DDN® STORAGE

DIO-pro Manual

DDN I/O Profiling Group

CONFIDENTIAL

April 29, 2016

Contents

Chapter 1

Introduction

1.1 Objective

This document is a reference manual for DIO-pro, DDN's I/O profiling tool. The document outlines the use of `DIO-pro` and its bundled applications. It also provides some technical background and best-use practices.

Disclaimer: this document is currently a work-in-progress.

1.2 Maintenance

Currently, Karel De Vogeleer, *performance analysis and modeling specialist* at DDN Storage, is the main developer of DIO-pro. If you have questions, comments, ideas or complaints about DIO-pro, Karel can be contacted on *kdevogeleer@ddn.com*.

1.3 Brief Introduction to DIO-pro

DIO-pro is an I/O profiling tool designed to capture I/O events emanating from an arbitrary application. DIO-pro observes I/O events from the point of view of the application, it is unaware of events in the file system or storage units.

DIO-pro writes I/O traces per process to a log file in a binary format. The profiling code of DIO-pro is located in libraries, called `dio-pro*.so`, that are loaded with the application under investigation. The programs `dio-pro-xml` and `dio-pro-stat` are bundled with DIO-pro. `dio-pro-xml` converts a binary log file to a human-readable format, by default formatted in `xml`. `dio-pro-stat` calculates performance statistics based on the human readable log file.

In the following example DIO-pro is contained in a shared library called `dio-pro.so`. The shared library is loaded dynamically with the application under investigation, in this case IOR, via `LD_PRELOAD`:

```
$ LD_PRELOAD=dio-pro.so IOR
```

Each time IOR calls a function that DIO-pro profiles, IOR's function call is redirected to DIO-pro, which DIO-pro then forwards to the original function call. Figure 1.1 shows the abstract time line of an arbitrary function call by IOR that is profiled by DIO-pro.

After the application the finished, a log file can be found in DIO-pro's default log directory: `/tmp/dio-pro`. For `MPI`-enabled applications, the DIO-pro log file is compressed with zlib. Log files for POSIX applications are not compressed. DIO-pro suffixes compressed log file names with the extension `.gz`. The compressed files can easily be decompressed using the tools `gunzip` and `zcat`.

`dio-pro-xml` converts the decompressed binary log file into human readable format. `dio-pro-xml` accepts the log file via a pipe, the output can then be redirected to a file:

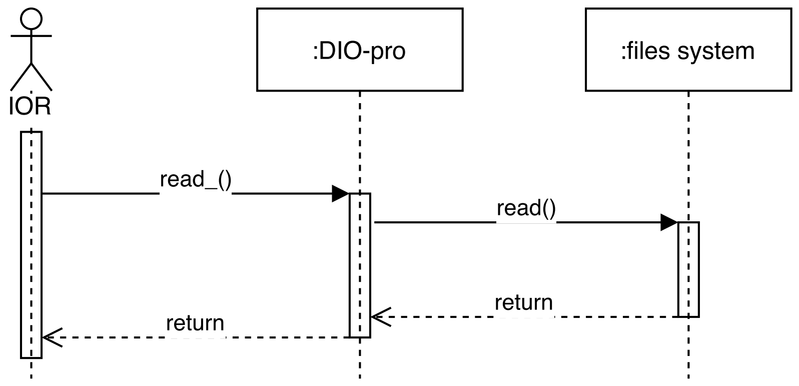

Figure 1.1: Sequence diagram of a function call from IOR that is profiled by DIO-pro. DIO-pro intercepts I/O function calls, in this example `read()`, and redirects the function to the original I/O function calls. At the same time, DIO-pro measures the performance of the function call and records other context information at the same time.

```
$ zcat /tmp/dio-pro/*.gz | dio-pro-xml > iotrace.txt
```

This example decompresses all the binary log files `*.gz` in directory `/tmp/dio-pro/`, converts them into human readable format, and stores all of it into the file `iotrace.txt`.

Similarly, aggregated I/O statistics can be displayed in the terminal using `dio-pro-stat` based on the output of `dio-pro-xml`:

```
$ zcat /tmp/dio-pro/*.gz | dio-pro-xml | dio-pro-stat
```

1.4 What is DIO-pro tracking?

DIO-pro records I/O events in the POSIX and MPIIO interface. Events are recorded per file and process. For each I/O event, the *time stamp*, *time span* and some context variables (event-dependent) are stored. The time stamp have microsecond accuracy. The time spans have nanosecond accuracy for `rdtsc`-enabled processors[1], otherwise the time span accuracy falls back to microseconds. An exact list of support I/O calls tracked by DIO-pro is listed in Table **??** on page **??**. The type and number of function arguments recorded depend on the function.

DIO-pro supports the POSIX and MPIIO interface.

DIO-pro uses `dlsym` (see next section) to profile I/O activity. Profiling with `dlsym` has the advantage that dynamically linked application don't have to be recompiled for profiling. By default in Linux binaries are dynamically linked. Dynamically linked binaries depend on external libraries for execution. Statically linked binaries, on the other hand, are self contained. Especially commercial binaries are sometimes linked statically. Sadly enough, `dlsym` doesn't work with statically linked binaries.

Like all `dlsym`-based profilers, DIO-pro is unable to profile statically linked binaries.

Unfortunately there is no way around this problem in user-space, without extremely intrusive and dangerous profiling methods. We are in progress of developing an in-kernel alternative for

[1]Most Intel processors sport `rdtsc` by default. For more information see Section 2.5.

DIO-pro that is able to track statically linked binaries. A kernel module of DIO-pro has the disadvantage of requiring root access and only handles POSIX calls. We remind the reader that I/O profiling is not a free lunch and can be tricky in certain context! As of April 2016, the kernel module implementation of DIO-pro is not yet available.

DIO-pro's default configuration is not tracking I/O activity in any of the following directories: `/tmp`, `/etc`, `/dev`, `/usr`, `/bin`, `/var`, `/boot`, `/lib`, `/opt`, `/sbin`, `/sys`, and `/proc`.

The rationale for this choice is that, surprisingly enough, there may be a high amount of I/O events during the execution of an application in one or more of these directories. Most I/O events are related to the application, but not directly managed by the application. For example, when executing a binary with `mpirun`, MPI accesses many files in the `/tmp`, for management purposes. The I/O behavior of MPI may not be of particular interest.

If one or more of the above mentioned directories are of particular interest, one can delete them from the `exclusions` array in file `dio-pro-posix.c` to let DIO-pro track them.

1.5 How Does DIO-pro Work?

DIO-pro is able to track I/O activities with the help of `dlsym`. The function `dlsym()` takes a "handle" of a dynamic library returned by `dlopen()` and the null-terminated symbol name, returning the address where that symbol is loaded into memory[2]. `dlsym` basically can redirect function calls of a dynamically linked binary to an arbitrary location. This allows DIO-pro to inject profiling code for I/O related functions. Moreover, the `dlsym` approach does not require root privileges. The drawback of `dlsym` is that it doesn't work for statically linked binaries, i.e., binaries compiled with the `static` flag.

Invoking DIO-pro with `LD_PRELOAD` does not require `root` privileges to work!

Throughout DIO-pro's source code, specifically the files `dio-pro-mpi.c` and `dio-pro-posix.c`, you will find functions looking like the below example:

```
 1 int DIO_PRO_FUNCT(functx)(FILE *fp)
 2 {
 3     int tmp_fd = fileno(fp);
 4     uint64_t start, stop;
 5     int ret;
 6
 7     DO_OR_DIE(functx);
 8
 9     start = Timer_tsc();
10     ret = __functx(fp);
11     stop = Timer_tsc();
12
13     RECORD_DATA(start, stop, ...);
14
15     return(ret);
16 }
```

This is the basic template that DIO-pro uses to profile a function, in this case `functx`. At line 7, original `functx` call is redirected by `dlsym` to DIO-pro's `functx`. The original pointer to `functx` is stored in `_functx`. Here, `DO_OR_DIE()` is a macro definition that contains the `dlsym` magic. Then,

[2]`http://linux.die.net/man/3/dlsym`

the original `functx` is called and encapsulated by a timer. `start` and `stop` measure the start and end of the function call. Section 2.5 explains in more detail what happens in `Timer_tsc()`. After the timer is finished, the time data, and possibly other `functx` arguments, are stored in a temporary log buffer. Then, DIO-proś `functx` returns and the application continues the execution of the binary.

The temporary log buffer is maintained by the so-called *log manager* in the file `log-manager.c`. The log manager contains procedures to write certain data types directly to a *log buffer*. This implies that the final log file will contain binary data. The binary log file, however, is coded quite straight forwardly. The log file's binary encoding is elaborated in Chapter **??**.

Chapter 2

Running DIO-pro

2.1 Objective

This chapter elaborates on the use of DIO-pro: how to run the program and how to deal with the generated data. If you are impatient and don't want to read the whole section, we suggest you to look at the « *DIO-pro Out-of-the-Box* » Section 2.2. This section will get you up and running in no time with the default settings!

2.2 DIO-pro Out-of-the-Box

In the main directory a `Makefile` is located that will compile the essential DIO-pro source code. While in the main DIO-pro directory, to profile an MPI-enabled application, one executes the following command:

```
$ LD_PRELOAD=libs/dio-pro.so ./my_application
```

By default DIO-pro stores recorded I/O traces in `/tmp/dio-pro`. To convert, the binary I/O traces, also referred to as log files, one pipes them into `dio-pro-xml`:

```
$ zcat /tmp/dio-pro/* | xml/dio-pro-xml
```

`dio-pro-stat` is used to compute statistics from the log file (note the `-n` after `dio-pro-xml`):

```
$ zcat /tmp/dio-pro/* | xml/dio-pro-xml -n | stat/dio-pro-stat
```

Custom statistics can also be generated based on the output of `dio-pro-xml`.

2.3 The Basics

In the main directory a `Makefile` is located that will compile the source code in the directories `libs`, `xml`, `stat`. The `libs` directory holds the profiling libraries, to be loaded with the application under investigation. `xml` holds the program to translates the binary log format into human readable format. `stat` holds the program that computes statistics from the human readable log format.

By using different DIO-pro libraries different aspects of a programs I/O can be recorded:

- **MPI** [`dio-pro-mpi.so`]: only MPI I/O function calls are recorded,
- **POSIX** [`dio-pro-posix.so`]: only POSIX function calls are recorded,

- **MPI+POSIX** [`dio-pro.so`][1]: MPI I/O and POSIX function calls are recorded,
- **POSIX (stream-mode)** [`dio-pro-posix.stream.so`]: only POSIX function calls recorded and streamed to a `FIFO`.

By default DIO-pro stores log files in `/tmp/dio-pro`. The binary log files are compressed with `zlib` for the logging involving MPI, otherwise the log file is not compressed. The POSIX-streaming mode does not create a log file. Instead the log is streamed to a `FIFO` file, located by default in `/tmp/dio-pro/dio-pro.fifo`[2]. The MPI-enabled libraries maintain a log buffer in memory, to minimize load on the file-system. For practical motivations, the POSIX libraries don't maintain a memory buffer. Instead log data is written immediately to the log file in the file system.

2.3.1 Collecting I/O Traces

An arbitrary application can be profiled by DIO-pro as follows:

```
$ LD_PRELOAD=libs/dio-pro.so ./my_application
```

The library `dio-pro.so` can be replaced by any of the above mentioned libraries, depending on your needs. After the program returns, a log file should be located in `/tmp/dio-pro`. The log directory can be changed by setting the environment variable `DIO_PRO_LOG_DIR` (see Section 2.4).

Files terminating with `.gz` are compressed and can be printed to `stdout` with `zcat`. Uncompressed log files end with `.bin` and are printed to `stdout` with `cat`.

2.3.2 Converting the Log File

Once, a log file is obtained `dio-pro-xml` can decode the binary format into a human readable format, by default XML:

```
$ zcat /tmp/dio-pro/* | xml/dio-pro-xml
```

An alternative, layout is printed using the `-n` flag. The column layout is as follows:

1. JOBID: the `jobid` of the scheduler (-1 indicates no jobid is detected),
2. PID: the processor unique `pid` of the process,
3. MPI-RANK: the MPI-rank (-1 indicates that the rank is not applicable),
4. IO-TYPE (#): numerical representation of the I/O type recorded,
5. IO-TYPE (STRING): alphanumerical representation of the I/O type recorded.

These are the basic headers elements present for each entry in the log file. Depending on the I/O event type, the following entries may follow:

- TIME STAMP [`t`]: time (seconds) at which the I/O event occurred (processor time),
- TIME SPAN [`dt`]: time (seconds) at which the I/O event occurred (processor time),
- FILE DESCRIPTOR [`fd`]: the file descriptor identifier of the applicable file,
- ...

It is noted that the file descriptor `fd` links that particular I/O event to the first preceding `open` statement with the same file descriptor.

[1]It is noted that the `dio-pro.so` library may be unstable, at the moment, only if POSIX operations are called before any MPI statements!

[2]The `FIFO` file should be emptied by reading its content as fast as possible otherwise DIO-pro will block. This is explained in the sequel.

2.3.3 Computing Statistics

dio-pro-stat is used to compute statistics from the set of log files (note the -n after dio-pro-xml):

```
$ zcat /tmp/dio-pro/* | xml/dio-pro-xml -n | stat/dio-pro-stat
```

By default dio-pro-stat will compute overall I/O performance. To show per-file statistics call dio-pro-stat -s file, and for per-process statistics dio-pro-stat -s proc.

The output of the statistics shows the following information:

- BLOCKSIZES: the different block sizes detected in the log file, number of occurrences are shown between square brackets,
- FUNCTION CALLS: the different types of I/O function calls listed in the log file, number of occurrences are shown between square brackets,
- SIZE: total number of bytes transferred,
- # BLOCKS: total number of blocks transferred,
- TIME: total time spend in I/O function calls,
- MIN SPEED: the fastest I/O transfer time amongst all I/O function calls,
- AVG SPEED: the average I/O transfer time,
- MAX SPEED: the slowest I/O transfer time amongst all I/O function calls,
- STDEV SPEED: the standard deviation of the I/O speed amongst all I/O function calls.

Custom statistics can be generated based on the output of dio-pro-xml.

2.4 Environment Variables

The location and the filename of dio-pro can altered by setting the following two environment values:

- DIO_PRO_JOBID_TAG: defines the environment variable that stores the JOBID. This is usefull because different ecosystems may use different job scheduling software, e.g., slurm. If either DIO_PRO_JOBID_TAG or $DIO_PRO_JOBID_TAG is not set in the environment, then dio-pro set jobid to the value 0.
- DIO_PRO_LOG_DIR: the directory to save the log files in. If DIO_PRO_LOG_DIR is not set then dio-pro falls back to the default log directory at /tmp/dio-pro .

2.5 On the Accuracy of Time-stamps and Time-spans

DIO-pro's I/O traces contain time-stamps t and time-spans Δt. Timestamps have a microsecond accuracy, whereas time-spans have nanosecond accuracy. The nanosecond accuracy for time-stamps was chosen as some *simple* I/O operations, e.g., stat() may be much faster than $1\,\mu$s. The nanosecond accuracy is obtained by reading the clock_tick_count register. The clock_tick_count register is available only for Intel processors and accessible via the rdtsc assembly instruction. A nanosecond value is obtained by dividing clock_tick_count by the number of clock cycles per nano seconds. Intel guarantees that the pace at which clock_tick_count is updated, is not affected by CPU frequency scaling, different CPU power states, etc.

Even though, in a source code a piece of code is encapsulated by `clock_tick_count` read outs, that doesn't guarantee that at runtime the program will be executed in that order. This may happen due to specific compiler optimization off-line or by out-of-order-execution on-line. Such out-of-order-execution may hamper time-span measurements. To prevent out-of-order-execution and force serial execution, parts of the instructions in the pipeline should be flushed[3]. This may incur a small performance cost which has been reported to be worst-case around 100 clock cycles on mainstream retail CPUs. On a high performance CPU this translates to a value of the order of 100 ps. This performance cost is referred to as o_c in Section 3.2. It is noted that there is no free lunch when it comes to accurate performance measurements. A 100 ps cost to obtain nanosecond accurate time measurements is well spent.

[3]"How to Benchmark Code Execution Times on Intel...": `http://www.intel.fr/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf`

Chapter 3

DIO-pro and IOR Statistics Comparison

3.1 Objective

This chapter clarifies the differences between the performance statistics displayed by IOR, a tool for benchmarking parallel file systems, and DIO-pro, DDN's I/O profiling tool.

3.2 Performance Statistics

In this section the differences in performance statistics computation between IOR and DIO-pro are highlighted. In the next section, this theoretical analysis will be applied to a practical example.

Universally, the I/O speed [B/s] is calculated as follows:

$$\text{speed} = \frac{\text{bytes}}{\text{seconds}} = \frac{b}{\Delta t}. \tag{3.1}$$

Both DIO-pro and IOR's notion of b are the same: it is the number of bytes read/written. The reason why IOR and DIO-pro's statistics differ lays in the interpretation of Δt. IOR assumes Δt to be the total execution time of certain code, which include I/O operations. On the other hand, DIO-pro defines Δt as the time spent solely in I/O operations. To understand what IOR[1] exactly does, one needs to dive into IOR's source code (enclosed).

IOR stores time-stamps of certain events in an array called `timer[]`. This time-stamp array is min/max map reduced over the multiple ranks into the `reduced` array. On line `IOR.c:1285` one can see that the total read and write time is calculated by the subtraction of array elements 11/6 and 5/0, respectively. When these indexes of `timer[]` are tracked back in the source code, one observes

- `timer[0]`: time-stamp before `IOR_Create()` (`IOR.c:2004`),
- `timer[5]`: time-stamp after `IOR_Close()` (`IOR.c:2030`),
- `timer[6]`: time-stamp before `IOR_Open()` (`IOR.c:2160`),
- `timer[11]`: time-stamp after `IOR_Close()` (`IOR.c:2178`).

The `read()` and `write()` in IOR for POISX happen at line

- `IOR.c:2173` → `IOR.c:2611` → `aiori-POSIX.c:251`, for reading, and line

[1]This discussion applies to IOR version 2.10.3.

- `IOR.c:2013` $\rightarrow$ `IOR.c:2608` $\rightarrow$ `aiori-POSIX.c:244`, for writing.

Two things are observed: 1) IOR opens a file, reads or writes, but not both, and closes the file, 2) IOR measures Δt as the difference in time of the start of `open()` and the end of `close()`. Moreover, it is clear from the source code that there is more than just I/O operations executed between the `open()` and `close()` statements. Consequently, IOR's definition of Δt is defined as follows:

$$\Delta t_{\mathrm{IOR}} = \mathrm{end}(\texttt{close}()) - \mathrm{start}(\texttt{open}()) = (t_{\texttt{close()}} + \Delta t_{\texttt{close()}}) - t_{\texttt{open()}}. \quad (3.2)$$

DIO-pro, on the other hand, assumes Δt to be the time effectively spend in I/O functions

$$\Delta t_{\mathrm{DIO}} = \sum_k^n \Delta t_k^{\mathrm{i/o}}, \quad (3.3)$$

where $\Delta t_k^{\mathrm{i/o}}$ is the time spend in I/O operation k, and n is the total number of I/O operations executed. We already notice that Δt_{IOR} will always be larger than Δt_{DIO}:

$$\mathrm{end}(\texttt{close}()) - \mathrm{start}(\texttt{open}()) > \sum_k^n \Delta t_k^{\mathrm{i/o}}. \quad (3.4)$$

This follows from DIO-pro's implementation and shown in Figure 1.1: Δt_{DIO} is contained by Δt_{IOR}. In particular, the left-hand side of Equation 3.4 equals to

$$\begin{aligned}\mathrm{end}(\texttt{close}()) - \mathrm{start}(\texttt{open}()) &= \Delta t_{\texttt{open()}} + \Delta t_{\texttt{close()}} + \sum_k^n \Delta t_k^{\mathrm{i/o}} + o_{\mathrm{f}} + o_{\mathrm{a}}(n) + n o_{\mathrm{c}} \\ &= \Delta t_{\texttt{open()}} + \Delta t_{\texttt{close()}} + \Delta t_{\mathrm{DIO}} + o_{\mathrm{f}} + o_{\mathrm{a}}(n) + n o_{\mathrm{c}}, \quad (3.5)\end{aligned}$$

where o_{f} is the time it takes to call DIO-pro from IOR, o_{c} accounts for the time lost due to serializing the pipeline content for `rdtsc` (see Appendix 2.5), and o_{a} is the time spend in between I/O operations on other code between the `open()` and `close()` statements. o_{f} and o_{c} are very small values, of the order of picoseconds, which can only be measured approximately. $o_{\mathrm{a}}(n)$ can assume an arbitrary value depending on the particular program. $\Delta t_{\texttt{open()}}$ and $\Delta t_{\texttt{close()}}$ are much bigger than o_{f} and o_{c}, but could be comparable to a $\Delta t^{\mathrm{i/o}}$ time wise. They are in the order of tens of microsecond but can show a wide variance.

From Equation 3.5 we can conclude that if n grows large enough than the relative weight of $\{\Delta t_{\texttt{open()}}, \Delta t_{\texttt{close()}}, o_{\mathrm{f}}, o_{\mathrm{a}}, o_{\mathrm{c}}\}$ in Δt_{IOR} become negligible small

$$\lim_{n\to\infty} \left[\Delta t_{\texttt{open()}} + \Delta t_{\texttt{close()}} + \sum_k^n \Delta t_k^{\mathrm{i/o}} + o_{\mathrm{f}} + o_{\mathrm{a}}(n) + n o_{\mathrm{c}} \right] \approx \Delta t_{\mathrm{DIO}} = \Delta t_{\mathrm{IOR}}. \quad (3.6)$$

In practice, this means that $\Delta t_{\mathrm{IOR}} = \Delta t_{\mathrm{DIO}}$ for sufficiently large values of n and b, and a well performing storage system.

3.3 Practical Example

To clarify the performance performance statistics differences between IOR-2.10.3 and DIO-pro, a practical example is given. An I/O trace from IOR is recorded by DIO-pro. To keep it simple, a single node is spawned by IOR, which writes and reads a block of 3 chucks of 4096 kB to the file system via the POSIX interface. After IOR is finished, DIO-pro calculates performance statistics based upon the I/O trace. The performance statistics of IOR are taken from `stdout` in the terminal. The output of IOR, `dio-pro-xml` and `dio-pro-stat` can be found in Appendix 3.5.

Table 3.1 shows values inferred from the `dio-pro-xml` output, these include: $\Delta t_{\texttt{open()}}$, $\Delta t_{\texttt{close()}}$, $\Delta t_{\mathrm{i/o}}$, n and b. It is observed that $\Delta t_{\texttt{open()}}$ for the write process is about ten times

Table 3.1: Values inferred from the `dio-pro-xml` output.

Read			Write			Common		
$\Delta t_{\texttt{open()}}$	=	0.866 μs	$\Delta t_{\texttt{open()}}$	=	9.639 μs	n	=	3
$\Delta t_{\texttt{close()}}$	=	0.401 μs	$\Delta t_{\texttt{close()}}$	=	0.694 μs	b	=	12288 B
$t_{\texttt{open()}}$	=	/3.005256 s	$t_{\texttt{open()}}$	=	/3.005212 s			
$t_{\texttt{close()}}$	=	/3.005264 s	$t_{\texttt{close()}}$	=	/3.005250 s			
Δt_1	=	0.712 μs	Δt_1	=	5.726 μs			
Δt_2	=	0.295 μs	Δt_2	=	2.237 μs			
Δt_3	=	0.315 μs	Δt_3	=	2.004 μs			

Table 3.2: Average values experimentally defined by instrumenting IOR.

Read			Write		
$\bar{e}_k$	=	0.219 μs	$\bar{e}_k$	=	0.366 μs
std(e_{k})	=	0.057 μs	std(e_{k})	=	0.075 μs
$\bar{e}_m$	=	2.562 μs	$\bar{e}_m$	=	6.263 μs
std(e_{m})	=	0.448 μs	std(e_{m})	=	1.336 μs

longer than for the read process. This is because the write process also creates the file to write to, whereas when the read `open()` is called, the file to read already exist. We deem o_{f} and o_{c} negligibly small. o_{a} is measured experimentally by instrumenting IOR and averaging the result over 1024 runs. In fact, we split up o_{a} in two parts such that $o_{\mathrm{a}} = e_{\mathrm{k}} + e_{\mathrm{m}}$, where

- e_{k} is the time between the start of the IOR timer[2] and `open()`, plus the time between the end of `close()` and the end of the IOR timer, and
- e_{m} accounts for the time between `open()`/`close()` excluding I/O operations.

We intuitively understand that in most cases e_{k} will be smaller than e_{m}. The values for e_{k} and e_{m} are shown in Table 3.2. It is observed that $\bar{e}$ for the write `open()` is about twice as large as for the read process. Now that we know the variables from Equation 3.3 and 3.5, we can compute Δt_{IOR} and Δt_{DIO}. Δt_{DIO} equals:

$$\begin{aligned}
\Delta t_{\mathrm{DIO}} &= \Delta t_1 + \Delta t_2 + \Delta t_3 \\
\Delta t_{\mathrm{DIO}}^{\mathrm{read}} &= 0.712 + 0.295 + 0.315 = 1.322\,\mu\mathrm{s} \\
\Delta t_{\mathrm{DIO}}^{\mathrm{write}} &= 5.726 + 2.237 + 2.004 = 9.967\,\mu\mathrm{s}.
\end{aligned}$$

Δt_{IOR} can easily be calculated via Equation 3.2:

$$\begin{aligned}
\Delta t_{\mathrm{IOR}} &= t_{\texttt{close()}} - t_{\texttt{open()}} + \Delta t_{\texttt{close()}} + e_{\mathrm{k}} \\
\Delta t_{\mathrm{IOR}}^{\mathrm{read}} &= (/5264 - /5256) + 0.401 + 0.219 = 8.620\,\mu\mathrm{s} \\
\Delta t_{\mathrm{IOR}}^{\mathrm{write}} &= (/5250 - /5212) + 0.694 + 0.366 = 39.060\,\mu\mathrm{s}.
\end{aligned}$$

[2]With *IOR timer* we refer to the timer that measures Δt for OIR.

Accordingly, the read and write speeds can be calculated:

$$\begin{aligned}
\text{speed}_{\text{DIO}}^{\text{read}} &= \frac{12288}{1.322} = 9295.008\,\text{B}/\mu\text{s} = 8.657\,\text{GiB/s} \\
\text{speed}_{\text{DIO}}^{\text{write}} &= \frac{12288}{9.967} = 1232.868\,\text{B}/\mu\text{s} = 1.148\,\text{GiB/s} \\
\text{speed}_{\text{IOR}}^{\text{read}} &= \frac{12288}{8.620} = 1425.522\,\text{B}/\mu\text{s} = 1359.484\,\text{MiB/s} \\
\text{speed}_{\text{IOR}}^{\text{write}} &= \frac{12288}{39.060} = 314.59.\,\text{B}/\mu\text{s} = 300.019\,\text{MiB/s}.
\end{aligned}$$

The values for $\text{speed}_{\text{DIO}}$ correspond to the output of `dio-pro-stat`, which is expected. The calculated values for IOR differ slightly than what is found in the terminal output. The read and write speed show 5.1% and 1.9% error, respectively, compared to IOR's terminal output. The error is introduced by the uncertainty of the o_a measurement. In fact, small changes of o_a can have a large effect on the speed numbers. For example, for the IOR read process $o_a/\Delta t_{\text{IOR}} \approx 32\%$ in the above example. This means that for the read process in IOR, only 32% of the time is actually spend on I/O operations. It is thus understood that o_a can control Δt_{IOR}, and hence also IOR's I/O speed indicator.

In this example, however, only three chucks of 4096 kiB are stored. In more realistic scenarios, much larger chunks, and many more chunks are stored, which would decrease the influence of o_a on Δt_{IOR}.

3.4 Conclusion

It is clear that the speed indicators of IOR and DIO-pro do not correspond. That is because the DIO-pro and IOR have different interpretations of the time-span used to calculate I/O speeds. However, we have shown that when the number of I/O operations are sufficiently large, the speed indicators of both DIO-pro and IOR will converge. Both approaches are equally acceptable, there is no right or wrong here. However, each performance measurement application should be used in its proper context. IOR is designed to test the performance of a file system. For example, the time to flush buffers by `close()` is an important performance indicator that must be accounted for by IOR. DIO-pro on the other hand is designed to measure the I/O performance of general applications. In the context of DIO-pro, o_a, the time spent between I/O operations, can be arbitrarily large. Thus for DIO-pro the IOR approach is not appropriate.

DDN maintains the DIO-pro software, therefore it is possible for us to implement IOR-like performance statistics in parallel to existing algorithms if their exists interest.

3.5 Terminal Outputs

3.5.1 IOR terminal output

```
$ LD_PRELOAD=/home/kare/local/lib/dio-pro.so \
                                        IOR -a POSIX -t 4096 -b 12288 -o testfile.tmp
IOR-2.10.3: MPI Coordinated Test of Parallel I/O

Run began: Fri Apr  8 10:03:23 2016
Command line used: IOR -a POSIX -t 4096 -b 12288 -o testfile.tmp
Machine: Linux karel

Summary:
        api                = POSIX
        test filename      = testfile.tmp
        access             = single-shared-file
        ordering in a file = sequential offsets
        ordering inter file= no tasks offsets
        clients            = 1 (1 per node)
        repetitions        = 1
        xfersize           = 4096 bytes
        blocksize          = 12288 bytes
        aggregate filesize = 12288 bytes


Operation  Max (MiB)  Min (MiB)  Mean (MiB)   Std Dev  Max (OPs)  Min (OPs)  Mean (OPs)
---------  ---------  ---------  ----------   -------  ---------  ---------  ----------
write         294.32     294.32      294.32      0.00   75346.78   75346.78    75346.78
read         1293.47    1293.47     1293.47      0.00  331129.26  331129.26   331129.26

Std Dev  Mean (s)
-------  --------
   0.00   0.00004   EXCEL
   0.00   0.00001   EXCEL


Max Write: 294.32 MiB/sec (308.62 MB/sec)
Max Read:  1293.47 MiB/sec (1356.31 MB/sec)

Run finished: Fri Apr  8 10:03:23 2016
```

3.5.2 DIO-pro log directory

After executing IOR, the log directly of DIO-pro contains the file shown below.

```
$ ls -l -h /tmp/dio-pro/*
-rwxr-xr-x 1 karel karel 286 Apr  8 10:03 \
                              /tmp/dio-pro/iotrace-jobid-1762-rank-0-n410943802.bin.gz
```

3.5.3 `dio-pro-xml` terminal output

The terminal output of the **dio-pro-xml** program can be found in the enclosed file named **dio-pro-xml.txt**. The complete output is to large to fit aesthetically in this document, unfortunately. Instead, below is a summary of the content relevant to the discussion. Three dots (...) indicates that (unimportant) characters were dropped.

```
$ zcat /tmp/dio-pro/iotrace-jobid-1762-rank-0-n410943802.bin.gz | dio-pro-xml
...
<event type="DIO_PRO_POSIX_OPEN64" ... t="1460102603.005212" dt="0.000009639" ... />
<event type="DIO_PRO_POSIX_LSEEK64" ... dt="0.000000306" ... offset="0" />
<event type="DIO_PRO_POSIX_WRITE" ... dt="0.000005726" ... count="4096" />
<event type="DIO_PRO_POSIX_LSEEK64" ... dt="0.000000125" ... offset="4096" />
<event type="DIO_PRO_POSIX_WRITE" ... dt="0.000002237" ... count="4096" />
<event type="DIO_PRO_POSIX_LSEEK64" ... dt="0.000000099" ... offset="8192" />
<event type="DIO_PRO_POSIX_WRITE" ... dt="0.000002004" ... count="4096" />
<event type="DIO_PRO_POSIX_CLOSE" ... t="1460102603.005250" dt="0.000000694" ... />
<event type="DIO_PRO_POSIX_XSTAT64" ... dt="0.000000343" ... path="testfile.tmp" />
<event type="DIO_PRO_POSIX_OPEN64" ... t="1460102603.005256" dt="0.000000866" ... />
<event type="DIO_PRO_POSIX_LSEEK64" ... dt="0.000000149" ... offset="0" />
<event type="DIO_PRO_POSIX_READ" ... dt="0.000000712" ... count="4096" />
<event type="DIO_PRO_POSIX_LSEEK64" ... dt="0.000000094" ... offset="4096" />
<event type="DIO_PRO_POSIX_READ" ... dt="0.000000295" ... count="4096" />
<event type="DIO_PRO_POSIX_LSEEK64" ... dt="0.000000102" ... offset="8192" />
<event type="DIO_PRO_POSIX_READ" ... dt="0.000000315" ... count="4096" />
<event type="DIO_PRO_POSIX_CLOSE" ... t="1460102603.005264" dt="0.000000401" ... />
<event type="DIO_PRO_POSIX_XSTAT64" ... dt="0.000000253" ... path="testfile.tmp" />
<event type="DIO_PRO_POSIX_UNLINK" ... dt="0.000021548" pathname="testfile.tmp" />
...
```

3.5.4 `dio-pro-stat` terminal output

```
$ zcat iotrace-jobid-1762-rank-0-n410943802.bin.gz | dio-pro-xml -n | dio-pro-stat
DIO-pro v0.1: DDNÂő Storage I/O Profiling
Date: 2016-04-08.15:57:11

Overall I/O Statistics:

blocksize(s) read:   4096
blocksize(s) write:  4096

            size       # blocks       time         min speed      avg speed    \
       -------------  ----------  ------------  -------------  -------------   \
read       12.288 kB           3    0.000001 s     5.358 GiB/s    8.657 GiB/s  \
write      12.288 kB           3    0.000010 s   682.195 MiB/s    1.148 GiB/s  \

  max speed     stdev speed
-------------  -------------
 12.931 GiB/s  680.526 MiB/s
  1.904 GiB/s    4.156 GiB/s
```

Appendix A

Tables

A.1 Supported I/O Calls

Table A.1: MPIIO I/O calls tracked by DIO-pro v1.35 and the actual arguments recorded. The associated *type*, as found in DIO-pro's log s, is also shown.

I/O call	Type	Arguments
`MPI_File_open`	`DIO_PRO_MPI_OPEN`	filename, path, amode
`MPI_File_close`	`DIO_PRO_MPI_CLOSE`	fh
`MPI_File_sync`	`DIO_PRO_MPI_SYNC`	fh
`MPI_File_set_view`	`DIO_PRO_MPI_SET_VIEW`	fh, disp
`MPI_File_read`	`DIO_PRO_MPI_READ`	fh, count
`MPI_File_read_at`	`DIO_PRO_MPI_READ_AT`	fh, size, offset
`MPI_File_read_at_all`	`DIO_PRO_MPI_READ_AT_ALL`	fh, size, offset
`MPI_File_read_all`	`DIO_PRO_MPI_READ_ALL`	fh, size
`MPI_File_read_shared`	`DIO_PRO_MPI_READ_SHARED`	fh, size
`MPI_File_read_ordered`	`DIO_PRO_MPI_READ_ORDERED`	fh, size
`MPI_File_read_at_all_begin`	`DIO_PRO_MPI_READ_AT_ALL_BEGIN`	fh, size, offset
`MPI_File_read_begin`	`DIO_PRO_MPI_READ_ALL_BEGIN`	fh, size
`MPI_File_read_ordered_begin`	`DIO_PRO_MPI_READ_ORDERED_BEGIN`	fh, size
`MPI_File_iread`	`DIO_PRO_MPI_IREAD`	fh, size
`MPI_File_iread_at`	`DIO_PRO_MPI_IREAD_AT`	fh, size, offset
`MPI_File_iread_shared`	`DIO_PRO_MPI_IREAD_SHARED`	fh, size
`MPI_File_write`	`DIO_PRO_MPI_WRITE`	fh, size
`MPI_File_write_at`	`DIO_PRO_MPI_WRITE_AT`	fh, size, offset
`MPI_File_write_at_all`	`DIO_PRO_MPI_WRITE_AT_ALL`	fh, size, offset
`MPI_File_write_all`	`DIO_PRO_MPI_WRITE_ALL`	fh, size
`MPI_File_write_shared`	`DIO_PRO_MPI_WRITE_SHARED`	fh, size
`MPI_File_write_ordered_begin`	`DIO_PRO_MPI_WRITE_ORDERED_BEGIN`	fh, size
`MPI_File_write_at_all_begin`	`DIO_PRO_MPI_WRITE_AT_ALL_BEGIN`	fh, size
`MPI_File_write_all_begin`	`DIO_PRO_MPI_WRITE_ALL_BEGIN`	fh, size
`MPI_File_iwrite`	`DIO_PRO_MPI_IWRITE`	fh, size
`MPI_File_iwrite_at`	`DIO_PRO_MPI_IWRITE_AT`	fh, size, offset
`MPI_File_iwrite_shared`	`DIO_PRO_MPI_IWRITE_SHARED`	fh, size
`MPI_File_file_delete`	`DIO_PRO_MPI_FILE_DELETE`	filename
`MPI_File_seek`	`DIO_PRO_MPI_SEEK`	fh, offset, whence
`MPI_File_seek_shared`	`DIO_PRO_MPI_SEEK_SHARED`	fh, offset, whence
`MPI_File_set_size`	`DIO_PRO_MPI_FILE_SET_SIZE`	fh, size
`MPI_Barrier`	`DIO_PRO_MPI_BARRIER`	comm

Table A.2: POSIX I/O calls tracked by DIO-pro v1.35 and the actual arguments recorded. The associated *type*, as found in DIO-pro's log s, is also shown.

I/O CALL	TYPE	ARGUMENTS
mkstemp	DIO_PRO_POSIX_MKSTEMP	template
mkostemp	DIO_PRO_POSIX_MKOSTEMP	template, flags
mkstemps	DIO_PRO_POSIX_MKSTEMPS	template, suffixlen
mkostemps	DIO_PRO_POSIX_MKOSTEMPS	template, suffixlen, flags
creat	DIO_PRO_POSIX_CREAT	path, mode
creat64	DIO_PRO_POSIX_CREAT	path, mode
open	DIO_PRO_POSIX_OPEN	path, flags
open64	DIO_PRO_POSIX_OPEN64	path, flags
close	DIO_PRO_POSIX_CLOSE	fd
write	DIO_PRO_POSIX_WRITE	fd, count
read	DIO_PRO_POSIX_READ	fd, count
lseek	DIO_PRO_POSIX_LSEEK	fd, offset, whence
lseek64	DIO_PRO_POSIX_LSEEK64	fd, offset, whence
pread	DIO_PRO_POSIX_PREAD	fd, count, offset
pwrite	DIO_PRO_POSIX_PWRITE	fd, count, offset
readv	DIO_PRO_POSIX_READV	fd
writev	DIO_PRO_POSIX_WRITEV	fd
__fxstat	DIO_PRO_POSIX_FXSTAT	vers, fd
__lxstat	DIO_PRO_POSIX_LXSTAT	vers, path
__xstat	DIO_PRO_POSIX_XSTAT	vers, path
unlink	DIO_PRO_POSIX_UNLINK	pathname
mmap	DIO_PRO_POSIX_MMAP	length, prot, flags, fd, offset
mmap64	DIO_PRO_POSIX_MMAP64	length, prot, flags, fd, offset
fopen	DIO_PRO_POSIX_FOPEN	path, mode
fopen64	DIO_PRO_POSIX_FOPEN64	path, mode
fclose	DIO_PRO_POSIX_FCLOSE	fp
fread	DIO_PRO_POSIX_FREAD	size, nmemb, stream
fwrite	DIO_PRO_POSIX_FWRITE	size, nmemb, stream
fseek	DIO_PRO_POSIX_FSEEK	stream, offset, whence
fsync	DIO_PRO_POSIX_FSYNC	fd
fdatasync	DIO_PRO_POSIX_FDATASYNC	fd
pread64	DIO_PRO_POSIX_PREAD64	fd, count, offset
pwrite64	DIO_PRO_POSIX_PWRITE64	fd, count, offset
__fxstat64	DIO_PRO_POSIX_FXSTAT64	vers, fd
__xstat64	DIO_PRO_POSIX_XSTAT64	vers, path
__lxstat64	DIO_PRO_POSIX_LXSTAT64	vers, path