NIST Technical Note 1860

Defensive code's impact on software performance

David Flater

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.TN.1860

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

NIST Technical Note 1860

Defensive code's impact on software performance

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.1860

January 2015

U.S. Department of Commerce
Penny Pritzker, Secretary

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

National Institute of Standards and Technology Technical Note 1860
Natl. Inst. Stand. Technol. Tech. Note 1860, 31 pages (January 2015)

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.TN.1860

CODEN: NTNOEF

Defensive code's impact on software performance

David Flater

January 2015

Abstract

Defensive code is instructions added to software for the purpose of hardening it against uncontrolled failures and security problems. It is often assumed that defensive code causes a significant reduction in software performance, which justifies its omission from all but the most security-critical applications. We performed an experiment to measure the application-level performance impact of seven defensive code options on two different workloads in four different environments. Of the seven options, only one yielded clear evidence of a significant reduction in performance; the main effects of the other six were either materially or statistically insignificant.

1 Introduction

Defensive code is instructions added to software for the purpose of hardening it against uncontrolled failures and security problems. Typically these instructions test for conditions that "can't happen" (more precisely, could not happen if the software were working as intended) and force the application to terminate if they in fact occur. This behavior is desirable not only to hasten the detection and correction of faults in the software, but also to prevent as-yet uncorrected faults from being exploitable for malicious purposes.

Defensive code can be explicit code added by the software author or it can be safety checks that are retrofit into an application using automated methods to protect "risky" operations. While the latter kind of safety checks are more powerful in mitigating failure modes and vulnerabilities that the software author never contemplated, by the same token one then acquires the burden of proving that they add practical value over the *status quo* without incurring unacceptable costs.

In a previous case study [1] we investigated one kind of automated protection at a micro-benchmark level and learned that defensive code need not always make a big difference to performance. In this work we expand on that result by measuring the application-level performance impact of several examples of both flavors of defensive code on two different workloads in four different environments. We found only one defensive code option that yielded clear evidence of a significant reduction in performance; the main effects of the others were either materially or statistically insignificant.

The remainder of this report is assembled as follows. Section 2 introduces terms used throughout the report. Section 3 describes the environmental conditions of the experiment. Section 4 introduces the software applications that were used as test subjects. Section 5 covers validation of the selected metrics and measures. Section 6 covers the experiment itself. Finally, Section 7 finishes 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 *predictor variables* or *input variables*.

The different values to which a factor is set in an experiment are called *levels* of that factor. Usually, the levels of all 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 to N for factors with more than 2 levels.

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

In the International Vocabulary of Metrology (VIM) [2], a *quantity* is an objective property (*e.g.*, length) that has a magnitude that can be expressed as a numeric value with a reference (*e.g.*, the metre) that defines what the numeric value indicates. In contrast, a software *metric*, as commonly understood, may be quantitative in nature without necessarily expressing the magnitude of an objective property (*e.g.*, a security metric meant to indicate risk), or it may quantify the magnitude of an objective property in units that have no realization as a measurement standard (*e.g.*, size expressed as a count of function points).

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 *measure* 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 VIM, the design of experiments literature, the statistics literature, and the terms' ordinary dictionary definitions.

3 Environments

The experiment was conducted on one version of Linux, two versions of Android, and one version of Windows on four separate devices.

3.1 Linux

The Linux device was a Dell Precision T5400 with dual Xeon X5450 4-core CPUs and 4 GiB of DDR2-667D ECC RAM.

Testing was conducted under Slackware 14.1 64-bit with a replacement kernel, version 3.15.5. The kernel's HZ value was set to 1000, which enables the maximum resolution of CPU time results from bash's `time` command.

To reduce extraneous and irrelevant variability, data collection was performed in single-user mode with SpeedStep disabled in the BIOS configuration as well as the kernel so that the CPUs would operate at a fixed frequency. The CPUs did not support Turbo Boost.

The test programs and kernel were built using the GNU Compiler Collection (GCC) version 4.9.0, GNU Binutils 2.23.52.0.1.20130226, Yasm 1.2.0, and the GNU C library.

3.2 Android 1

The first Android device was a Droid X phone with a single-core, 32-bit Cortex-A8 (OMAP3630) CPU. The operating system had been upgraded to Android 4.4.4 (KitKat, API level 19) via CyanogenMod 11 rev. 20140713 (unofficial) [3]. The kernel identified itself as version 2.6.32.9-AeroKernel but a config file was not provided.

The CPU's frequency was locked at 1 GHz by changing the CPU governor option under Settings, System, Performance, Processor from "interactive" to "performance."

The test programs were built using the Android Native Development Kit (NDK) Revision 9d for x86_64, arm-linux-androideabi-4.8 toolchain, API level 17 (for compatibility with Android 2), and the Bionic C library.

Test programs were run in a shell opened using the Android Debug Bridge (ADB) [4] with the device connected to a PC with a USB cable. Sleep mode and screen savers were disabled.

3.3 Android 2

The second Android device was a Nexus 7 (2012 version) tablet with a quad-core, 32-bit Cortex-A9 (Tegra 3 T30L) CPU. It had been upgraded to Android 4.2.2 (Jelly Bean, API level 17) via AOKP buttered_aokp_tilapia_2013-07-14 [5]. A custom build of kernel version 3.2.47 was used in this experiment. That kernel's HZ value was set to 100.

The slightly older ROM installed on the tablet lacked the option to change the CPU governor in the setup menu, but equivalent tweaks were possible using control files in `/sys/devices/system/cpu/cpu0/cpufreq`. Choosing "performance" raised the frequency to 1.3 GHz which is supposedly restricted to single-core mode [6], so to be safer we chose "userspace" with a frequency of 1.2 GHz instead.

Android 2 used the same compiled binaries as Android 1.

Test programs were run in an ADB shell in the same manner as for Android 1.

3.4 Windows

The Windows device was a Dell Latitude E6330 laptop with a Core i5-3320M 2-core, 4-thread CPU, running Windows 7 Professional Service Pack 1 64-bit with minimal unnecessary software. SpeedStep and Turbo Boost were disabled in the BIOS configuration to reduce the variability of results.

The test programs were built using Visual Studio Express 2013 (VS2013) for Windows Desktop 12.0.21005.13 x64 Cross Tools and Yasm 1.2.0 and deployed with Visual C++ 2013 Redistributable (x64) 12.0.21005.1.

4 Applications

We sought test applications with the following features:

- Can be compiled for all of the targeted platforms;
- Has some CPU-intensive use on all of the targeted platforms;
- Contains explicit defensive code that can be turned on or off;
- Can be run non-interactively for ease of benchmarking.

The two applications that we chose for testing are FFmpeg and GNU Go, described in the following subsections.

4.1 FFmpeg

FFmpeg [7] is a multimedia framework containing both libraries and command-line applications for audio and video processing. We tested version 2.3.

Multi-threaded decoding when the requisite thread libraries are linked and multiple CPU cores are available.

Although many platforms now provide hardware-assisted decoding of video, FFmpeg's software decoders still have plausible use for the many video formats that the hardware acceleration does not support.

FFmpeg is a particularly good example for showing relevance to security as there is the potential for a fault in FFmpeg to be remotely exploitable through a maliciously crafted video file to execute arbitrary code with the privileges of the user attempting to decode the video. There is of course no guarantee that vulnerabilities will not manifest in unchecked places, such as the remotely exploitable integer overflow in the LZO implementation that was corrected in release 2.2.4 [8], but this is no reason to omit safety checks where they already seem warranted.

4.2 GNU Go

GNU Go [9] (hereafter called Gnugo) is simply a program that plays the strategy board game Go. We tested a version pulled from the Git repository on 2014-06-06. We cannot cite a meaningful version number as there had not been a numbered release since early 2009.

Gnugo is a single-threaded program. For testing and benchmarking purposes it has several non-interactive modes of execution.

5 Validation of measures

In three of the four environments a version of the Bash shell [10] was used to obtain measurements. Bash 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 resolution of 0.001 s. On Linux and Android, the resolution of the CPU time metrics is limited by the kernel's compiled-in HZ configurable, which can be as low as 100 Hz.

On Windows, the original plan was to use the Bash shell provided by the Cygwin environment [11]. However, this version failed validation, as did the Bash shell included in MSYS version 1.0.11 [12]. The builtin time commands of these shells registered little or no user CPU time for the test applications that were built using VS2013, presumably due to some disconnect between the Bash shell environment and the Visual C++ runtime environment. As a substitute, we used a stripped-down version of the TimeMem / time-windows utility [13], built with VS2013, to obtain equivalent measurements.

With that substitution, all four environments had an instrument that reported results for elapsed, user, and system time to a numerical resolution of 0.001 s, although the resolution of the measurements proved to be less than that in 3 of the 4 cases.

To validate the measures we reused the *10sec-user étalon* from [14]. *10sec-user* is a single-threaded C program that executes a busy-waiting loop polling the system function `gettimeofday` until the timer readings indicate that at least 10 s have elapsed. As reported in [14], when profiled under Linux, this workload produces results for elapsed and CPU time that have much less variability than is normally seen.

5.1 Face validity

As was done in [14], we checked that the *10sec-user* workload did indeed execute for approximately 10 s of elapsed time using a digital wristwatch and a script that simply printed "Start" and "Stop" before and after the program ran, with an approximately 2 s pause between the start of the script and the start of the timed run. A failure of this primitive validation would indicate that the timekeeping on the platform being tested was significantly off, which *was* a plausible risk considering that unofficial ROMs and kernels were being employed on the two Android devices.

Results are shown in Table 1. Allowing for varying reaction times of the timekeeper and varying latencies for printing the "start" and "stop" messages, there is no evidence in this small sample of an egregious timekeeping anomaly.

Table 1: Elapsed times (s) from manual stopwatch validation of *10sec-user*

Environment	Try 1	Try 2	Try 3
Linux	10.01	10.01	10.03
Android 1	9.99	10.01	10.06
Android 2	9.99	10.01	10.00
Windows	10.07	10.07	10.01

5.2 Linux

In Linux, `gettimeofday` is implemented in user space [15]. Results are shown in Figure 1. The single outlier of 10.012 s for elapsed time occurred on the first iteration, possibly reflecting one or more one-time overheads such as loading the executable binary into the file system cache.

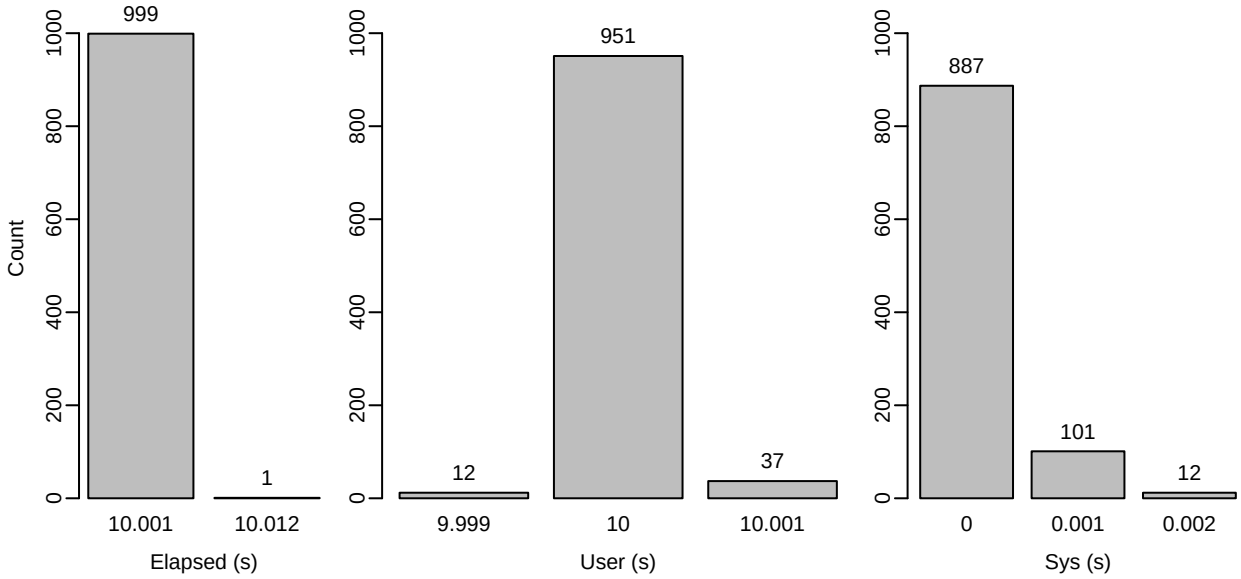

Figure 1: Bar plot of results from 1000 iterations of the *10sec-user* workload in Linux

5.3 Android 1 and 2

Results are shown in Figure 2 and Figure 3. The bins for each histogram were aligned so that there is one bar for each distinct measurement value that was returned.

Not only did the Android environment shift a large part of the CPU time burden of *10sec-user* from user to system, indicating that `gettimeofday` may still be implemented in kernel space on this platform, it also yielded a wider distribution of measurement values, greater variability in the division of CPU time between user space and kernel space, and a lower resolution of CPU time. The resolution of 0.01 s on the tablet was as expected given that the kernel's HZ value was set to 100. On the other hand, the resolution of between 0.0075 s and 0.008 s on the phone corresponds to none of the normally configurable HZ values for the kernel (which are 100 Hz, 250 Hz, 300 Hz, and 1000 Hz).

It is worth noting that an initial configuration of the Android phone that we tried was invalidated by this exercise, with the anomaly being that the user and system CPU time measures added together fell significantly short of 10 s. Investigation using the command `top -m 5 -n 1 -s cpu` under the environmental conditions of the test but with no test workload revealed a continuous load from com.android.inputmethod.latin and system_server accounting for approximately 22 % of the available CPU cycles. Reducing the background loads required both a clean flash of the latest ROM and switching from the terminal app to ADB. For consistency we switched to ADB for Android 2 as well, although it had less impact on the tablet.

Despite those mitigations, the sums of user and system time (Figure 4) still reached as low as 9.71 s, indicating that background loads and overhead still had a greater impact on the test application than they did under Linux.

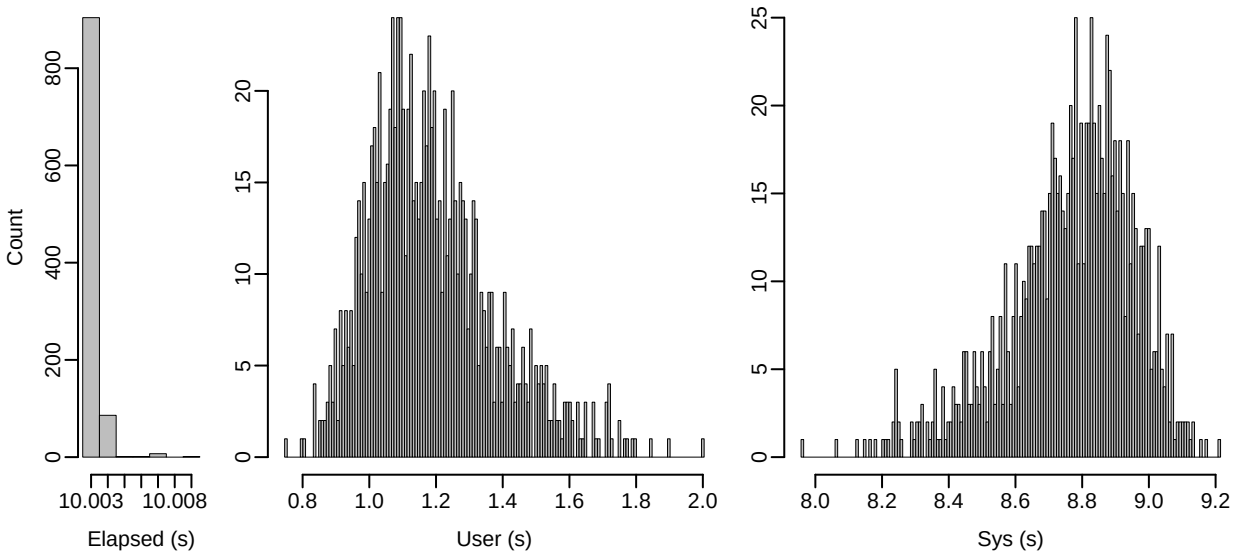

Figure 2: Histogram of results from 1000 iterations of the *10sec-user* workload in Android 1

5.4 Windows

10sec-user would not build using VS2013, so it was built using GCC 4.8.3 in Cygwin instead. Results are shown in Figure 5. Clearly, the resolution of all three measurements is approximately 0.016 s.

6 Experiment

We now proceed to the main experiment. Its goal is to measure and identify patterns in the performance impacts of several types of defensive code on two different applications in four different environments.

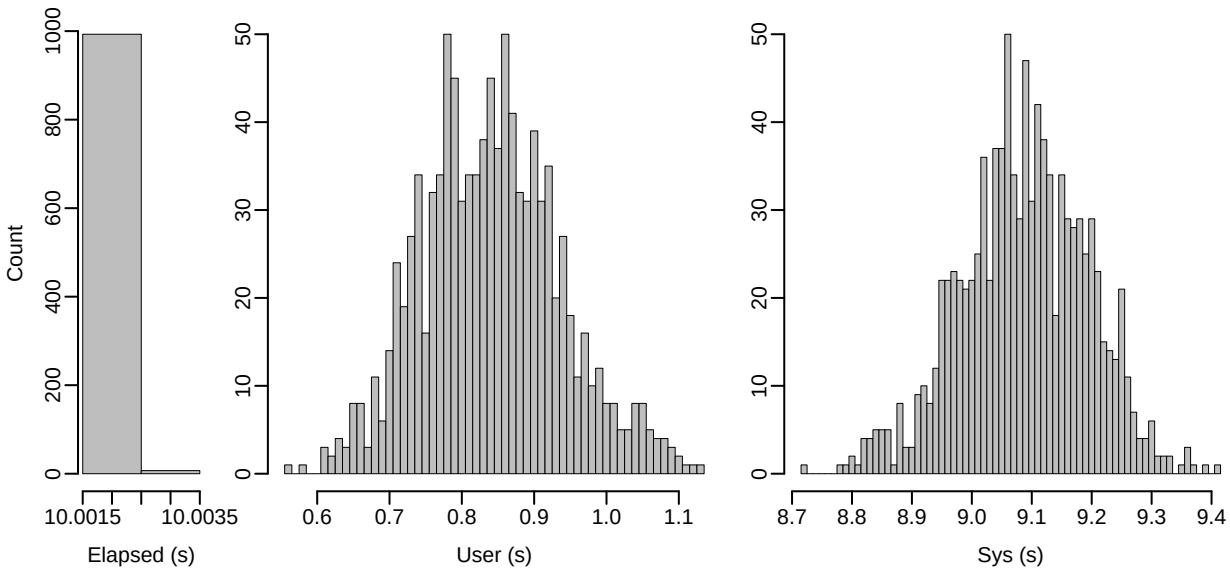

Figure 3: Histogram of results from 1000 iterations of the *10sec-user* workload in Android 2

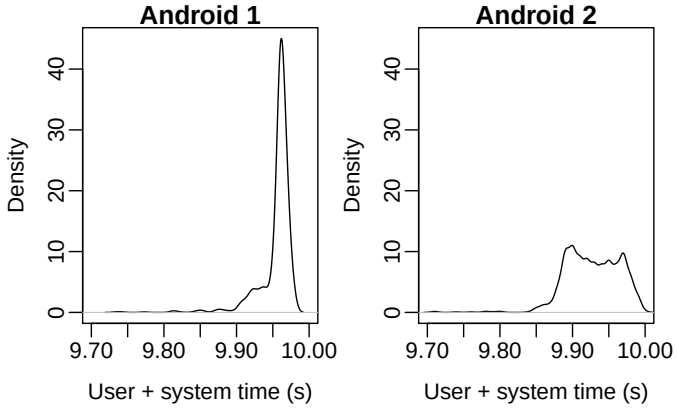

Figure 4: Kernel density plots (bandwidth 0.005 s) of the sums of user and system times

6.1 Workloads

For FFmpeg, we transcoded a test video. The transcoding process reduced the size and frame rate of the video, reduced the sample rate and bit depth of the audio, and changed the video encoding from h264 to mpeg4. The input file was .mov format, 65.17 s in duration, and 178 099 858 bytes in size. It contained an approximately 20 292 kb/s, 1920×1080 pixels, 29.97 Hz, h264 (High) video stream and a 1536 kb/s, 48 kHz, pcm_s16le (2 bytes per sample) stereo audio stream. The resulting output files were less than 5 MB in size yet easily recognizable as transcodings of the input. This was the command line:

```
ffmpeg -y -v quiet -i test.mov -vf scale=320:180 -r 3 -ar 24000 -c:a pcm_u8 out.mov
```

For Gnugo, we created a non-interactive test workload by having it calculate what moves it would have made at each of the 252 board states of the famous "Game of the Century" between Go Seigen and Honinbo Shusai which began on 1933-10-16. The standard output stream of the program was captured for comparison with expected results.

```
gnugo --seed 13131313 --infile GameOfTheCentury.sgf --replay both
```

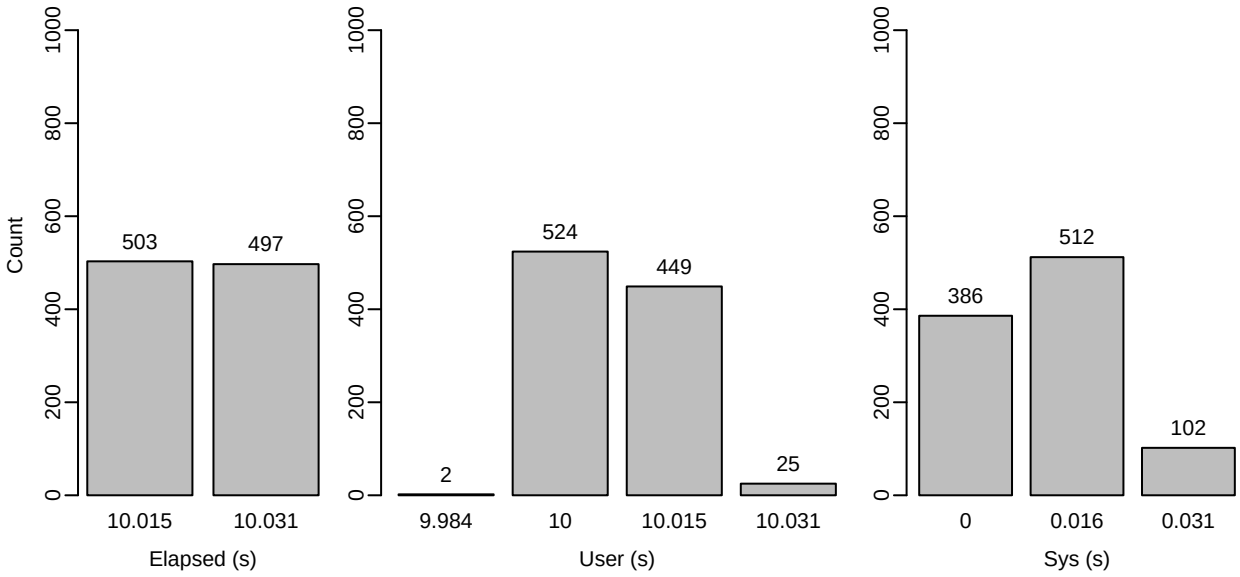

Figure 5: Bar plot of results from 1000 iterations of the *10sec-user* workload in Windows

This non-interactive benchmark makes necessary compromises in order to enable useful data collection. The interactivity of a typical game-playing session has been removed; the focus is entirely on the computation of moves. The use of a transcoding workload with FFmpeg instead of playing a video in real time makes an analogous tradeoff, but transcoding is itself a representative use case.

6.2 Builds

To simplify production we used pre-configured sources and replaced the native build scripts with our own, carrying over those compilation options that did not conflict with independent variables. Configurables relating to independent variables were changed to command line options. We tweaked options to favor execution speed while being cautions about compromises of correctness that were not already made in the stock builds.

For brevity, the following subsections omit long strings of routine options such as preprocessor definitions, include paths, library paths, and linked libraries that were carried over from the original build scripts. Complete details can be found in the Makefiles provided in the raw data distribution available from the Software Performance Project's web page [16].

6.2.1 Linux

Build options used for both applications included `-O3`, `-fno-strict-aliasing`, `--gc-sections`, and `--as-needed`.

The Gnugo build added the `-flto` link-time optimization option. It could not be used with FFmpeg as it caused a link failure.

The FFmpeg build used `-f elf -m amd64` with Yasm and for GCC added `-std=c99`, `-pthread`, `-fno-math-errno`, `-fno-signed-zeros`, and `-fno-tree-vectorize`.

Since the GCC compiler was built natively with `--with-arch=core2`, command-line options to tune the binaries for the specific hardware were unnecessary.

6.2.2 Android

The Android binaries were built to use hardware floating point and linked with libm_hard.a. GCC's auto-vectorization pass doesn't use NEON ("Advanced SIMD") unless `-funsafe-math-optimizations` is specified [17, §3.17.4]; however, FFmpeg includes its own NEON assembly code.

Build options used for both applications included `-march=armv7-a`, `-mtune=cortex-a8`, `-marm`, `-mfloat-abi=hard`, `-mfpu=neon`, `-O3`, `-flto`, `-fno-strict-aliasing`, `-D_NDK_MATH_NO_SOFTFP=1`, `--gc-sections`, `--as-needed`, and `--no-warn-mismatch`. The last was needed to avoid link errors when using hard floating point in this version of the NDK.

The FFmpeg build added `-std=c99`, `-pthread`, `-fno-math-errno`, `-fno-signed-zeros`, and `-fno-tree-vectorize`.

6.2.3 Windows

Build options used for both applications included the compiler options `/O2`, `/Gw`, `/MD`, `/analyze-`, and `/favor:INTEL64` and the linker options `/INCREMENTAL:NO`, `/SUBSYSTEM:CONSOLE`, and `/OPT:REF,ICF`.

The Gnugo build added the `/GL` (with implicit `/ltcg`) global/link-time optimization option, which was used only for Gnugo as it caused a link failure with FFmpeg. This failure could be related to the similar one occurring on Linux.

The FFmpeg build used `-f win64 -m amd64` with Yasm and set the configuration variable `HAVE_STRUCT_POLLFD` for compatibility with current Windows headers (`_WIN32_WINNT` $\geq$ 0x0600).

Some C source files were renamed to eliminate collisions when Visual Studio wrote object files to the working directory, an issue that arose as a result of simplifications made in our build scripts.

6.3 Independent variables

Table 2 shows the factors and levels for the experiment. The *os* and *app* factors are consistent with the descriptions in previous sections. The other factors address defensive code options, some of which are general and others of which are specific to one app or the other.

Table 3 shows how the defensive code factors were reduced to command-line options for the various compilers, assemblers, and linkers used.

A gamut of new debugging options that might qualify as defensive code was introduced in GCC version 4.8 and expanded significantly in GCC 4.9 [17, §3.9]. These options, which have the form `-fsanitize=...` on the command line, were not tested in this experiment.

Additional options to facilitate address space layout randomization and memory protection are commonly used in combination with the defensive code options to harden software against attacks that evade the active checking, but as these additional options do not modify the application logic itself they are not relevant to the experiment.

Several of the defensive code options that were tested support multiple levels of checking thoroughness. To avoid a combinatorial explosion of possible treatments, for each one we tested only the minimum and maximum levels of checking. Assuming monotonicity of performance impact, this suffices to determine the worst-case performance impact of each kind of checking but sacrifices information on the compromises that one might fall back on if the worst case turns out to be unacceptable.

The defensive code options are described in more detail in the following subsections.

Table 2: Independent variables for the experiment

Factor	Description	Level 0	Level 1	Level 2	Level 3
os	Environment	Linux	Android 1	Android 2	Windows
app	Workload	FFmpeg	Gnugo		
stk	GCC stack protector	No	All		
trp	GCC integer overflow traps	No	Yes		
gs	VS buffer security check	No	All		
for	FORTIFY_SOURCE	No	2		
ffa	FFmpeg assert level	0	2		
ffb	FFmpeg safe bitstream reader	Disabled	Enabled		
gga	Gnugo assertions	Disabled	Enabled		

Table 3: Reduction of defensive code factors to compiler command line

Factor	Level 0	Level 1
stk	`-fno-stack-protector`	`-fstack-protector-all`
trp[1]	`-DCONFIG_FTRAPV=0 -fno-trapv`	`-DCONFIG_FTRAPV=1 -ftrapv`
gs[2]	`/GS-`	`/sdl`
for	`-U_FORTIFY_SOURCE`	`-D_FORTIFY_SOURCE=2`
ffa	`-DNDEBUG -DASSERT_LEVEL=0`	`-UNDEBUG -DASSERT_LEVEL=2`
ffb	`-DCONFIG_SAFE_BITSTREAM_READER=0`	`-DCONFIG_SAFE_BITSTREAM_READER=1`
gga	`-DNDEBUG -DGG_TURN_OFF_ASSERTS`	`-UNDEBUG -UGG_TURN_OFF_ASSERTS`

6.3.1 GCC stack protector

The stack protector is a GCC feature that is generally applicable to any compiled C or C++ program and consequently was used for both of the test workloads.

GCC supports the following options (quoted from [17, §3.10]):

`-fstack-protector`
: Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

`-fstack-protector-strong`
: Like `-fstack-protector` but includes additional functions to be protected—those that have local array definitions, or have references to local frame addresses.

`-fstack-protector-all`
: Like `-fstack-protector` except that all functions are protected.

Stack-protector-all is reputed to have an onerous impact on performance [18]. Stack-protector and stack-protector-strong attempt to mitigate the performance impact by protecting only a subset of functions. Together with the default of no-stack-protector, these options would create a four-level factor. To limit the size of the experiment, we chose to test only no-stack-protector and stack-protector-all.

To confirm the efficacy of stack-protector-all we ran the following test program, which yielded a specific abort at run time:

[1] `CONFIG_FTRAPV` applies to FFmpeg only.
[2] For gs = 1, both applications required additional options to disable static checks that blocked compilation.

```
#include <string.h>
int main () {
  char buf[10];
  strcpy (buf, "01234567890123456789012345678 9");
  return 0;
}
```

On Linux:

```
*** stack smashing detected ***: ./stktest1 terminated
```

On Android (from `logcat`):

```
F/libc    ( 6678): stack corruption detected
F/libc    ( 6678): Fatal signal 6 (SIGABRT) at 0x00001a16 (code=-6), thread 6678 (stktest1)
```

6.3.2 GCC –ftrapv

The `-ftrapv` code generation option of GCC "generates traps for signed overflow on addition, subtraction, multiplication operations" [17, §3.18]. Unfortunately, the operability of `-ftrapv` is dubious given an old but still open bug [19] and more recent reports of failure [20]. Empirically, it did not trip on the following test program on either Linux or Android:

```
#include <stdio.h>
#include <limits.h>
int main () {
  int x = INT_MAX;
  printf ("%d\n", x);
  x++;
  printf ("%d\n", x);
  return 0;
}
```

However, GCC documentation for the `-fsanitize=signed-integer-overflow` option, which has functionality similar to `-ftrapv`, asserts that this case is "not an overflow" because of integer promotion rules [17, §3.9].

Regardless, `-ftrapv` did have observable effects on the compilation and execution of programs for the experiment. We therefore retained it as a factor to provide a plausible if imperfect estimate of the performance impact that a fully functional `-ftrapv` option might have.

The most similar option with VS2013 appears to be `/RTCc`, "Reports when a value is assigned to a smaller data type and results in a data loss," but `/RTC` cannot be combined with optimization at all.

6.3.3 Visual C++ stack protector

The Visual Studio analog of GCC's stack protector is the `/GS` ("Buffer Security Check") switch. It is on by default, but without additional steps its behavior is similar to `-fstack-protector` or `-fstack-protector-strong`, not `-fstack-protector-all`.

The `/sdl` ("Enable Additional Security Checks") switch enables a stricter form of `/GS` (similar to `-fstack-protector-all`) and changes some compile-time warnings into errors. To obtain the desired behavior, we used the `/sdl` switch but then turned off any compile-time checks that prevented the applications from building.

For the following test program, an executable built with `/GS-` exited with a segmentation violation while an executable built with `/sdl` (plus `/D_CRT_SECURE_NO_WARNINGS` to disable a static check) exited quietly:

```
#include <stdio.h>
#include <string.h>
int main () {
  char buf[10];
  strcpy (buf, "01234567890123456789012345678 9");
  printf ("Length is %d\n", strlen(buf));
  return 0;
}
```

Rebuilding and running the test program within the IDE with the VS-project equivalent of `/sdl /D_CRT_SECURE_NO_WARNINGS` obtained the following diagnostic:

```
Run-Time Check Failure #2 - Stack around the variable 'buf' was corrupted.
```

Presumably, the run-time checks were working in the command line environment, but there was no visible indication because there was no error handler registered to output diagnostics [21].

Visual Studio 2013 supports more aggressive run-time error checks with the `/RTC` switch, but `/RTC` cannot be combined with optimization.

6.3.4 FORTIFY_SOURCE

FORTIFY_SOURCE is a GNU libc feature that adds safety checking for calls of a set of standard libc functions that are known to be vulnerable to buffer overflows. The following is quoted from the feature_test_macros man page [22]:

> **_FORTIFY_SOURCE** (since glibc 2.3.4)
> Defining this macro causes some lightweight checks to be performed to detect some buffer overflow errors when employing various string and memory manipulation functions. Not all buffer overflows are detected, just some common cases. In the current implementation checks are added for calls to **memcpy**(3), **mempcpy**(3), **memmove**(3), **memset**(3), **stpcpy**(3), **strcpy**(3), **strncpy**(3), **strcat**(3), **strncat**(3), **sprintf**(3), **snprintf**(3), **vsprintf**(3), **vsnprintf**(3), and **gets**(3). If **_FORTIFY_SOURCE** is set to 1, with compiler optimization level 1 (*gcc –O1*) and above, checks that shouldn't change the behavior of conforming programs are performed. With **_FORTIFY_SOURCE** set to 2 some more checking is added, but some conforming programs might fail. Some of the checks can be performed at compile time, and result in compiler warnings; other checks take place at run time, and result in a run-time error if the check fails. Use of this macro requires compiler support, available with **gcc**(1) since version 4.0.

On Linux, with FORTIFY_SOURCE level 1 or 2, the following test program:

```
#include <string.h>
int main () {
  char buf[10];
  strcpy (buf, "01234567890123456789");
  return 0;
}
```

Yielded a warning at compile time and a specific abort at run time:

```
*** buffer overflow detected ***: ./fortest1 terminated
```

However, in the Android environment specified in Section 3.2 and Section 6.2.2, neither the warning nor the abort would reproduce. Although support for FORTIFY_SOURCE level 1 reportedly was introduced in Android 4.2 [23], the binary produced with `-D_FORTIFY_SOURCE=1` on the compiler command line was exactly the same size as that produced with `-U_FORTIFY_SOURCE`. Moreover, the string "FORTIFY" occurs in none of the header files included with NDK r9d or even r10.

Finding no evidence that FORTIFY_SOURCE was doing anything at all in our Android NDK builds, we tested this feature only on Linux.

6.3.5 FFmpeg-specific options

FFmpeg's configure script supports the following options (quoted from configure --help):

```
  --disable-safe-bitstream-reader
                           disable buffer boundary checking in bitreaders
                           (faster, but may crash)
  --assert-level=level     0(default), 1 or 2, amount of assertion testing,
                           2 causes a slowdown at runtime.
  --enable-ftrapv          Trap arithmetic overflows
```

These options affect definitions of the preprocessor variables `CONFIG_SAFE_BITSTREAM_READER`, `ASSERT_LEVEL`, and `CONFIG_FTRAPV` that are placed in the generated files config.h and config.asm.

Although three source files contain code that is conditional on `CONFIG_FTRAPV`, the primary mechanism of `--enable-ftrapv` is the addition of `-ftrapv` to CFLAGS. We therefore considered this combination equivalent to the use of `-ftrapv` with Gnugo.

`--enable-safe-bitstream-reader` does not enable the safe bitstream reader in all contexts. In modules where that checking was believed to be redundant, including some modules that are relevant to our test case, the authors inserted `#define UNCHECKED_BITSTREAM_READER 1` statements to override the global configure setting. Commentary in libavcodec/get_bits.h explains:

```
/*
 * Safe bitstream reading:
 * optionally, the get_bits API can check to ensure that we
 * don't read past input buffer boundaries. This is protected
 * with CONFIG_SAFE_BITSTREAM_READER at the global level, and
 * then below that with UNCHECKED_BITSTREAM_READER at the per-
 * decoder level. This means that decoders that check internally
 * can "#define UNCHECKED_BITSTREAM_READER 1" to disable
 * overread checks.
 * Boundary checking causes a minor performance penalty so for
 * applications that won't want/need this, it can be disabled
 * globally using "#define CONFIG_SAFE_BITSTREAM_READER 0".
 */
```

Similarly, `--assert-level=0` does not disable *all* assertions. There are many assertions that use the standard C macro directly, and those are not affected by `--assert-level`. Furthermore, a subset of the C assertions are protected by statements like the following (from libavcodec/ratecontrol.c):

```
#undef NDEBUG // Always check asserts, the speed effect is far too small to disable them.
```

Although global deletion of `#define UNCHECKED_BITSTREAM_READER 1` and `#undef NDEBUG` statements would be a straightforward patch, we refrained from overriding the authors' intent. Instead, we simply used `-DNDEBUG -DASSERT_LEVEL=0` for ffa level 0 and `-UNDEBUG -DASSERT_LEVEL=2` for ffa level 1.

6.3.6 Gnugo-specific options

According to the Gnugo documentation, Gnugo developers "are strongly encouraged to pepper their code with assertions to ensure that data structures are as they expect" [24, §4.6.3]. The Gnugo engine contains many such assertions, including defensive code to ensure that references to board locations are actually *on the board* by validating indices. They are enabled by default and may be disabled by defining the variable `GG_TURN_OFF_ASSERTS` during compilation. Other Gnugo components use standard C language assertions, which are similarly disabled by defining the variable `NDEBUG`. For the *gga* factor we bundled these two together, defining both variables or not to disable or enable all assertions.

6.4 Executables

The sizes of the built executables for Linux, Android, and Windows are compared in Figure 6, Figure 7, and Figure 8 respectively. On Linux, evidently `-ftrapv` caused the most significant expansion, followed by the stack protector. On Android, FFmpeg shows the same clear pattern as on Linux. For Gnugo the pattern is too weak to separate the groups. On Windows, the patterns are different for the two applications and the changes in size are all much smaller than for Linux or Android.

6.5 Experimental design

Table 4: Applicability of defensive code factors

	Linux (os=0)		Android (os=1,2)		Windows (os=3)	
app	FFmpeg (app=0)	Gnugo (app=1)	FFmpeg (app=0)	Gnugo (app=1)	FFmpeg (app=0)	Gnugo (app=1)
stk	0, 1	0, 1	0, 1	0, 1	X	X
trp	0, 1	0, 1	0, 1	0, 1	X	X
gs	X	X	X	X	0, 1	0, 1
for	0, 1	0, 1	X	X	X	X
ffa	0, 1	X	0, 1	X	0, 1	X
ffb	0, 1	X	0, 1	X	0, 1	X
gga	X	0, 1	X	0, 1	X	0, 1
# treat	32	16	16	8	8	4

Table 4 shows the applicability of the defensive code factors by *os* and *app*. An X indicates that a factor is not available in the given context (particular combinations of *os* and *app*).

We collected as many samples as possible for every applicable treatment without attempting to further reduce the number of treatments through a fractional design.

As shown, the experimental design is not orthogonal. However, orthogonality can be achieved by subsetting the data by combinations of *os* and *app*, essentially breaking the experiment into 8 smaller experiments that have full factorial designs.[3] With the main effects of *os* and *app* being irrelevant, the only negatives of this approach are that we do not get singular summary results for the defensive code factors and analyzing interactions between those factors and *os*/*app* is more cumbersome.

6.6 Ordering

Data collection was broken into multiple sessions on each device to provide availability for other tasks, to enable inspection of partial results, and to account for any "boot dependency" (the possibility that performance could have a component that varies randomly from one reboot of the device to the next but remains constant for the period of uptime between reboots).

Within each session, each "iteration" consisted of a single run of each of the test binaries, executed in a random order with all Gnugo and all FFmpeg binaries shuffled together. 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. A session would terminate only at the end of an iteration so that the same number of samples would be collected for each treatment.

[3] In principle, the two versions of Android could be combined in an experiment with *os* as a factor.

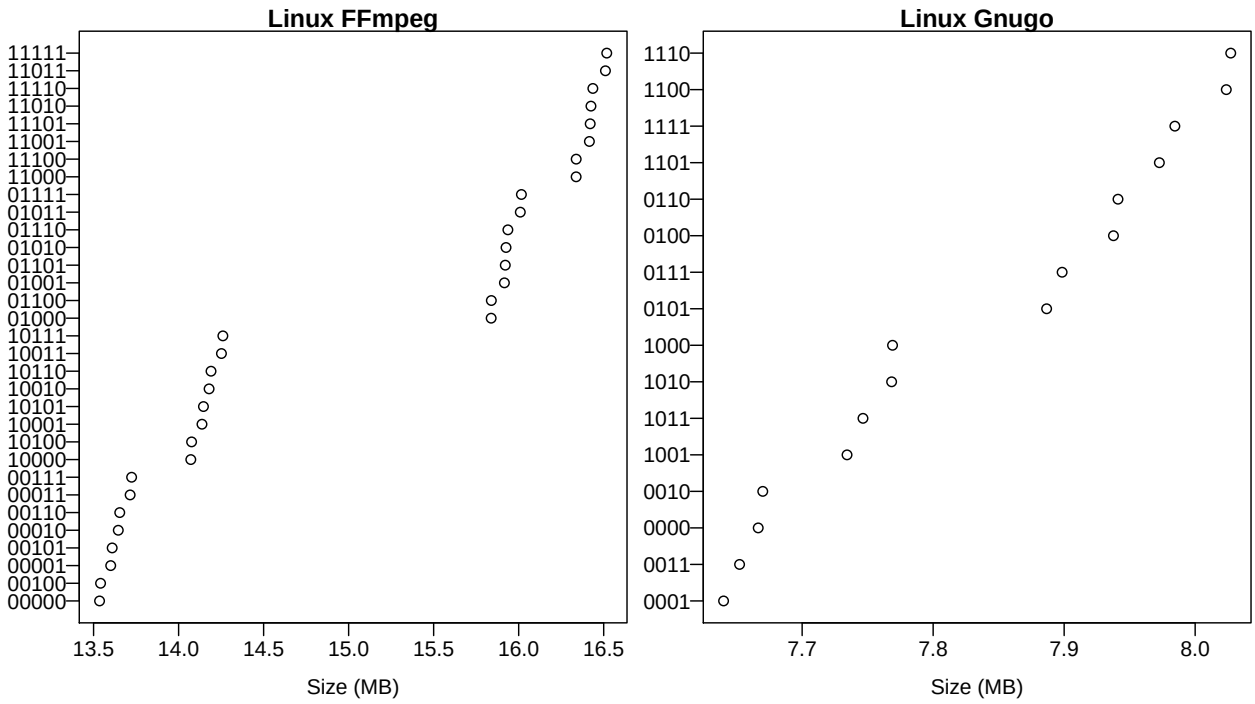

Figure 6: Sorted data plots of executable binaries for Linux, identified by treatment. FFmpeg treatment encoding: stk-trp-for-ffa-ffb; Gnugo treatment encoding: stk-trp-for-gga.

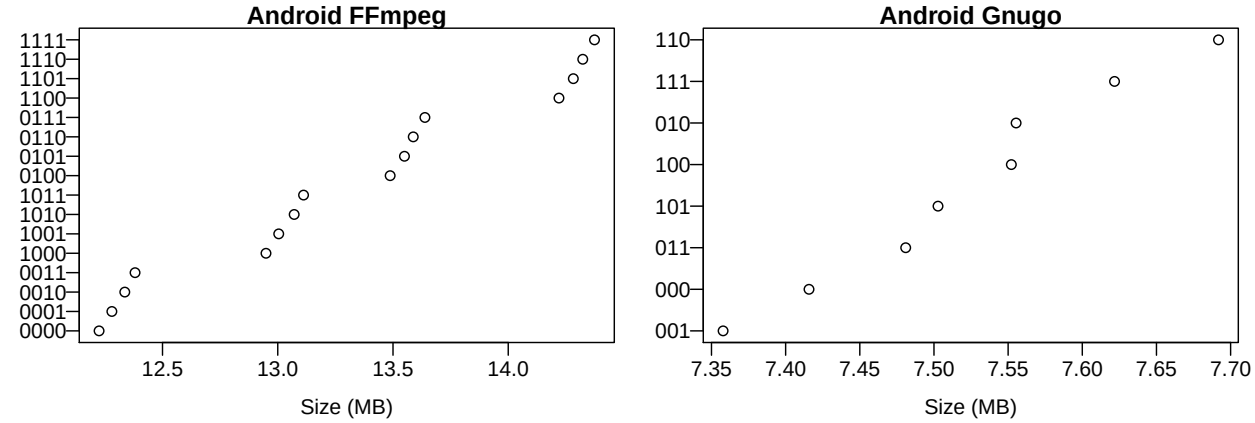

Figure 7: Sorted data plots of executable binaries for Android, identified by treatment. FFmpeg treatment encoding: stk-trp-ffa-ffb; Gnugo treatment encoding: stk-trp-gga.

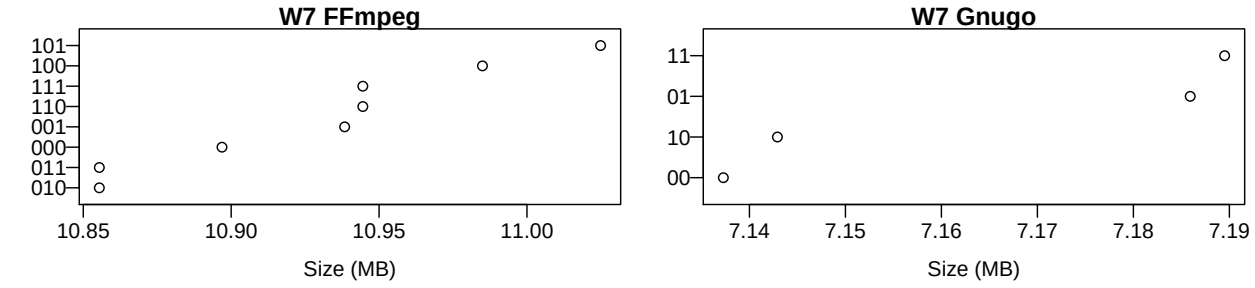

Figure 8: Sorted data plots of executable binaries for Windows, identified by treatment. FFmpeg treatment encoding: gs-ffa-ffb; Gnugo treatment encoding: gs-gga.

Table 5: Size of FFmpeg workload output by levels of *os*

os	Size of out.mov
0	4 876 207 B
1	4 874 685 B
2	4 875 251 B
3	4 875 071 B

Table 6: Counts of sessions and numbers of iterations per treatment or session

		Iterations per treatment		session	
os	Sessions	min	max	min	max
0	5	106	106	9	69
1	4	65	65	14	19
2	4	78	79	14	24
3	4	371	371	61	127

6.7 Validation of workload outputs

Each execution of an experimental workload was followed by a comparison of its output with an expected output. A mismatch would have resulted in an abort of the session with retention of the discrepant data for further investigation.

Although FFmpeg outputs were repeatable and consistent among treatments within each *os* environment, none of the four environments exactly reproduced the output of any other, as shown in Table 5. Nevertheless, all four outputs were apparently valid transcodings of the input, based on inspection.

For Gnugo, the only discrepancy of results was the addition of carriage return characters at line ends in the Windows environment.

6.8 Results

6.8.1 Sample sizes

See Table 6. An infrastructural fault on the tablet caused 3 of the first 4 sessions to stop prematurely in the middle of an iteration, resulting in slightly different numbers of samples among the treatments. The next 3 attempts at collecting more data on the tablet led to a spontaneous shutdown after only 1 or 2 complete iterations. Since the standard deviation cannot be calculated for a sample of size one, the few data resulting from the last 3 sessions were set aside. Among the retained data, the most significant imbalance within a session was 14 versus 15 samples, and after combining sessions the worst imbalance was 78 versus 79 samples. These imbalances were deemed to have an insignificant effect on the orthogonality of the experiments as analyzed.

6.8.2 Distributions

In general, most of the distributions were less skewed than those seen in previous experiments of this project, with the caveat that the samples here were smaller and would be less likely to capture low-frequency events.

On Linux, each session began with a high outlier in elapsed time for every treatment of FFmpeg. The pattern was much weaker for Gnugo and did not appear on Windows or Android.

Evidence of drift and possibly a small boot-dependency effect was apparent in some plots from Windows where the range of results was very narrow. Nevertheless, it appeared only under such magnification (stretching of the Y axis) that there is no potential for a materially significant impact.

6.8.3 Main effects and interactions

On three of the four devices, system time remained always negligible. The CPU time accounting on Android 1 was discrepant, apparently attributing CPU time to kernel space that on other devices was attributed to user space. This discrepancy resulted in confounding of the effects and interactions plots. To resolve the confounding, we summed the user and system time measures to yield a single CPU time measure.

Table 7 shows the relative magnitudes of the main effects (level 1 versus level 0) using pooled data (*i.e.*, combining the data from multiple sessions). 95 % confidence intervals were determined using the ordinary method, applying the t-distribution based on an assumption of approximate normality of the observed times.[4] Effective degrees of freedom for the uncertainties were computed using the Welch-Satterthwaite formula [27, §G.4.1].

Table 8 restates the relative magnitudes using the mean-of-means method to reduce the data to summary statistics instead of pooling all of the data across sessions. This is one way to mitigate a boot-dependency effect. With the number of reboots having been constrained by practicality, it makes the confidence intervals much wider,[5] and more of the marginal effects lose statistical significance. However, the estimates do not change very much given the essentially-balanced nature of the data.

Figure 9 through Figure 16 in Appendix A provide a complete set of elapsed time plots for main effects and 2-factor interactions for pooled data subset by combinations of *os* and *app*. The corresponding plots for CPU time and mean-of-means, which proved to be redundant, are not provided here but can be found in the raw data distribution available from the Software Performance Project's web page [16].

Each figure is organized as the cross product of a set of factors with itself. Main effects appear along the diagonal. For interactions, the columns determine which factor is used for the X axis while the rows determine which factor is used for separate lines.

In the main effects plots, what is important is the differences in the value of the dependent variable, which result in a sloping line. In the interaction plots, what is important is the differences in slope or lack of parallelism when comparing multiple lines. A different slope means that the magnitude of the effect of one factor differed depending on the level of another factor.

The vertical scales of all plots within a figure are the same and are set to cover the range of the biggest effect. Thus, it is valid to compare the plots within a figure to each other to gauge the relative magnitudes of effects, but similar relative comparisons between figures are not valid. Note in particular that Figure 15 has a magnified vertical axis because the effects were all very small, and the presence of sloping lines in that figure does not indicate the presence of large effects.

The numbers inside of the plots indicate the count of samples represented by each mean. No corrections were attempted for the slight imbalances in the data for Android 2 ($os = 2$).

6.9 Analysis

6.9.1 Main effects

Assuming that the estimates are accurate, we have a significant slowdown attributable to *trp*, marginal slowdowns attributable to *stk* and *gga*, and negligible slowdowns for all other factors. The *trp* slowdown varies by *os* and *app*, but has a relative magnitude on the order of 50 % and remains statistically significant in every context even with the more conservative mean-of-means analysis.

[4] Although in previous experiments we found the bias-corrected and accelerated (BC_a) bootstrap method [25, 26] to perform better on non-normally-distributed data, the small samples in this experiment—especially with mean-of-means—are a contraindication for use of BC_a.

[5] Ref. [28] describes an analogous problem and illustrates methods for incorporating assumptions to narrow the confidence intervals.

Table 7: Relative magnitudes of main effects (level 1 versus level 0) computed using pooled data

Factor	os	app	Elapsed time effect (%)	ν_{eff}	CPU time effect (%)	ν_{eff}
stk	0	0	0.02 ± 0.92	3389.87	0.58 ± 0.83	3389.86
	0	1	2.39 ± 2.49	1689.98	2.39 ± 2.49	1689.98
	1	0	6.86 ± 1.56	1029.40	6.88 ± 1.57	1029.40
	1	1	5.67 ± 3.95	513.35	5.67 ± 3.95	513.35
	2	0	2.14 ± 0.57	1207.52	6.90 ± 1.59	1254.94
	2	1	5.22 ± 3.94	624.01	5.21 ± 3.93	624.03
trp	0	0	31.40 ± 0.10	3156.14	27.85 ± 0.06	3364.10
	0	1	68.19 ± 0.33	1368.76	68.19 ± 0.33	1368.75
	1	0	27.11 ± 0.55	966.73	27.25 ± 0.55	966.68
	1	1	54.79 ± 1.04	395.12	54.82 ± 1.04	394.99
	2	0	9.39 ± 0.31	1078.17	30.48 ± 0.57	1256.76
	2	1	62.04 ± 0.96	510.67	62.01 ± 0.96	510.95
gs	3	0	0.37 ± 0.05	2883.79	0.73 ± 0.07	2935.45
	3	1	0.64 ± 0.20	1475.81	0.64 ± 0.20	1475.77
for	0	0	0.11 ± 0.92	3389.88	−0.09 ± 0.82	3388.27
	0	1	−0.36 ± 2.42	1693.59	−0.36 ± 2.42	1693.59
ffa	0	0	0.69 ± 0.92	3389.87	0.59 ± 0.83	3388.43
	1	0	0.80 ± 1.52	1032.54	0.81 ± 1.53	1032.74
	2	0	0.16 ± 0.58	1253.80	0.52 ± 1.54	1256.98
	3	0	1.11 ± 0.04	2957.47	1.47 ± 0.05	2924.13
ffb	0	0	0.06 ± 0.92	3389.93	−0.29 ± 0.82	3389.14
	1	0	0.07 ± 1.51	1037.60	0.07 ± 1.52	1037.62
	2	0	−0.04 ± 0.57	1256.89	−0.31 ± 1.52	1256.94
	3	0	−0.39 ± 0.05	2824.76	−0.29 ± 0.07	2893.95
gga	0	1	2.61 ± 2.49	1693.98	2.61 ± 2.49	1693.98
	1	1	4.13 ± 3.90	517.24	4.13 ± 3.90	517.24
	2	1	4.49 ± 3.91	624.56	4.48 ± 3.91	624.55
	3	1	4.05 ± 0.03	1302.60	4.05 ± 0.03	1300.52

6.9.2 Interactions

The plots in Appendix A reveal only minor 2-way interactions. However, it is evident from Table 7 that *trp* caused a significantly worse slowdown for Gnugo ($app = 1$) than it did for FFmpeg ($app = 0$).

The $os \times app$ interactions are not captured by the summary data presented, but they would be unsurprising: when the demand for CPU cycles increases because of a change in *app*, devices with slower and fewer CPU cores are impacted more.

6.9.3 Root cause of *trp* performance hit

Level 1 of *trp* corresponds to use of the `-ftrapv` compiler switch. A 2005 article on compiler checks described the implementation of `-ftrapv` as follows:

"In practice, this means that the GCC compiler generates calls to existing library functions rather than generating assembler instructions to perform these arithmetic operations on signed integers." [29]

Disassembly of unoptimized binaries for the test program shown in Section 6.3.2 confirmed that `-ftrapv` had the effect of replacing a single `addl` instruction with a call to a function that performed addition, checked for overflow, and aborted the program if an overflow had occurred. The overflow checking involved two conditional branches. Apparently, the costs of this code expansion were not completely erased by high levels of optimization, but some combinations of *os* and *app* suffered much more than others.

Table 8: Relative magnitudes of main effects (level 1 versus level 0) computed using mean-of-means

Factor	os	app	Elapsed time effect (%)	ν_{eff}	CPU time effect (%)	ν_{eff}
stk	0	0	−0.01 ± 4.27	158.00	0.58 ± 3.87	157.99
	0	1	2.39 ± 11.76	77.81	2.39 ± 11.76	77.81
	1	0	6.86 ± 6.51	61.49	6.88 ± 6.54	61.49
	1	1	5.68 ± 17.07	29.73	5.68 ± 17.08	29.73
	2	0	2.14 ± 2.37	58.79	6.86 ± 7.19	61.88
	2	1	5.37 ± 18.71	29.90	5.37 ± 18.70	29.90
trp	0	0	31.39 ± 0.26	118.77	27.85 ± 0.26	156.20
	0	1	68.18 ± 1.56	63.00	68.19 ± 1.56	63.00
	1	0	27.10 ± 2.26	57.62	27.24 ± 2.27	57.59
	1	1	54.80 ± 4.51	22.81	54.82 ± 4.51	22.80
	2	0	9.37 ± 0.77	41.15	30.45 ± 2.29	60.61
	2	1	62.06 ± 4.60	24.38	62.02 ± 4.60	24.39
gs	3	0	0.37 ± 0.48	28.86	0.72 ± 0.57	29.13
	3	1	0.64 ± 2.29	13.94	0.64 ± 2.29	13.94
for	0	0	0.10 ± 4.27	157.99	−0.09 ± 3.84	157.92
	0	1	−0.37 ± 11.45	77.98	−0.37 ± 11.45	77.98
ffa	0	0	0.69 ± 4.30	158.00	0.58 ± 3.87	157.93
	1	0	0.81 ± 6.35	61.67	0.82 ± 6.38	61.69
	2	0	0.16 ± 2.39	61.79	0.51 ± 6.96	62.00
	3	0	1.11 ± 0.29	29.18	1.47 ± 0.32	27.08
ffb	0	0	0.01 ± 4.27	158.00	−0.29 ± 3.84	157.96
	1	0	0.07 ± 6.31	61.97	0.07 ± 6.33	61.98
	2	0	−0.10 ± 2.38	62.00	−0.37 ± 6.90	62.00
	3	0	−0.39 ± 0.48	27.42	−0.29 ± 0.62	28.61
gga	0	1	2.60 ± 11.78	78.00	2.60 ± 11.78	78.00
	1	1	4.13 ± 16.85	29.96	4.13 ± 16.86	29.96
	2	1	4.33 ± 18.56	29.94	4.33 ± 18.55	29.94
	3	1	4.05 ± 0.40	12.17	4.05 ± 0.39	12.15

7 Conclusion

We performed an experiment to measure and identify patterns in the performance impacts of several types of defensive code on two different applications in four different environments. For the most part, the performance differences were small in magnitude and materially if not statistically insignificant. The exception was GCC's `-ftrapv` option, which resulted in over 50 % slowdown for one app.

Additional plots and raw data from the experiment are available from the Software Performance Project's web page [16].

Acknowledgments

Thanks to William F. Guthrie and Jim Filliben of the Statistical Engineering Division for consultation. Thanks to Vadim Okun and other reviewers for helpful reviews.

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

[3] CyanogenMod, 2014. http://www.cyanogenmod.org/.

[4] Android Debug Bridge, 2014. http://developer.android.com/tools/help/adb.html.

[5] Android Open Kang Project (AOKP), 2013. http://aokp.co/.

[6] Nexus 7 (2012 version), 2014. http://en.wikipedia.org/wiki/Nexus_7_(2012_version).

[7] FFmpeg, 2014. https://www.ffmpeg.org/.

[8] Don A. Bailey. Raising Lazarus - The 20 Year Old Bug that Went to Mars, June 2014. http://blog.securitymouse.com/2014/06/raising-lazarus-20-year-old-bug-that.html.

[9] GNU Go, 2014. http://www.gnu.org/software/gnugo/.

[10] GNU Bash, 2014. https://www.gnu.org/software/bash/.

[11] Cygwin, 2014. https://www.cygwin.com/.

[12] MSYS, 2014. http://www.mingw.org/wiki/msys.

[13] Time-windows, Windows port of Unix time utility, 2014. https://code.google.com/p/time-windows/.

[14] David Flater. Screening for factors affecting application performance in profiling measurements. NIST Technical Note 1855, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, October 2014. http://dx.doi.org/10.6028/NIST.TN.1855.

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

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

[17] GCC Manual, version 4.9.0, April 2014. https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/.

[18] Han Shen. [PATCH] Add a new option "–fstack-protector-strong". *GCC-patches mailing list*, June 2012. https://gcc.gnu.org/ml/gcc-patches/2012-06/msg00974.html.

[19] Steven Bosscher. GCC Bug 35412: Correctness with –ftrapv dependended on libcall notes, May 2009. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=35412.

[20] Stack Overflow. How to make gcc ftrapv work?, December 2013. https://stackoverflow.com/questions/20851061/how-to-make-gcc-ftrapv-work.

[21] Nikola Dudar. Response to '_set_security_error_handler no longer declared'. *Visual C++ General forum*. http://www.windows-tech.info/17/088c02904eb201ba.php.

[22] Feature_test_macros man page. *Linux Programmer's Manual,* §7, August 2012.

[23] Security enhancements in Android 4.2, 2012. https://source.android.com/devices/tech/security/enhancements42.html.

[24] GNU Go Program Documentation, Edition 3.8, June 2014. Info document included with GNU Go software distribution.

[25] 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.

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

[27] Joint Committee for Guides in Metrology. *Evaluation of measurement data—Guide to the expression of uncertainty in measurement*. JCGM 100:2008, http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf.

[28] Mark S. Levenson, David L. Banks, Keith R. Eberhardt, Lisa M. Gill, William F. Guthrie, Hung-kung Liu, Mark G. Vangel, James H. Yen, and Nien-fan Zhang. An approach to combining results from multiple methods motivated by the ISO GUM. *NIST Journal of Research*, 105(4):571–579, July 2000. http://dx.doi.org/10.6028/jres.105.047.

[29] Robert C. Seacord. Compiler checks. *Build Security In*, May 2013. https://buildsecurityin.us-cert.gov/articles/knowledge/coding-practices/compiler-checks.

A Main effect and interaction plots

Note: Confidence intervals (95 %) are not drawn when the range is vanishingly small.

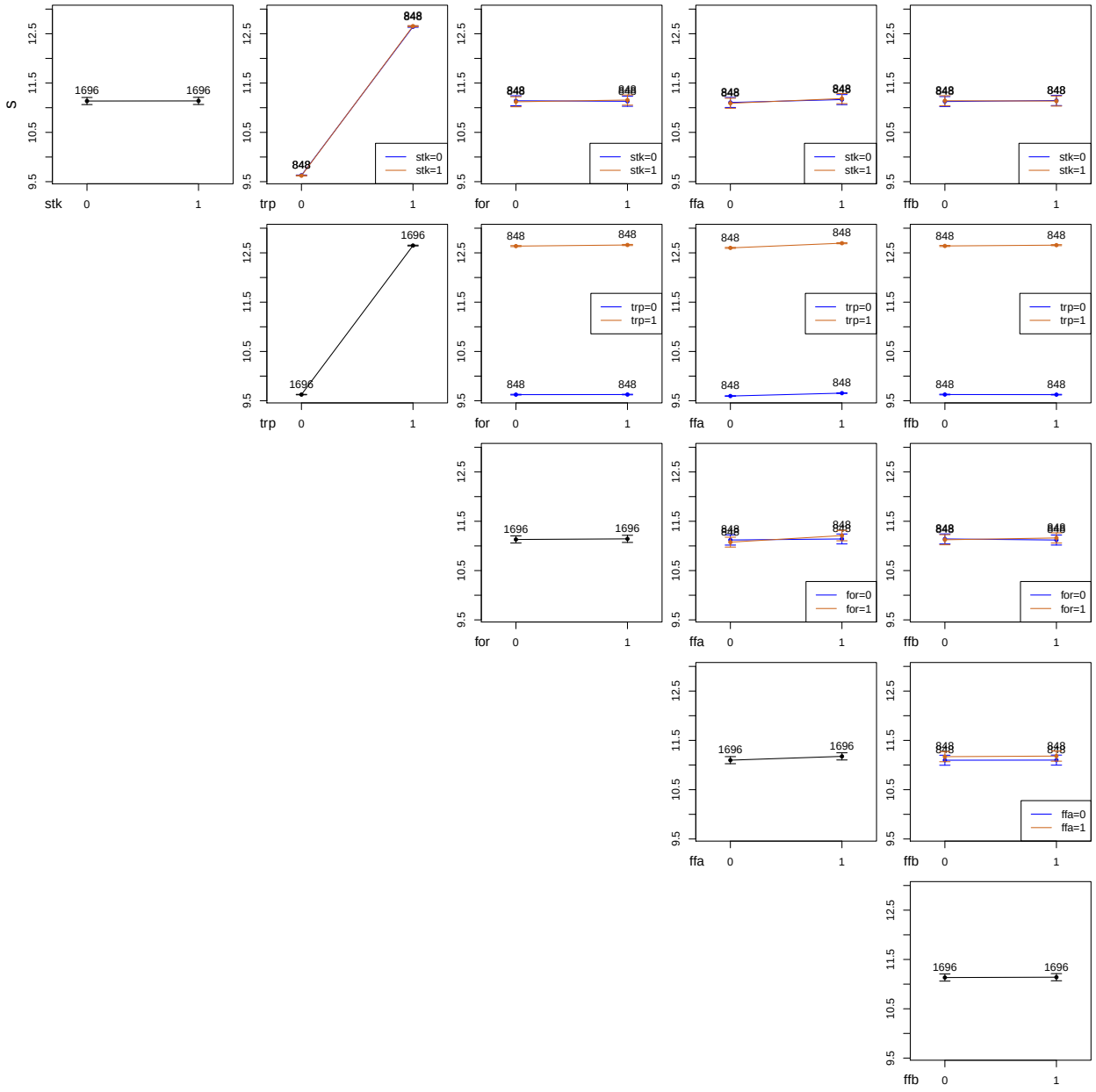

Figure 9: Plots for elapsed time in $os = 0$, $app = 0$ using pooled data

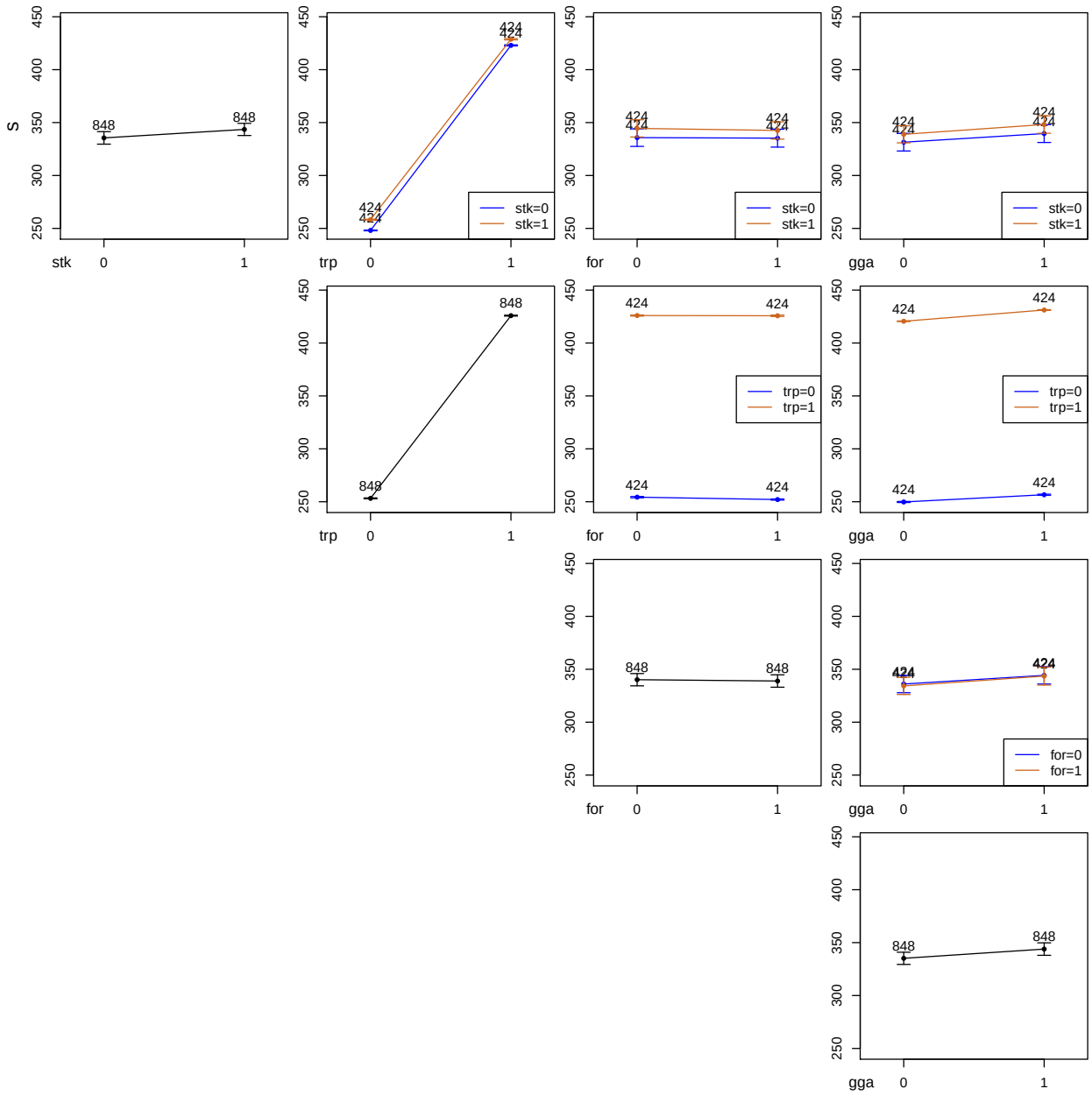

Figure 10: Plots for elapsed time in $os = 0$, $app = 1$ using pooled data

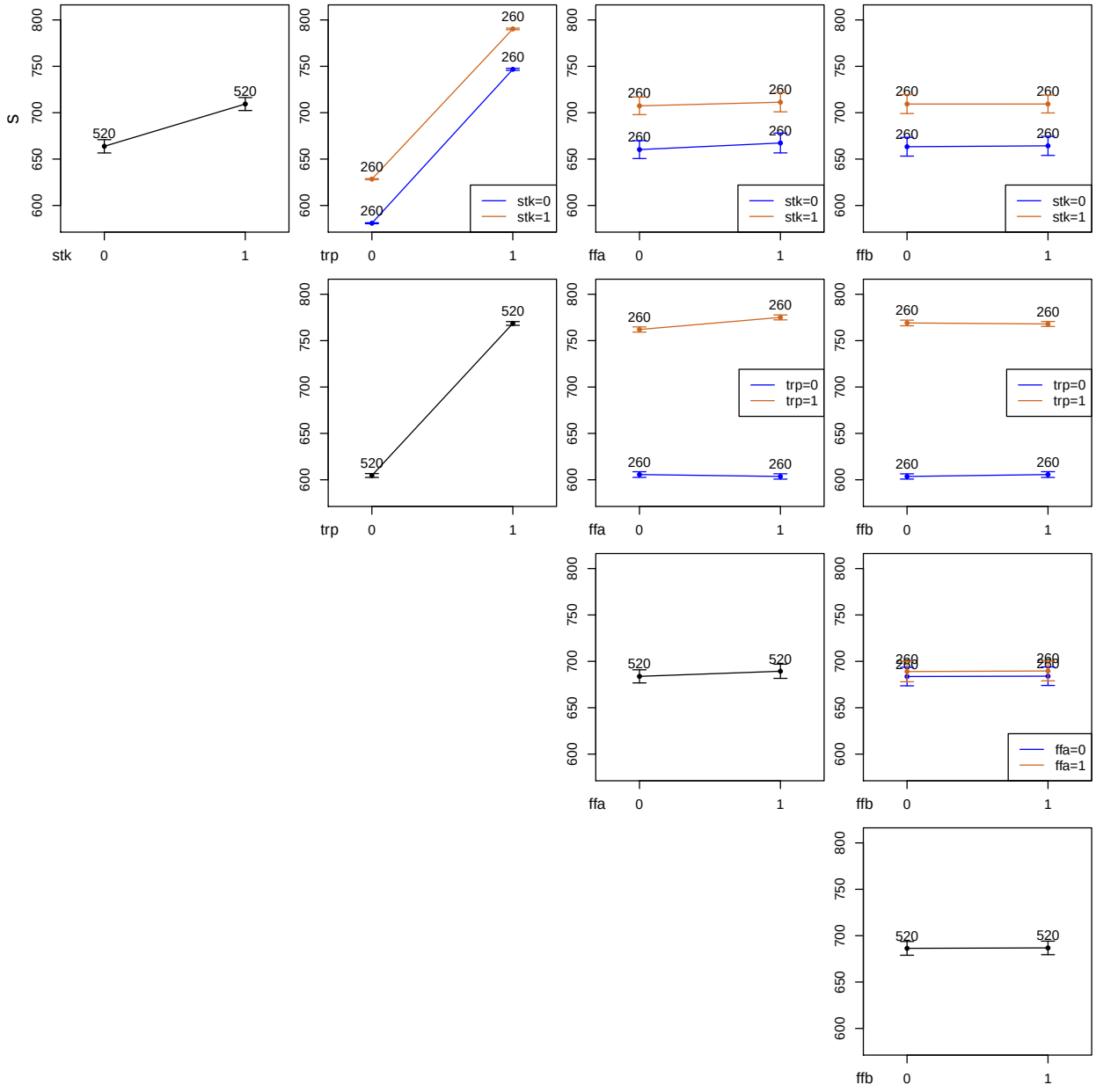

Figure 11: Plots for elapsed time in $os = 1$, $app = 0$ using pooled data

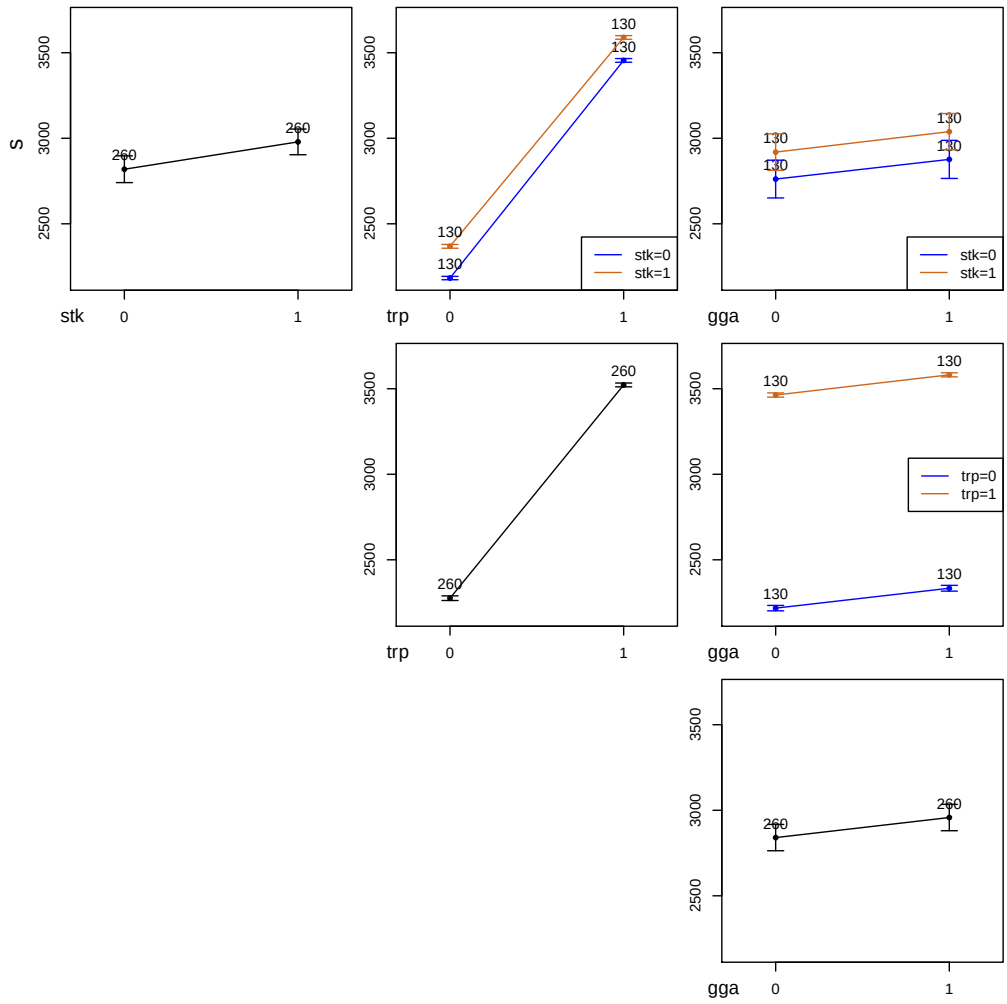

Figure 12: Plots for elapsed time in $os = 1$, $app = 1$ using pooled data

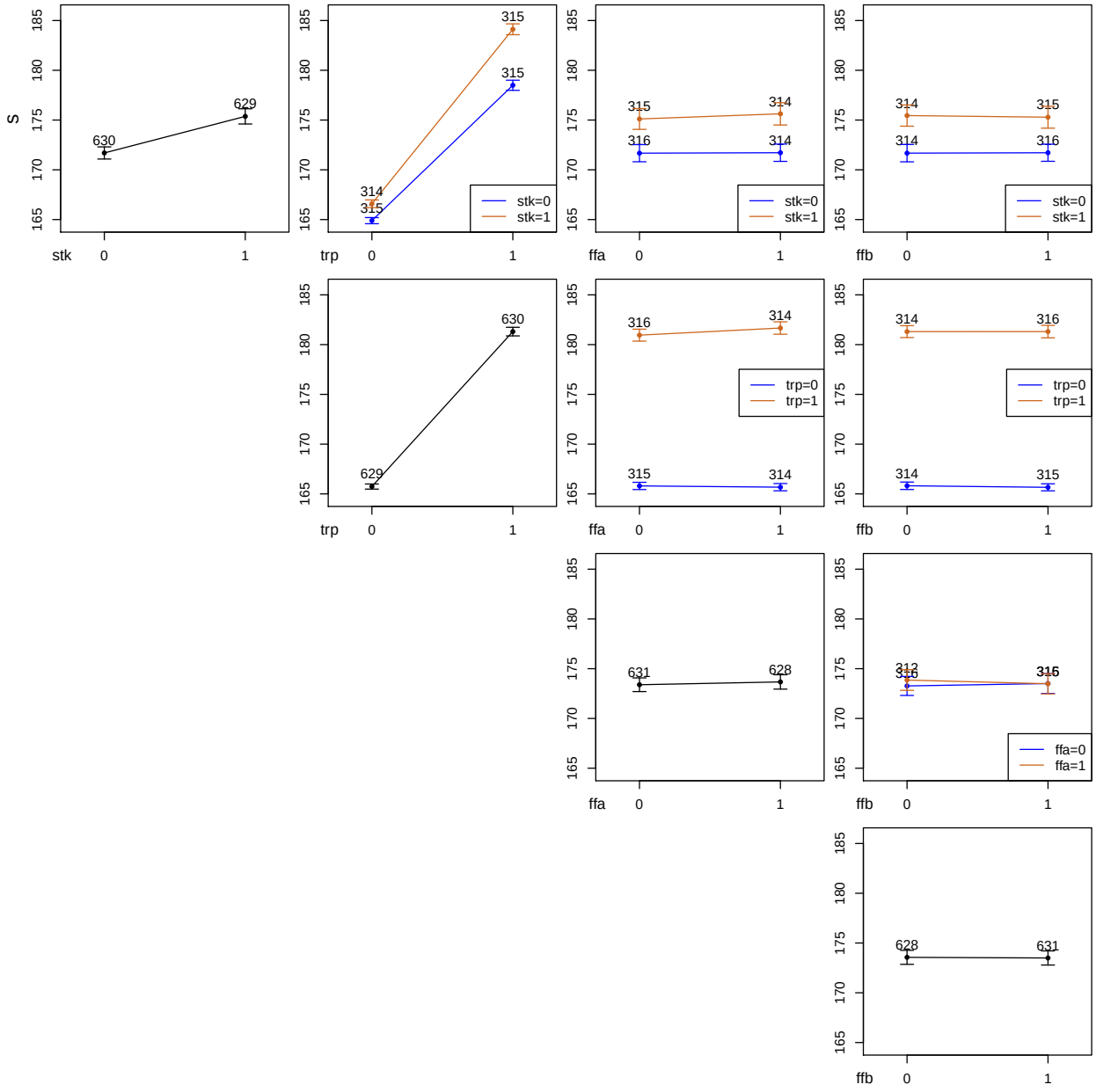

Figure 13: Plots for elapsed time in $os = 2$, $app = 0$ using pooled data

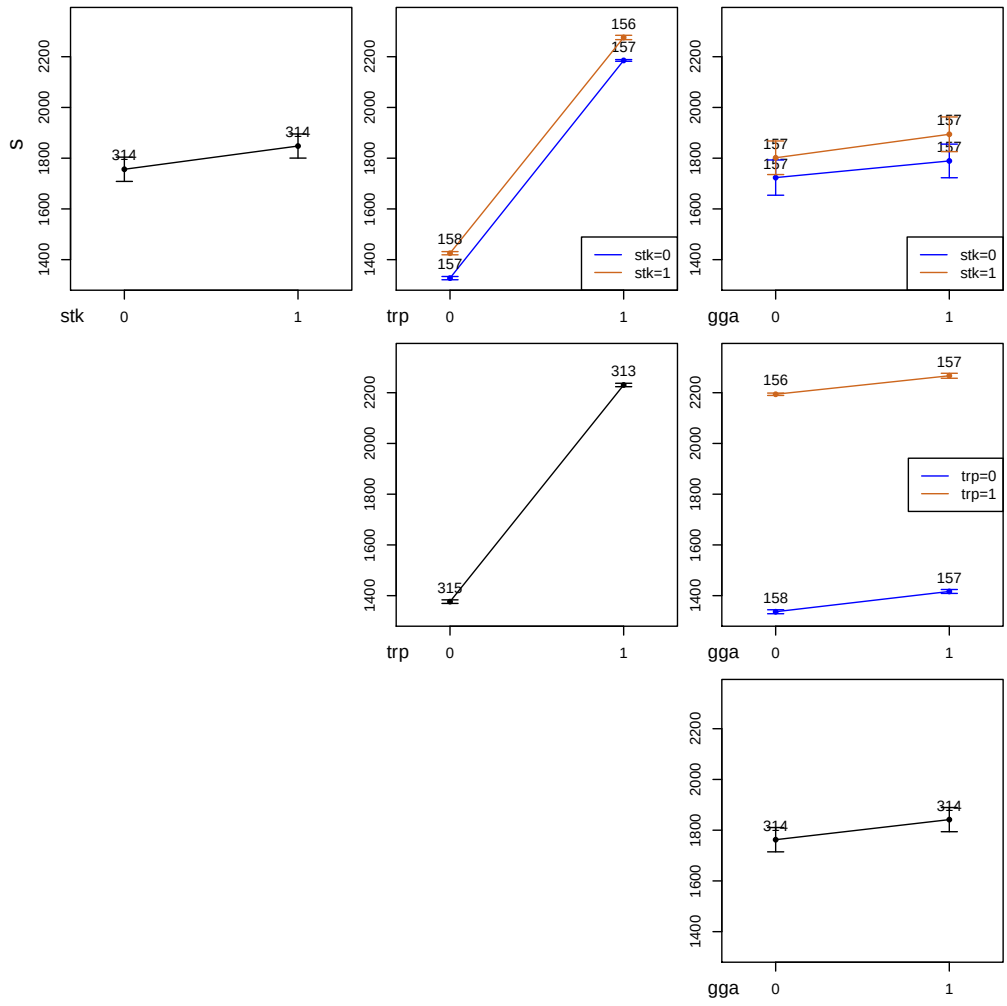

Figure 14: Plots for elapsed time in $os = 2$, $app = 1$ using pooled data

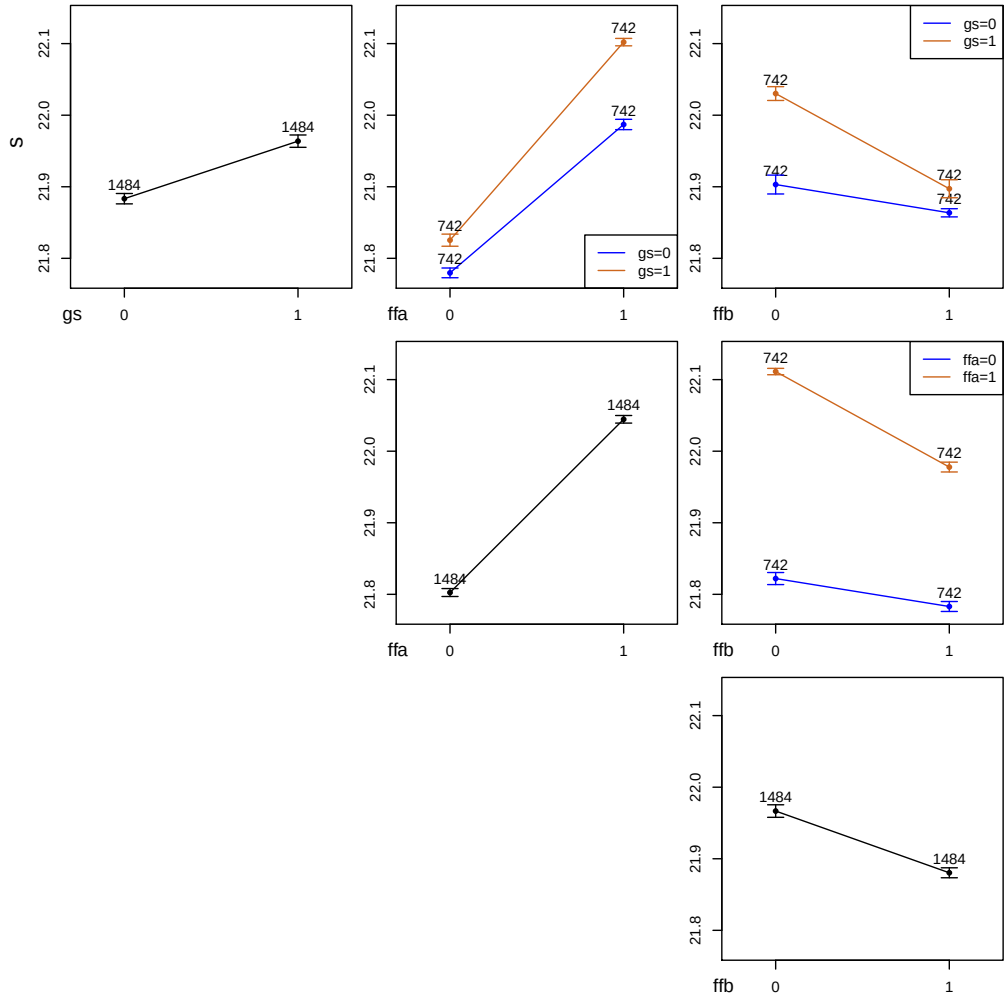

Figure 15: Plots for elapsed time in $os = 3$, $app = 0$ using pooled data

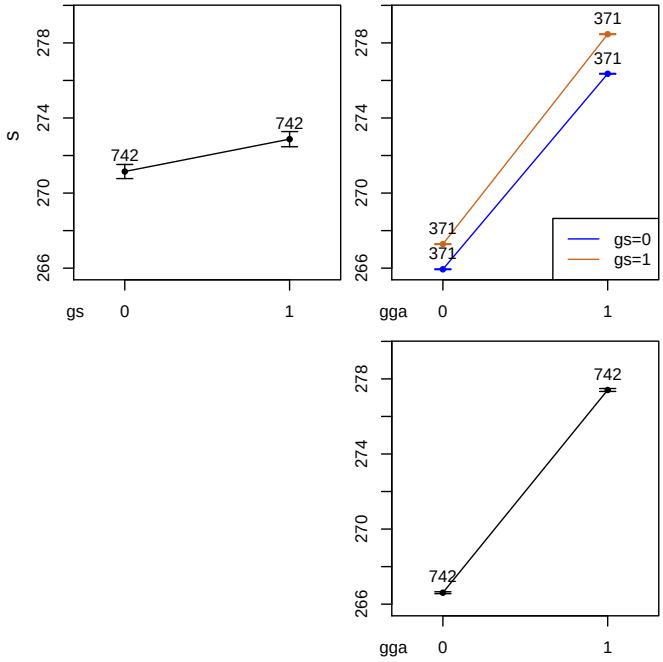

Figure 16: Plots for elapsed time in $os = 3$, $app = 1$ using pooled data