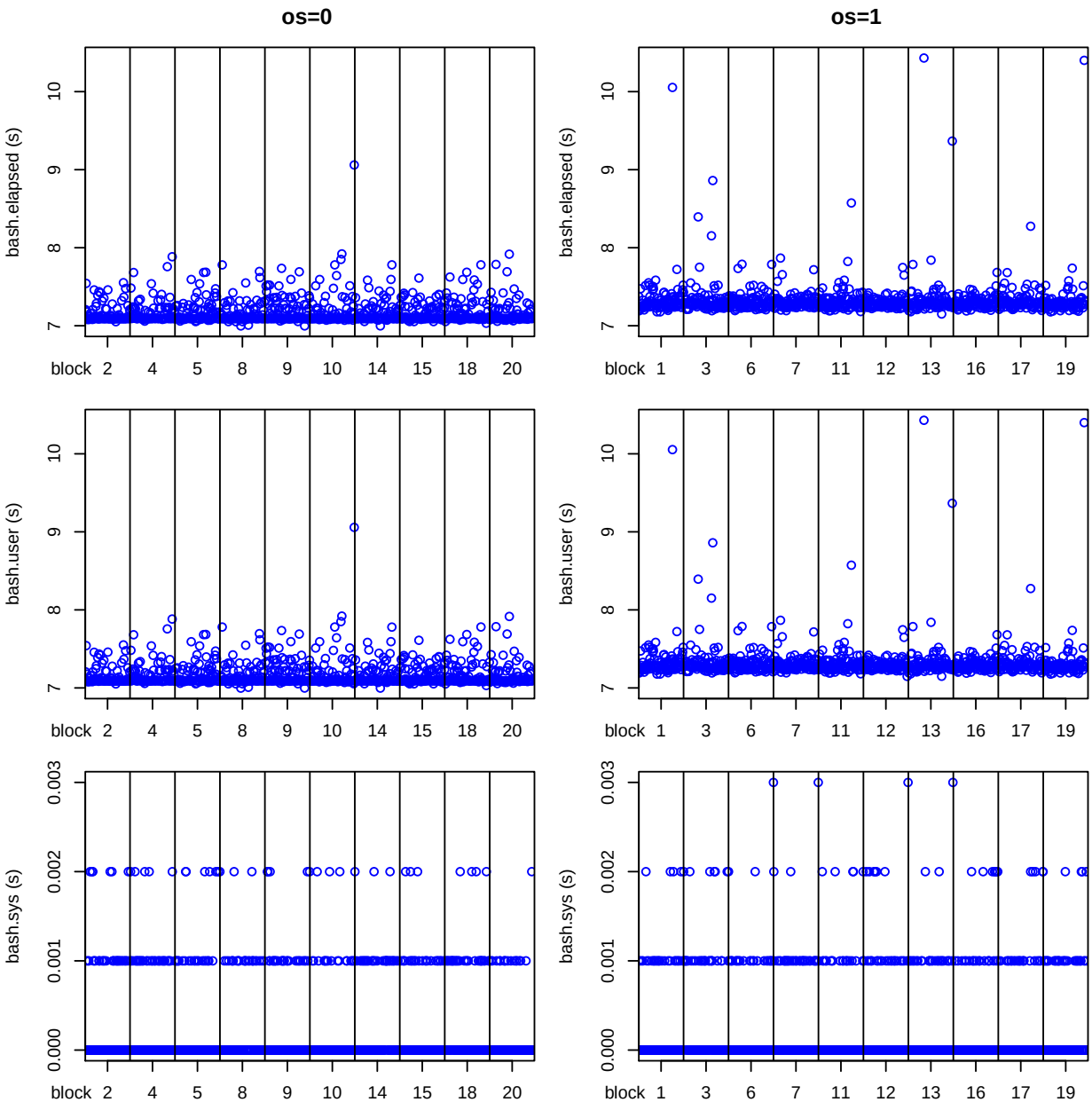

Figure 5: Scatter plots of bash metrics from boot dependency mini-experiment. No boot dependency is evident.

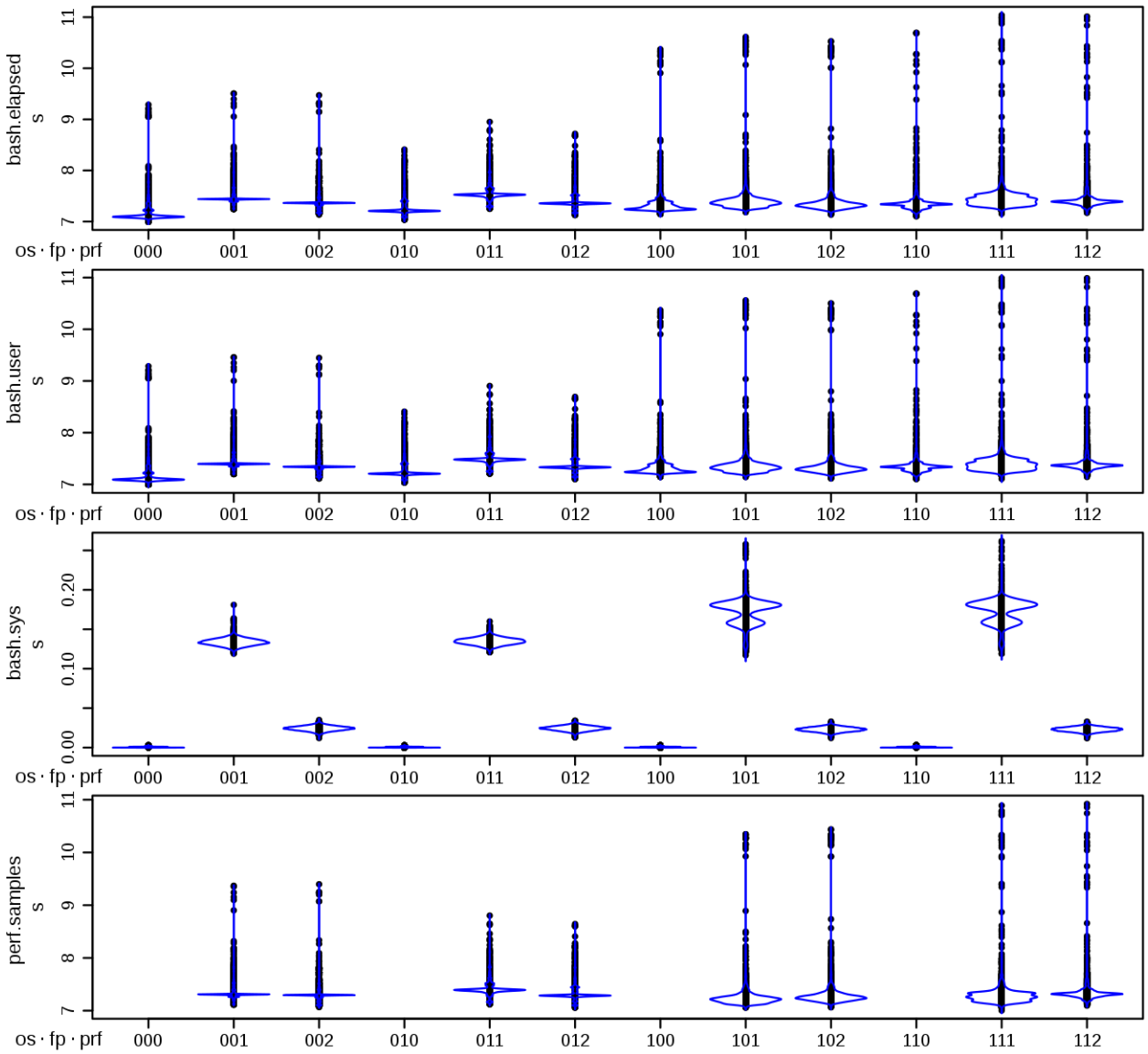

Figure 6: Hybrid violin-scatter plots for all treatments of the screening experiment.

The third abridgment, "Only os=* fp=0 prf=1 and fp=1 prf=2," retains only the 4 treatments that allow a balanced comparison between DWARF profiling of an application without frame pointers and frame pointer profiling of an application that was built with them.

Using this method to search for effects, the number of combinations and abridgements makes for an unfortunate number of figures. The next section demonstrates an alternative method that is more concise.

4.6.3 Block plots

Block plots [25, 26] provide a simple visualization of the significance of different factors in an experiment. One factor (dubbed the "primary factor") is chosen for the vertical axis. All combinations of the levels of the remaining factors are assigned places on the horizontal axis. The results for the treatments that correspond to a given place on the horizontal axis are then plotted inside a vertical box and identified by the level of the primary factor.

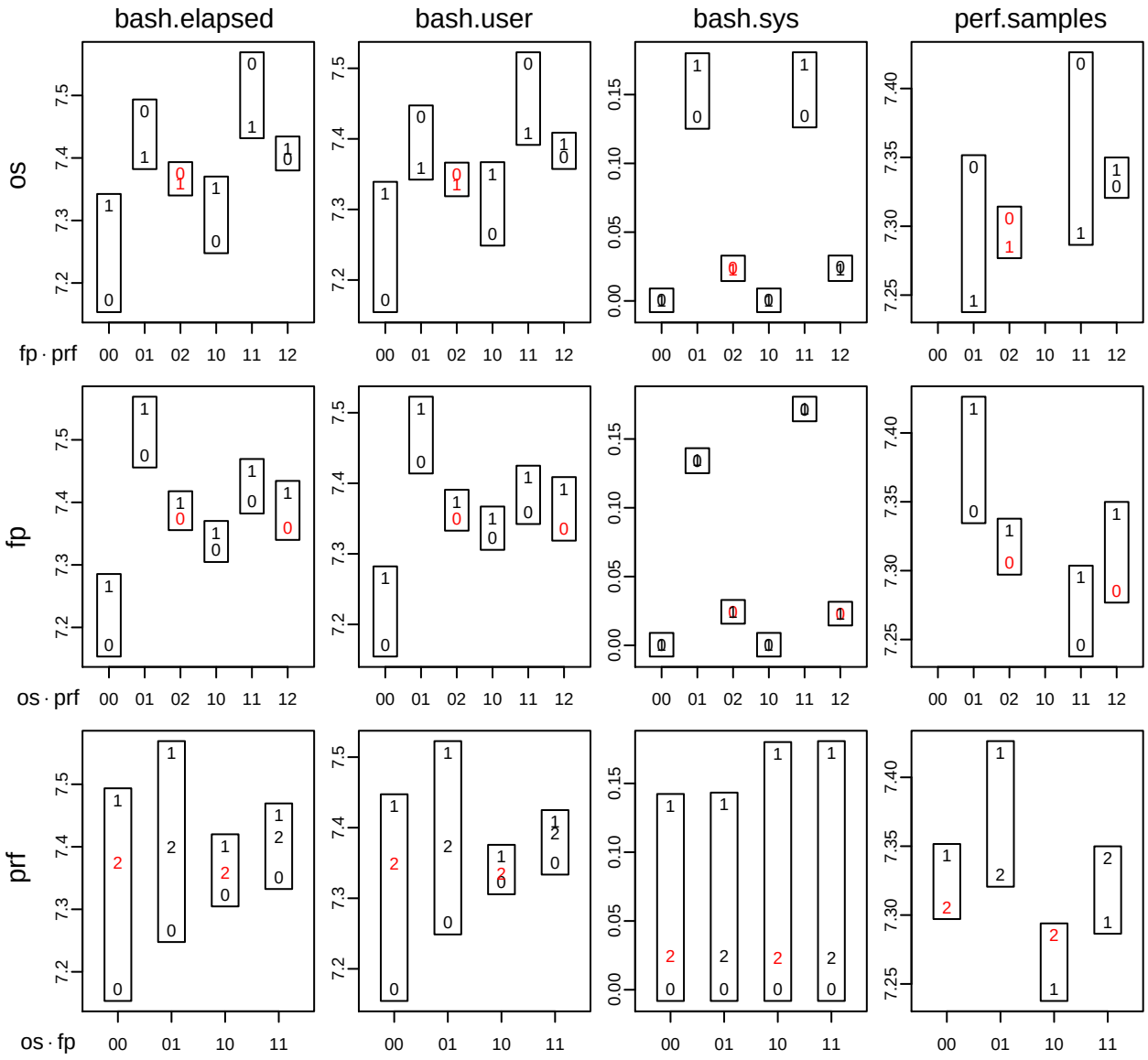

Figure 7: Block plots of treatment means for the screening experiment. Results for "dysfunctional" use cases are colored red.

Although the uncertainty of each individual mean is not shown, it is accounted for indirectly by the interpretation of the plot. A factor is called significant only if there is a deterministic pattern in the ordering of its levels across all of the boxes. If the variability of the means were large enough to be material it would tend to randomize the ordering: the probability of a 2-level factor showing the same ordering across N boxes purely by chance is $1/2^{(N-1)}$.

A particular advantage of the block plot in this experiment is that excluding questionable treatments has a localized and easily understandable impact, whereas with main effects plots, interaction plots, and Analysis of Variance (ANOVA) techniques the impact can be wider and less obvious.

Figure 7 visualizes the complete experiment with the dysfunctional cases shown in red. Figure 8 reduces the experiment in accordance with the third abridgement.

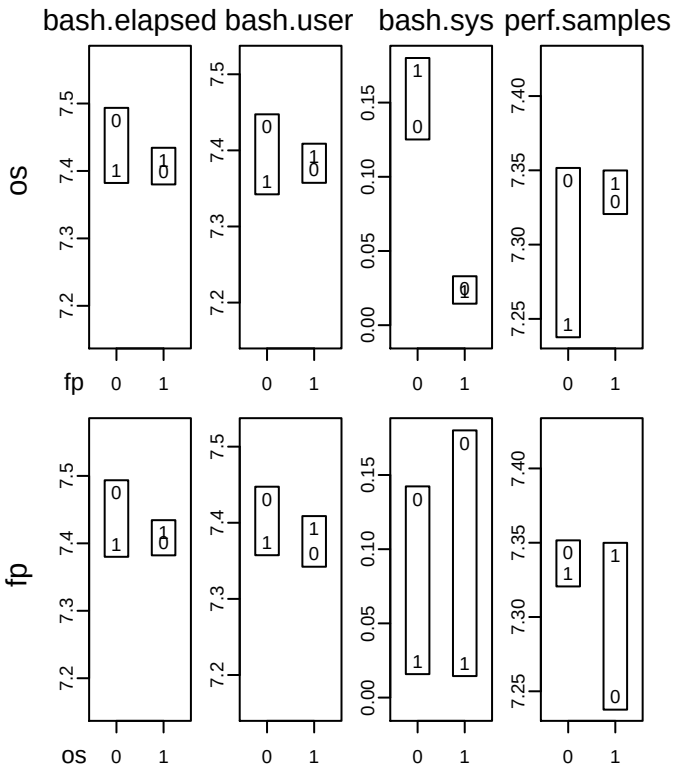

Figure 8: Block plots of treatment means for the DWARF versus frame pointers comparison.

4.7 Analysis

The block plot of Figure 7 shows that the fp and prf factors each had discernible effects. The effect of prf is larger in magnitude but it emerges only in the bash metrics. Relative to no profiling, profiling with frame pointers incurred a performance penalty and profiling with DWARF incurred a greater penalty, but this did not consistently result in a higher perf sample count.

According to the plots in Appendix A.1,

1. In the bash metrics, the operating system upgrade incurred a performance penalty. perf-samples decreased.
2. In three of the four measures, adding frame pointers to the application incurred a performance penalty. bash-sys was unaffected.
3. In the bash metrics, which due to the nesting of instruments discussed in Section 4.4 measured both the perf infrastructure and the application together, profiling with frame pointers incurred a performance penalty and profiling with DWARF incurred a greater penalty.
4. There was interaction between the os factor and the other two factors.

Unbalanced as it is, the first abridgment shows few disagreements with the complete experiment. When balance is restored in the second abridgment, the apparent disagreements go away.

The third abridgment addresses the choice between building and profiling with frame pointers or omitting frame pointers and profiling with DWARF. As implemented in the versions tested, `--call-graph dwarf` resulted in over 180 megabytes of profiling data being written during a typical 7.4-second run of the application while `--call-graph fp` produced little over one megabyte of profiling data under the same conditions. The flood of profiling data alone would explain a large difference in application run time. Nevertheless, such a difference did not appear consistently at different levels of os.

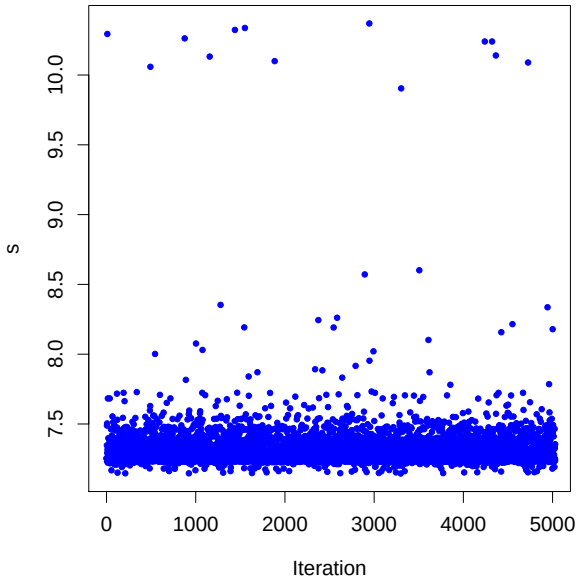

Figure 9: Scatter plot of bash-user time series for treatment os=1, fp=0, prf=0.

We expect that the fp overhead would have been more significant if the kernel and shared libraries had also been compiled with –fno-omit-frame-pointer, but it is not always necessary to profile those objects.

4.7.1 On material significance

With the exception of bash-sys, the ranges of the dependent variables as shown in the violin-scatter plots are so much greater than the differences between the means in Appendix A that it raises the question of whether any of the effects are materially, rather than statistically, significant.

For the 12 main effects of the complete experiment (3 factors times 4 dependent variables), the median effect magnitude was 0.053 s and the maximum (bash-elapsed for prf=1 versus prf=0, admittedly not the most interesting comparison) was 0.19 s. Using treatment os=1, fp=0, prf=0 and dependent variable bash-user as a test case, we subsampled the original data (shown in Figure 9) to determine how large a sample would need to be in order for the measured effects to matter. For small samples, the magnitude of the largest effects may still exceed the estimated uncertainty of the mean (Figure 10), but the accuracy of those estimates as indicated by attained coverage of the mean of the original 5000 samples using nominally 95 % confidence intervals is less than advertised (Figure 11).[3] The coverage failure in this case is consistent with the findings of a previous simulation study of non-Gaussian distributions having low-frequency secondary modes [3, §5.1].

Coverage failure notwithstanding, the overly optimistic uncertainties are still large enough to obscure most of the main effects most of the time with a sample size of 40. Reliable detection of the main effects requires hundreds of samples. So in this example, a casual benchmarker who is not motivated to run the application more than 40 times (as a very charitable lower bound) would have bigger problems than the extra overhead of frame pointers.

5 Conclusion

This report described a screening experiment that showed that three factors (environment, frame pointers, and profiler options) all affected the performance of the application under test. According to 3 of 4 measures,

[3] In these plots, "original interval" refers to a confidence interval derived using the t-distribution and an assumption that the distribution of the mean of a given number of repeated measurements is approximately normal (called the "original method" in [3]).

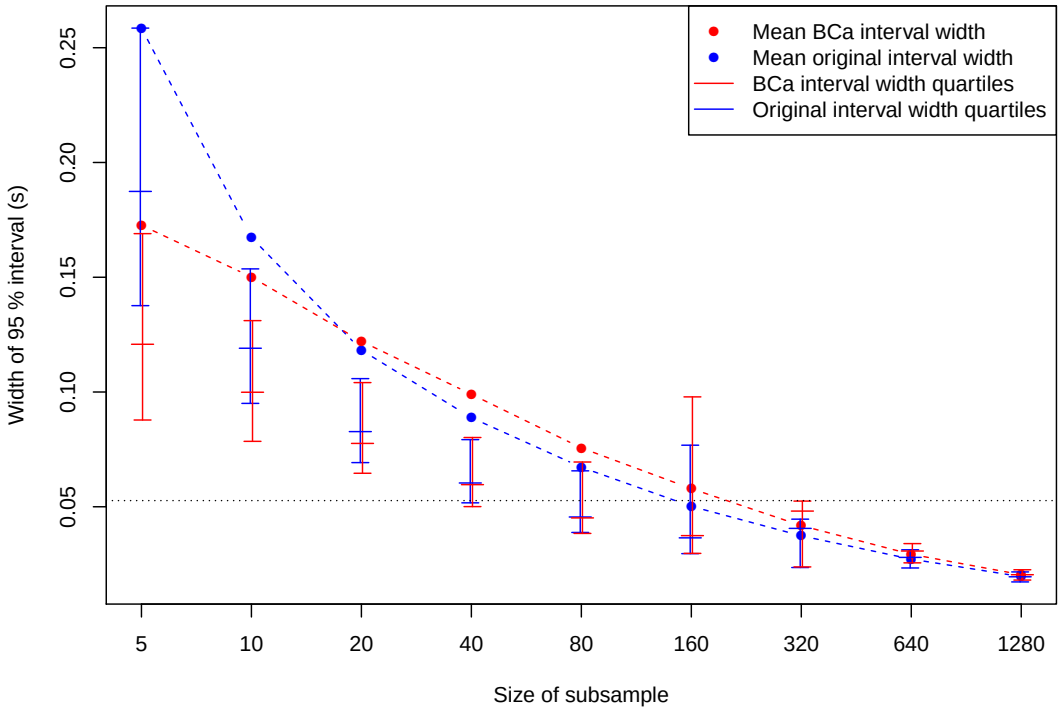

Figure 10: Widths of 95 % confidence intervals for the means resulting from different sizes of subsamples from the 5000 data for treatment os=1, fp=0, prf=0, dependent variable bash-user. The "error bars" indicate the first, second, and third quartiles of the sample of 15 000 interval widths. The horizontal dotted line indicates the median magnitude of main effects found in the experiment. A constant 50 000 bootstrap replications were used for the BC_a intervals.

the operating system upgrade incurred a performance penalty. According to 3 of 4 measures, adding frame pointers to the application incurred a performance penalty. Predominantly, profiling with DWARF incurred greater overhead than profiling with frame pointers, but the effect measured was not robust across different os levels.

Because the distributions of measurement results have non-Gaussian shapes and wide ranges, experiments using smaller samples would be subject not only to greater statistical uncertainty but also to an elevated risk of coverage failure in summary statistics. Consequently, in experiments that nevertheless use small samples to measure performance, the overheads that we measured are likely to be immaterial.

Considering the strong reasons to use DWARF instead of frame pointers that were discussed in Section 4, greater overhead would not automatically outweigh a requirement for accurate call chains or a need to profile an executable that was built without frame pointers as it was deployed. If the impact of overhead is significant in a particular measurement, it may be possible to include a correction for it. N.B., some optimization of the DWARF processing chain is in the works [27], as is a third option (besides fp and DWARF) that relies on new features of recent Intel processors [28].

The validation phase of the experiment yielded evidence that the ill-behaved distributions of results are attributable to actual performance variations rather than measurement error. Such distributions have occurred in most of the project's experiments thus far and on operating systems other than Linux, but the evidence presented here does not generalize beyond Linux.

Finally, in addition to these results on profiling overhead, we have provided an example application of screening techniques in experimental computer science. Together with the cited references, we hope that it will encourage other computer scientists to make more use of established statistical methods in performance evaluations.

The raw data from the experiment are available from the Software Performance Project's web page [4].

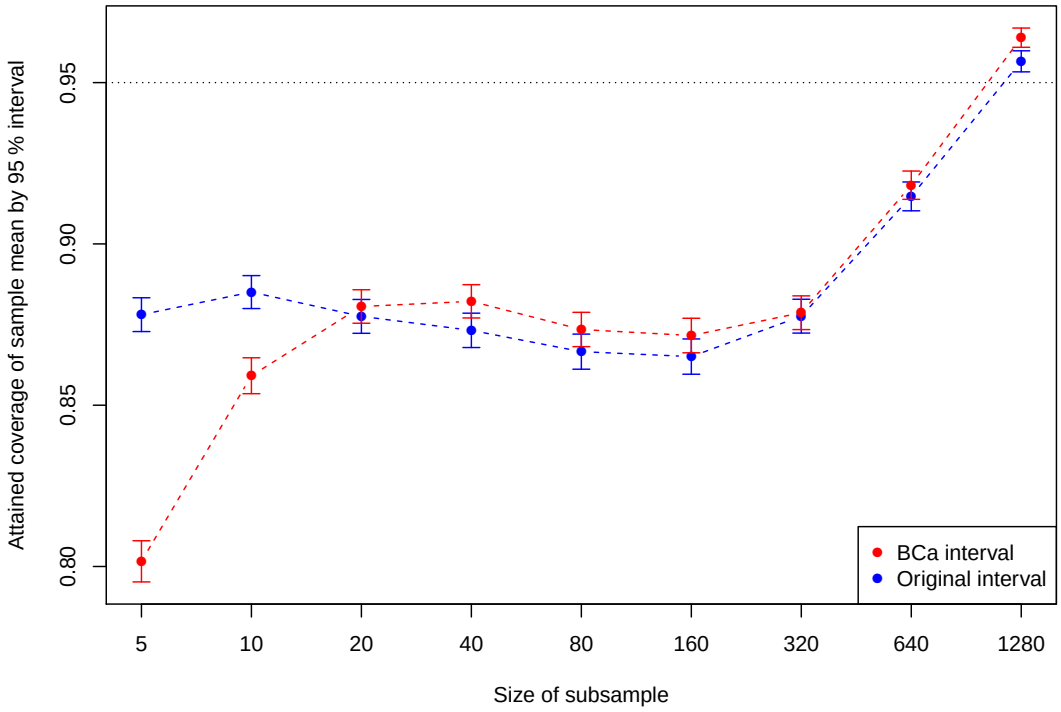

Figure 11: Coverage of the sample mean of the 5000 data for treatment os=1, fp=0, prf=0, dependent variable bash-user, attained by nominally 95 % confidence intervals for the means resulting from different sizes of subsamples. The error bars indicate 95 % confidence intervals for the coverage estimates according to the binomial distribution for $n = 15\,000$ and the estimated value of p.

Acknowledgments

Thanks to William F. Guthrie and Jim Filliben of the Statistical Engineering Division for extensive consultation.

Special thanks to Barbara Guttman and Paul Black for group management and operational support to make this research possible.

References

[1] David Flater and William F. Guthrie. A case study of performance degradation attributable to run-time bounds checks on C++ vector access. *NIST Journal of Research*, 118:260–279, May 2013. http://dx.doi.org/10.6028/jres.118.012.

[2] David Flater. Configuration of profiling tools for C/C++ applications under 64-bit Linux. NIST Technical Note 1790, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, March 2013. http://dx.doi.org/10.6028/NIST.TN.1790.

[3] David Flater. Estimation of uncertainty in application profiles. NIST Technical Note 1826, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, January 2014. http://www.nist.gov/customcf/get_pdf.cfm?pub_id=914648.

[4] Software Performance Project web page, 2014. http://www.nist.gov/itl/ssd/cs/software-performance.cfm.

[5] R package bootBCa, 2014. http://bootbca.r-forge.r-project.org.

[6] Gem for Experimental Computer Science (GECS), 2014. https://rubygems.org/gems/GECS.

[7] NIST/SEMATECH. Screening designs, in NIST/SEMATECH e-Handbook of Statistical Methods, §5.3.3.4.6, April 2012. http://www.itl.nist.gov/div898/handbook/pri/section3/pri3346.htm.

[8] JCGM 200. *International vocabulary of metrology—Basic and general concepts and associated terms (VIM)*. Joint Committee for Guides in Metrology, 3rd edition, 2012. http://www.bipm.org/en/publications/guides/vim.html.

[9] David Flater. GCC Bug 55667 [4.7 regression] –O1 enables frame pointer push to move around on x86_64, December 2012. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=55667.

[10] Wikipedia. Aliquot sequence, 2014. https://en.wikipedia.org/wiki/Aliquot_sequence.

[11] Stack Overflow. What are vdso and vsyscall?, November 2013. https://stackoverflow.com/questions/19938324/what-are-vdso-and-vsyscall.

[12] Perf: Linux profiling with performance counters, 2012. https://perf.wiki.kernel.org/.

[13] Bradley Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171–185, March 1987. http://www.jstor.org/stable/2289144. See also the comments and rejoinder that follow on pages 186–200, http://www.jstor.org/stable/i314281.

[14] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, 1993.

[15] Joint Committee for Guides in Metrology. *Evaluation of measurement data—Supplement 1 to the "Guide to the expression of uncertainty in measurement"—Propagation of distributions using a Monte Carlo method*. JCGM 101:2008, http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf.

[16] Perf version 3.14.3 manual. perf-list—list all symbolic event types, May 2014. Linux man page, available from command line by typing `man perf-list`.

[17] The DWARF debugging standard website, 2012. http://dwarfstd.org/.

[18] John Levon et al. Interpreting call-graph profiles, in OProfile manual, 2012. http://oprofile.sourceforge.net/doc/interpreting-callgraph.html.

[19] Andi Kleen. Re: Two questions on perf dwarf callchains. In perf-users mailing list, January 2014. http://www.mail-archive.com/linux-perf-users@vger.kernel.org/msg01437.html.

[20] Martin Schatzoff. Design of experiments in computer performance evaluation. *IBM Journal of Research and Development*, 25(6):848–859, November 1981.

[21] Richard F. Gunst and Robert L. Mason. Fractional factorial design. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(2):234–244, 2009.

[22] NIST/SEMATECH. Fractional factorial designs, in NIST/SEMATECH e-Handbook of Statistical Methods, §5.3.3.4, April 2012. http://www.itl.nist.gov/div898/handbook/pri/section3/pri334.htm.

[23] Douglas C. Montgomery. *Design and analysis of experiments*. John Wiley & Sons, Inc., New York, USA, 8th edition, 2013.

[24] NIST/SEMATECH. Split-plot designs, in NIST/SEMATECH e-Handbook of Statistical Methods, §5.5.5, April 2012. http://www.itl.nist.gov/div898/handbook/pri/section5/pri55.htm#SplitPlot.

[25] J. J. Filliben, S. Cetinkunt, W. L. Yu, and A. Donmez. Exploratory data analysis techniques as applied to a high-precision turning machine. In Way Kuo and Marcia Martens Pierson, editors, *Quality Through Engineering Design*, pages 199–223. Elsevier, New York, 1993.

[26] NIST/SEMATECH. Block plot, in NIST/SEMATECH e-Handbook of Statistical Methods, §1.3.3.3, April 2012. http://www.itl.nist.gov/div898/handbook/eda/section3/blockplo.htm.

[27] Jiri Olsa. [PATCH 00/16] perf tools: Add libdw DWARF unwind support, January 2014. http://marc.info/?l=linux-kernel&m=138909914404574.

[28] Zheng Z. Yan. [PATCH v3 00/14] perf, x86: Haswell LBR call stack support, February 2014. http://marc.info/?l=linux-kernel&m=139270367113826.

A Main effect and interaction plots

In all of the following plots, the means of the data refactored according to the pertinent effects are shown with 95 % BC_a confidence intervals.

A.1 Complete screening experiment

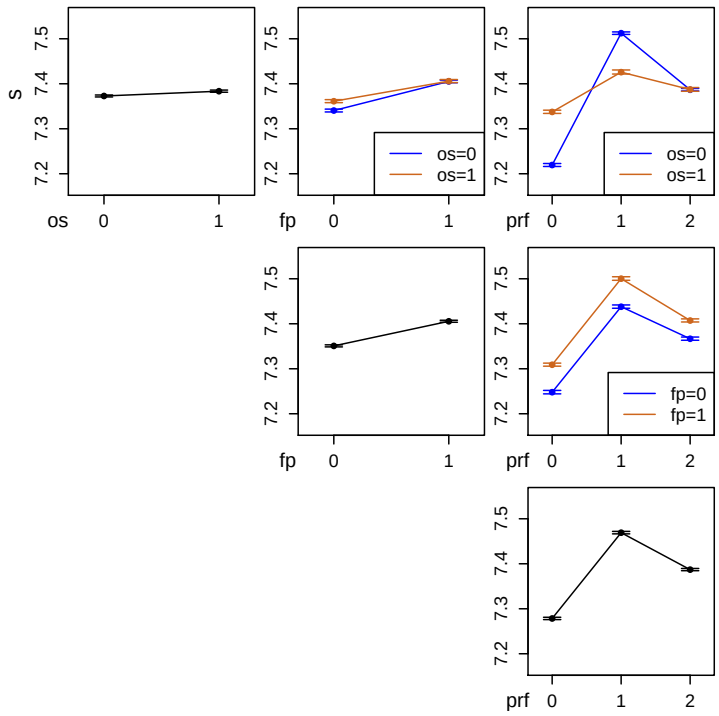

Figure 12: Main effects and interactions for bash.elapsed.

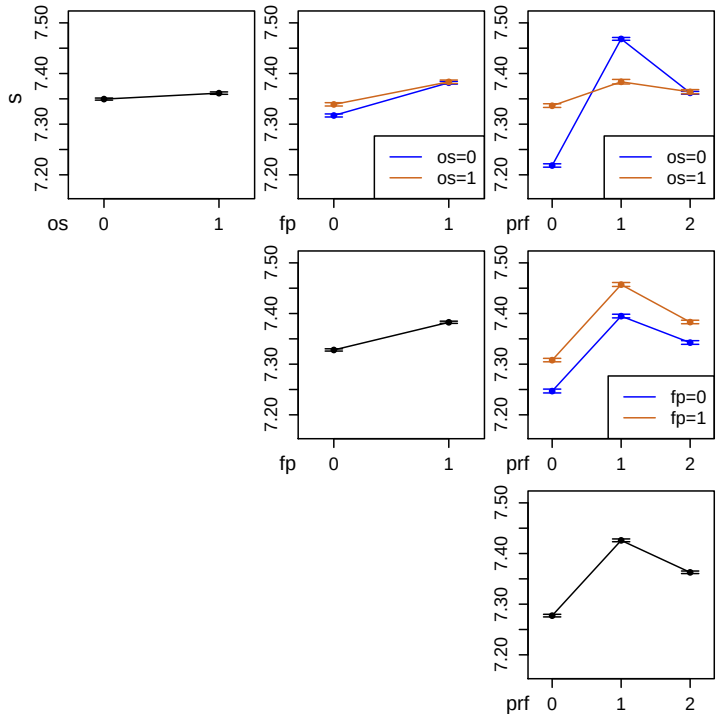

Figure 13: Main effects and interactions for bash.user.

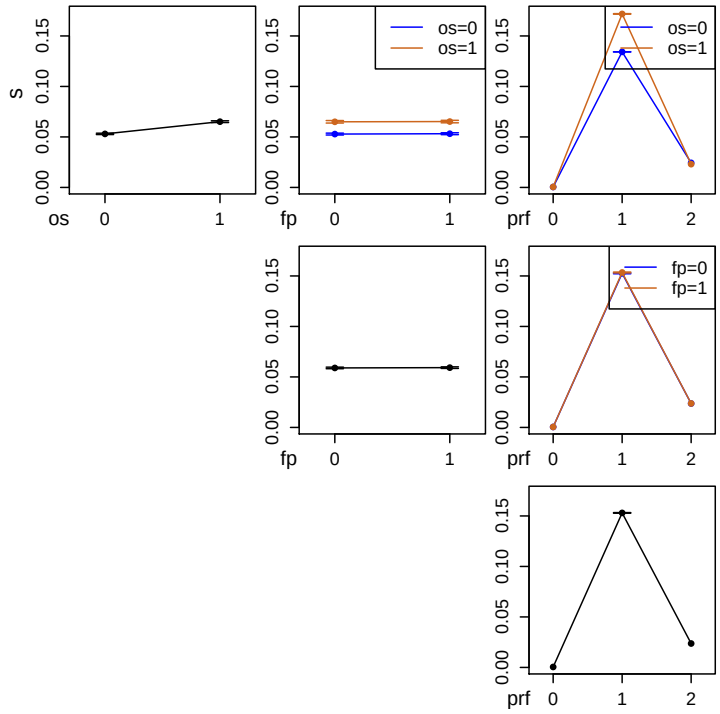

Figure 14: Main effects and interactions for bash.sys.

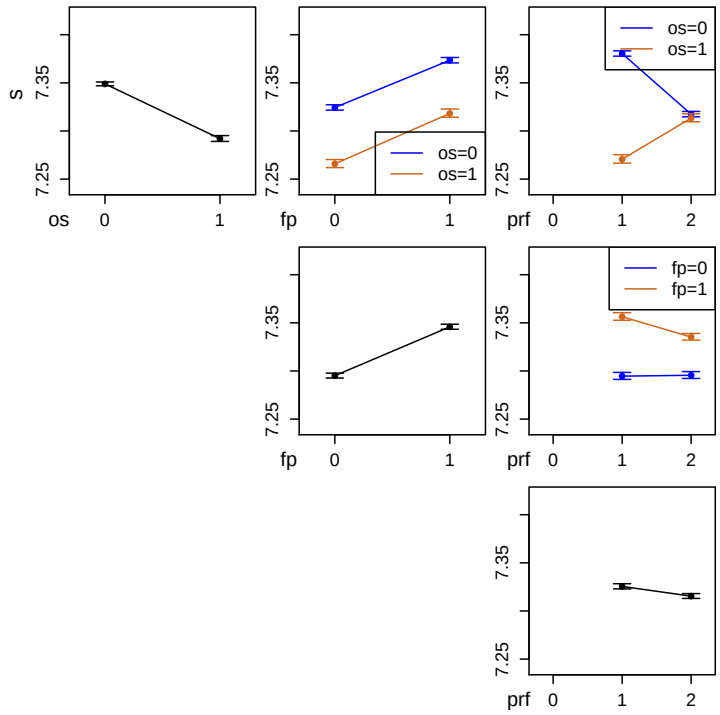

Figure 15: Main effects and interactions for perf.samples.

A.2 Minus os=* fp=0 prf=2

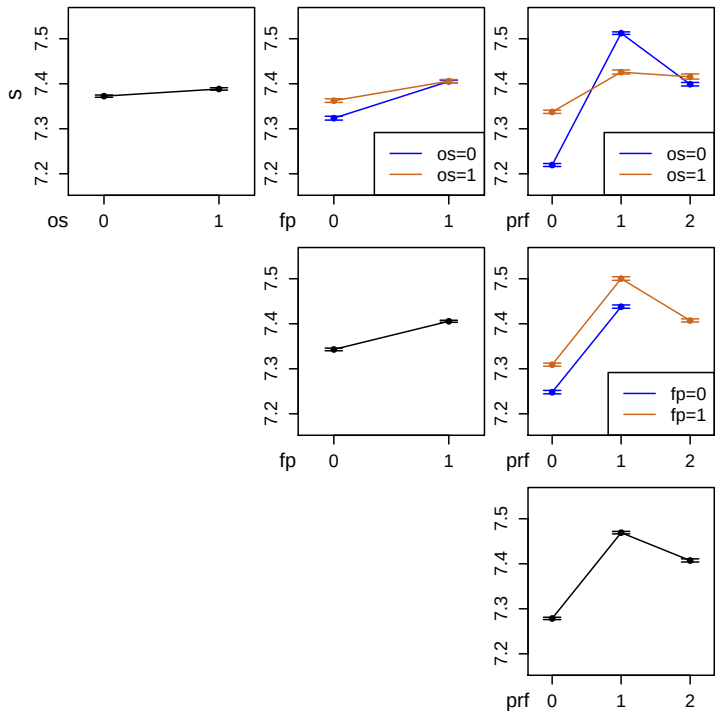

Figure 16: Main effects and interactions for bash.elapsed.

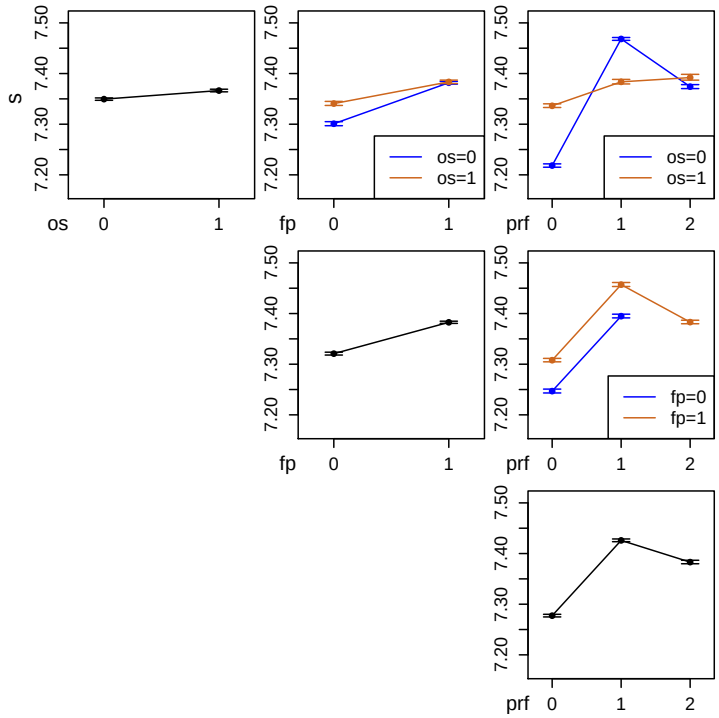

Figure 17: Main effects and interactions for bash.user.

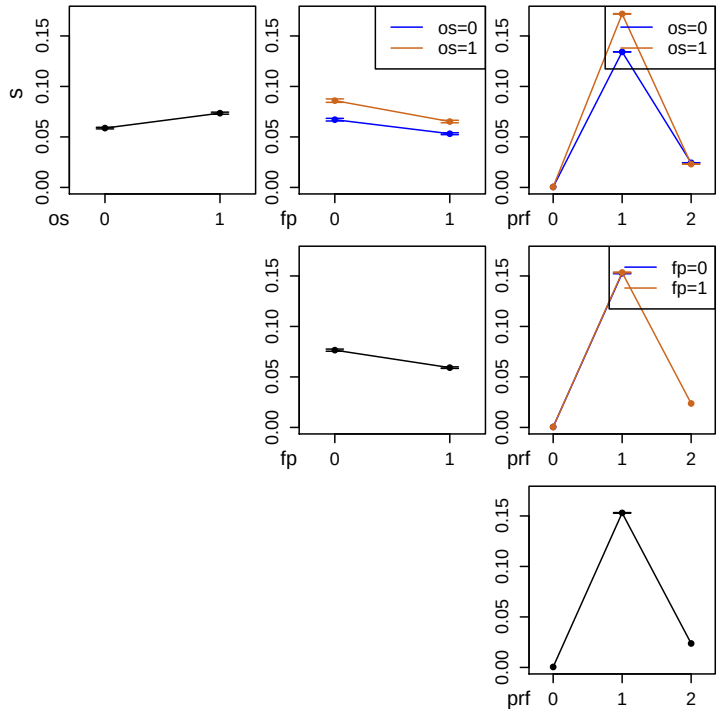

Figure 18: Main effects and interactions for bash.sys.

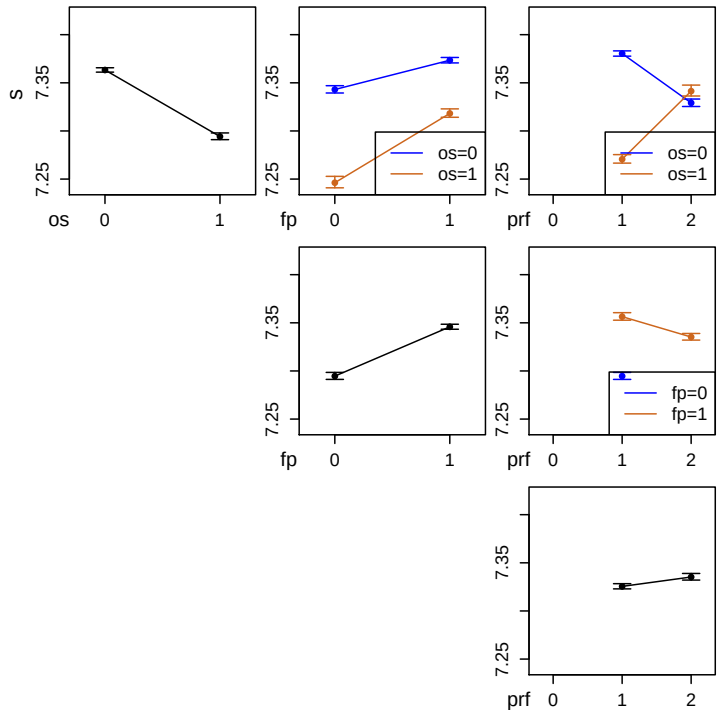

Figure 19: Main effects and interactions for perf.samples.

A.3 Minus os=* fp=* prf=2

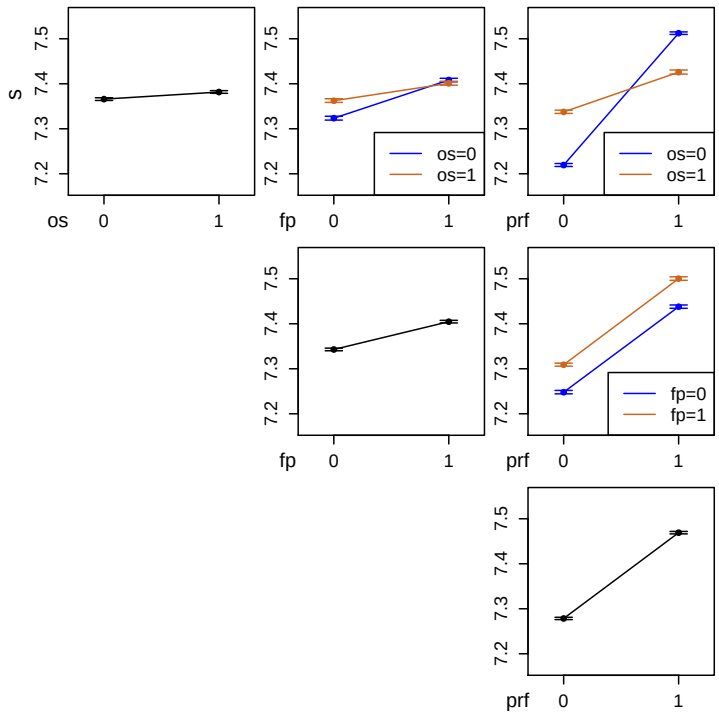

Figure 20: Main effects and interactions for bash.elapsed.

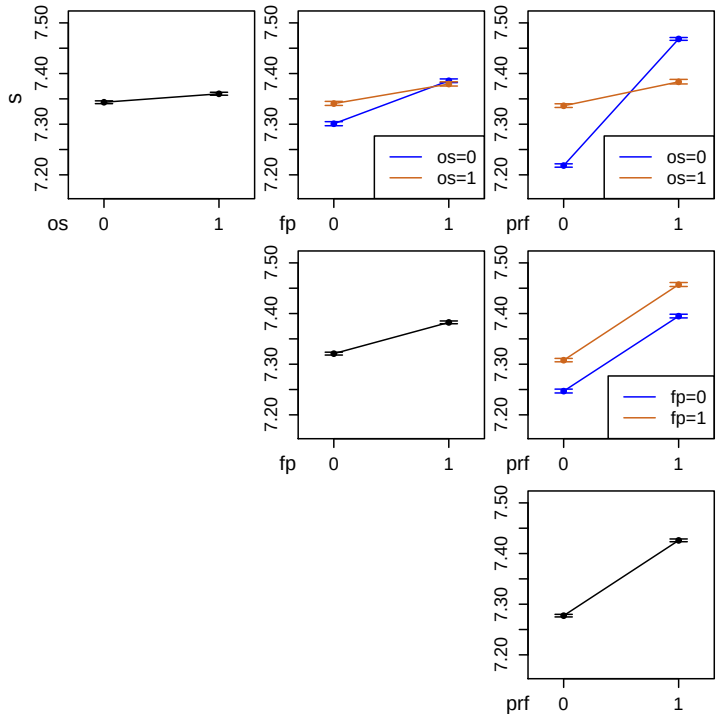

Figure 21: Main effects and interactions for bash.user.

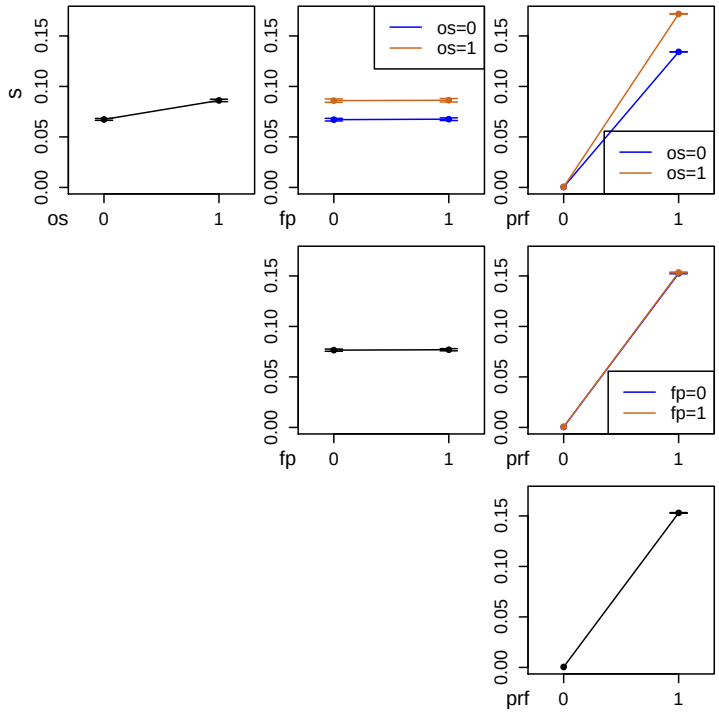

Figure 22: Main effects and interactions for bash.sys.

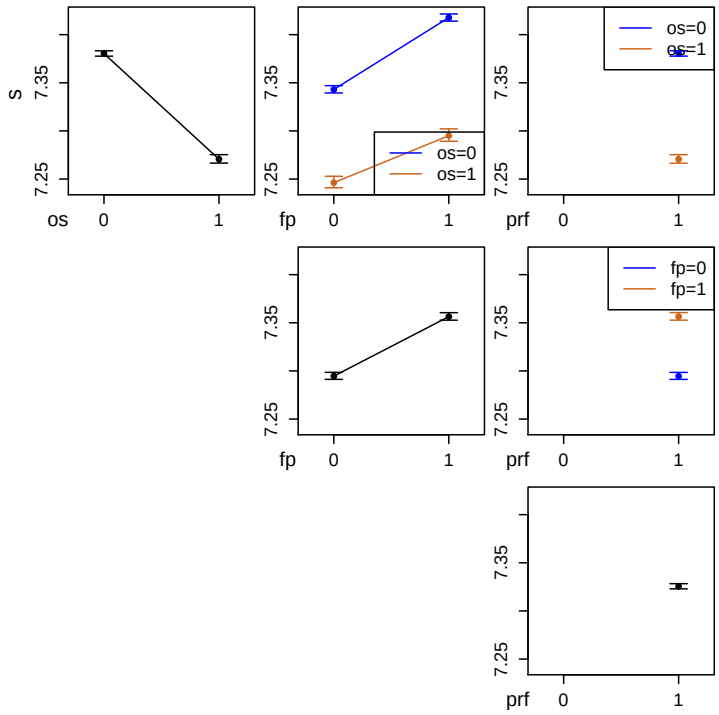

Figure 23: Main effects and interactions for perf.samples.

A.4 Only os=* fp=0 prf=1 and fp=1 prf=2

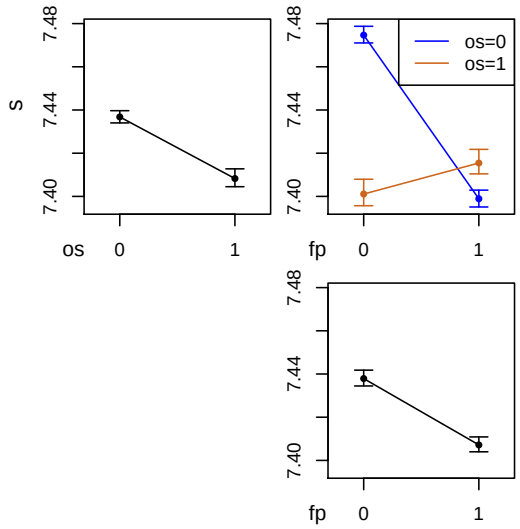

Figure 24: Main effects and interactions for bash.elapsed.

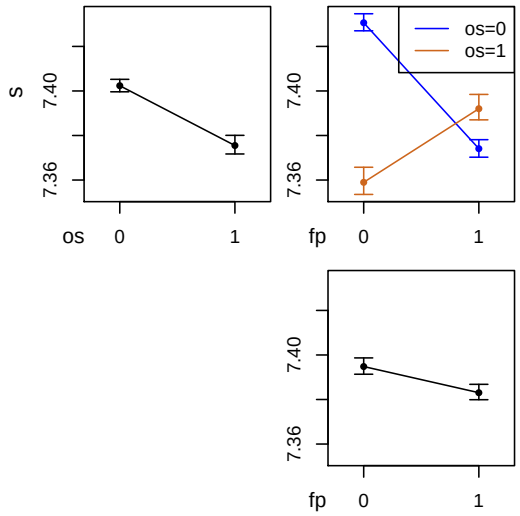

Figure 25: Main effects and interactions for bash.user.

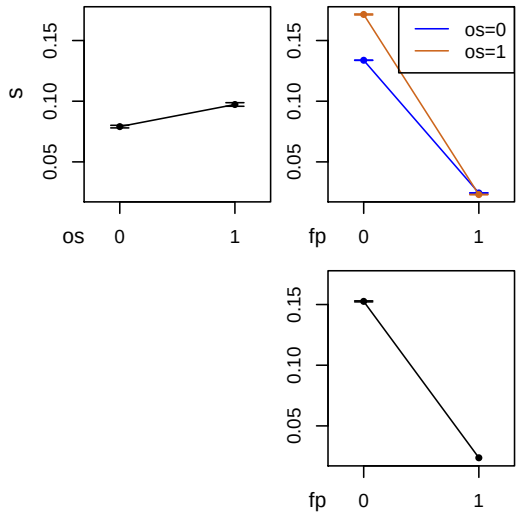

Figure 26: Main effects and interactions for bash.sys.

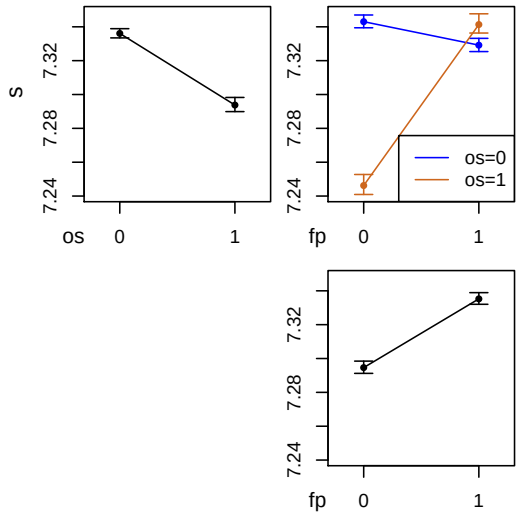

Figure 27: Main effects and interactions for perf.samples.