L41: Lab 1 - Getting Started with Kernel Tracing - I/O

Dr Robert N.M. Watson

Michaelmas Term 2015

The goals of this lab are to:

- Introduce you to our experimental environment and DTrace
- Have you explore user-kernel interactions via system calls and traps
- Gain experience tracing I/O behaviour in UNIX
- Build intuitions about the probe effect

You will do this by using DTrace to analyse the behaviour of a potted, kernel-intensive block-I/O benchmark.

Background: POSIX I/O system calls

POSIX defines a number of synchronous I/O APIs, including the `read()` and `write()` system calls, which accept a file descriptor to a file or other storage object, a pointer to a buffer to read to or write from, and a buffer length as arguments. Although they are synchronous calls, some underlying aspects of I/O are frequently deferred; for example, `write()` will store changes in the buffer cache, but they may not make it to disk for some time unless a call is made to `fsync()` to force a writeback.

You may wish to read the FreeBSD `read(2)`, `write(2)`, and `fsync(2)` system-call manual pages to learn more about the calls before proceeding with this lab. You can do this using the `man` command; you may need to specify the *manual section* to ensure that you get API documentation rather than program documentation; for example, `man 2 write` for the system call to ensure that you don't get the `write(1)` program man page.

The benchmark

Our I/O benchmark is straightforward: it performs a series of `read()` or `write()` I/O system calls in a loop using configurable buffer and total I/O sizes. Optionally, an `fsync()` system call be issued at the end of the I/O loop to ensure that buffered data is written to disk. The benchmark samples the time before and after the I/O loop, optionally displaying an average bandwidth. The lab bundle will build two versions of the benchmark: `io-static` and `io-dynamic`. The former is statically linked (i.e., without use of the run-time linker or dynamic libraries), whereas the latter is dynamically linked.

Compiling the benchmark

You will need to copy the lab bundle to your BeagleBone Black, untar it, and build it before you can begin work. Once you have configured the BBB so that you can log in (see *L41: Lab Setup*), you can use SSH to copy the bundle to the BBB:

```
scp /anfs/www/html/teaching/1516/L41/labs/io.tgz guest@192.168.141.100:/data
```

Logged in as `guest`, untar and build the bundle:

```
cd /data
tar -xzf io.tgz
cd io
make
```

Running the benchmark

Once built, you can run the benchmark binaries as follows, with command-line arguments specifying various benchmark parameters:

```
./io-static
```

or:

```
./io-dynamic
```

The benchmark must be run in one of three operational modes: *create*, *read*, or *write*, specified by flags. In addition, the target file must be specified. If you run the `io-static` or `io-dynamic` benchmark without arguments, a small usage statement will be printed, which will also identify the default buffer and total I/O sizes configured for the benchmark.

In your experiments, you will need to be careful to hold most variables constant in order to isolate the effects of specific variables; for example, you may wish to hold the total I/O size constant as you vary the buffer size. You may wish to experiment initially using `/dev/zero` – the kernel's special device node providing an unlimited source of zeros, but will also want to run the benchmark against a file in the filesystem.

Required operation flags

Specify the mode in which the benchmark should operate:

-c Create the specified file using the default (or requested) total I/O size.

-r Benchmark `read()` of the target file, which must already have been created.

-w Benchmark `write()` of the target file, which must already have been created.

Optional I/O flags

-b *buffersize* Specify an alternative buffer size in bytes. The total I/O size must be a multiple of buffer size.

-t *totalsize* Specify an alternative total I/O size in bytes. The total I/O size must be a multiple of buffer size.

-B *Bare mode* disables the normal preparatory work done by the benchmark, such as flushing outstanding disk writes and sleeping for one second. This mode is preferred when performing whole-program benchmarking.

-d *Direct I/O mode* disables use of the buffer cache by the benchmark by specifying the `O_DIRECT` flag to the `open()` system call on the target file. When switching from buffered mode, the first measurement using `-d` should be discarded, as some cached data may still be used.

-s *Synchronous mode* causes the benchmark to call `fsync()` after writing the file to cause all buffered writes to complete before the benchmark terminates – and in particular before the final timestamp is taken.

Terminal output flags

The following arguments control terminal output from the benchmark; remember that output can substantially change the performance of the system under test, and so you should ensure that output is either entirely suppressed during tracing and benchmarking, or that tracing and benchmarking only occurs during a period of program execution unaffected by terminal I/O:

-q *Quiet mode* suppress all terminal output from the benchmark, which is preferred when performing whole-program benchmarking.

-v *Verbose mode* causes the benchmark to print additional information, such as the time measurement, buffer size, and total I/O size.

Example benchmark commands

This command creates a default-sized data file in the `/data` filesystem:

```
./io-static -c /data/iofile
```

This command runs a simple `read()` benchmark on the data file, printing additional information about the benchmark run:

```
./io-static -v -r /data/iofile
```

This command runs a simple *write()* benchmark on the data file, printing additional information about the benchmark run:

```
./io-static -v -w /data/iofile
```

If performing whole-program analysis using DTrace, be sure to suppress output and run the benchmark in bare mode:

```
./io-static -B -q -r /data/iofile
```

The following command disables use of the buffer cache when running a read benchmark; be sure to discard the output of the first run of this command:

```
./io-static -d -r /data/iofile
```

To better understand kernel behaviour, you may also wish to run the benchmark against `/dev/zero`, a pseudo-device that returns all zeroes, and discards all writes:

```
./io-static -r /dev/zero
```

To get a high-level summary execution time, including a breakdown of total wall-clock time, time in userspace, and 'system time', use the UNIX `time` command:

```
/usr/bin/time -p ./io-static -r -B -d -q /data/iofile
real 1.31
user 0.00
sys 0.31
```

However, it may be desirable to use DTrace to collect more granular timestamps by instrumenting return from `execve()` and entry of `exit()` for the program under test.

This run of `io-static`, reading the data file `/data/iofile`, bypassing the buffer cache, and running in bare mode (i.e., without quiescing prior to the benchmark) took 1.31 seconds of wall-clock time to complete. Of this, (roughly) 0.00 seconds were spent in userspace, and (roughly) 0.31 seconds were spent in the kernel on behalf of the process. From this output, it is unclear where the remaining 1.00 seconds were spent, but presumably a substantial fraction was spent blocked on (slow) SD Card I/O. Time may also have been spent running other processes – and lower-precision time measurement, such as provided by `time`, may suffer from non-trivial rounding error.

Objects on which to run the benchmark

You may wish to point the benchmark at one of two objects:

`/dev/zero` The zero device: an infinite source of zeroes, and also an infinite sink for any data you write to it.

`/data/iofile` A writable file in a journalled UFS filesystem on the SD card. You will need to create the file using `-c` before performing `read()` or `write()` benchmarks.

Notes on using DTrace

You will want to use DTrace in two ways during this lab:

- To analyse just the I/O loop itself; or
- To analyse whole-program execution including setup, run-time linking, etc.

In the former case, it is useful to know that the system call `clock_gettime` is both run immediately before, and immediately after, the I/O loop. You may wish to bracket tracing between a return probe for the former, and an entry probe for the latter. For example, you might wish to include the following in your DTrace scripts:

```
syscall::clock_gettime:return
/execname == "io-static" && !self->in_benchmark/
{
    self->in_benchmark = 1;
}

syscall::clock_gettime:entry
/execname == "io-static" && self->in_benchmark/
{
    self->in_benchmark = 0;
}

END
{
    /* You might print summary statistics here. */
    exit(0);
}
```

Other DTrace predicates can then refer to `self->in_benchmark` to determine whether the probe is occuring during the I/O loop. The benchmark is careful to set up the runtime environment suitably before performing its first clock read, and to perform terminal output only after the second clock read, so it is fine to leave benchmark terminal output enabled.

In the latter case, you will wish to configure the benchmark for whole-program analysis by disabling all output (`-q`) and enabling bare mode (`-B`). In this case, you will simply want to match the executable name for system calls or traps using `/execname == "io-static"/` (or `io-dynamic` as required).

Notes on using `ministat`

While more sophisticated statistics and graphing packages such as **R** may be appropriate for preparing your lab report, they can be rather heavyweight during casual performance investigation. You may wish to use the command-line `ministat` tool to analyse simple distributions and perform statistical tests. `ministat` accepts file arguments consisting of a list of numeric values, one per line, and generates text output to the terminal. Different data sets passed via multiple file arguments will be compared using Student's *t*-test.

Notes on benchmark

This benchmark calculates average I/O rate for a particular run. Be sure to run the benchmark more than once (ideally, perhaps a dozen times), and discard the first output which may otherwise be affected by prior benchmark runs (e.g., if data is left in the buffer cache and the benchmark is not yet in the steady state). Do be sure that terminal I/O from the benchmark is not included in tracing or time measurements.

Exploratory questions

These questions are intended to help you understand the behaviour of the I/O benchmark, and may provide supporting evidence for your experimental questions. However, they are just suggestions – feel free to approach the problem differently!

1. Baseline benchmark performance analysis:
    - How do `read()` and `write()` performance compare?
    - Describe the performance effect of using the buffer cache? Be sure to consider the effects of both `-d` and `-s`.
    - What proportion of benchmark time is spent in userspace vs. kernel?
    - How many times are different system calls called during the I/O loop?
    - How much time is spent, cumulatively, in each system call by type during the I/O loop?
    - What is the role of traps in execution of the I/O loop?
    - How does work performed in just the I/O loop compare with whole-program behaviour?
2. Probe effect and measurement decisions
    - How does performance change if you insert system-call or trap probes in the I/O loop?
    - What sources of variance may be affecting benchmark performance, and how can we measure them?

Experimental questions

Your lab report will compare several configurations of the benchmark, exploring (and explaining) performance differences between them. Do ensure that your experimental setup suitably quiesces other activity on the system, and also use a suitable number of benchmark runs; you may wish to consult the *FreeBSD Benchmarking Advice* wiki page linked to from the module's reading list for other thoughts on configuring the benchmark setup. The following questions are with respect to the benchmark reading a file through the buffer cache:

- Holding the total I/O size constant (16MB), how does varying I/O buffer size affect IO-loop performance?
- Holding the buffer size constant (16K) and varying total I/O size, how does static vs. dynamic linking affect whole-program performance?
- At what file-size threshold does the performance difference between static and dynamic linking fall below 5%? At what file-size threshold does the performance difference fall below 1%?

For the purposes of performance graphs, measured performance (the dependent variable, or Y axis) is with respect to I/O bandwidth rather than literal execution time. This will allow us to analyse relative efficiency as file and buffer sizes vary.