I/O analysis using DTrace

By Senthil Ramanujam on Jun 24, 2009

Introduction:

In almost any commercial application, the role of storage is critical, in terms of capacity, performance and data availability. The future roadmap of the storage industry shows disk storage getting cheaper, capacity getting larger, and SAS technology acquiring wider adoption.

Disk performance is becoming more important as CPU power has increased, and is more rarely the bottleneck. Newly available analysis tools have provided a simpler way to understand the internal workings of the I/O subsystem as part of the Operating System, increasing effectiveness of solving I/O performance issues.

Solaris's powerful and open sourced DTrace set of tools allows any user interested in solving I/O performance issues to observe how I/O are carried out by the Operating System and related drivers. The ever increasing set of DTrace features and adoption by various vendors is a strong proof of its usefulness.

Many performance problems can be captured and solved using basic performance tools, such as vmstat, mpstat, prstat and iostat. However, some problems require deeper analysis to be resolved, or require live analysis to solve performance problems and analyze events out of the ordinary.

Environment:

DTrace allows analysis of real or simulated workloads, of arbitrary complexity. There are several applications that can be used to generate the workload. My choice was to use Oracle database software, in order to generate a more interesting and realistic I/O workloads.

An Oracle table was created similar to a fact table in a data warehousing environment, then loaded with data from a flat-file directly into the fact table. This is a write-intensive workload that generates almost all writes to the underlying disk storage.

Just a few important points about the whole setup:

- Solaris10 or OpenSolaris is required to run DTrace commands.
- No filesystem was used for this experiment. Since we have few filesystem options, and the architecture of these filesystems differ, I might address filesystem performance analysis in a future blog.
- Only one disk was used, to keep it very simple. However, I don't see any problem using the same techniques to diagnose I/O issues on a huge disk farm environment.
- Although I used a workload similar to real, the main goal is to monitor the I/O subsystem using DTrace. So, let's not dive deep into the tricks used to generate I/O load. They are simple enough to generate the target workload.
- The scripts to setup the environment are in the zip-file.

- An account with admin privilege can run DTrace commands. In fact, an account with just bare minimum DTrace privilege can run DTrace commands as well. For more details, check out the document here.

Live Analysis:

Note that while the workload was being generated, iostat was started, and then the DTrace commands to get more details. Since iostat without any arguments wouldn't produce the desired details, extended statistics were requested. The options used are:

```
iostat -cnxz 2
```

The quick summary of those arguments:

```
-c : display CPU statistics
-n : display device names in descriptive format
```

```
   -x : display extended device statistics
   -z : do not display lines with all zeros
    2 : display io statisitcs for every 2 seconds
```

For more details, check out the man page.

About 6 SQL*Loader sessions were started to load the data, which in-turn generated huge writes to the disk where the table was located. Oracle SQL*Loader is a utility that is used to load flat-file data into a table. The disk was saturated with about 4-6 SQL*Loader sessions. The following iostat output was captured during the run:

```
     cpu
 us sy wt id
 58  8  0 34
                    extended device statistics
    r/s    w/s   kr/s    kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.0    3.0    0.0    23.9  0.0  0.0    0.0    4.2   0   1 c0t1d0
    0.0  520.4    0.0 32535.2  0.0  4.9    0.0    9.4   2  98 c2t8d0
    0.0    1.5    0.0   885.1  0.0  0.0    0.0   13.5   0   1 c2t13d0
     cpu
 us sy wt id
 57  7  0 36
                    extended device statistics
    r/s    w/s   kr/s    kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.0  546.7    0.0 31004.3  0.0  9.3    0.0   16.9   1  98 c2t8d0
    0.0    1.0    0.0   832.1  0.0  0.0    0.0   11.7   0   1 c2t13d0
```

The tablespace, where the target table was created, is on the disk device identified by c2t8d0. We can see that we are writing about 30MB/sec. Let's now use DTrace to explore further...

Let's start with very basic info. The following DTrace command prints the number of reads, writes and asynchronous I/Os initiated on the system. [In this example, and in **all other examples** in this article, white space has been added for readability. In real life, you would either put the whole dtrace command on a single line, or would put it in a file.]

```
# dtrace -qn
   'BEGIN {
      rio=0;
      wio=0;
      aio=0
   }
   io:::start {
      args[0]->b_flags&B_READ?rio++:0;
      args[0]->b_flags&B_WRITE?wio++:0;
      args[0]->b_flags&B_ASYNC?aio++:0
   }
   tick-1s {
      printf("%Y: reads :%8d, writes :%8d, async-ios :%8d\\n", walltimestamp, rio, wio, aio);
      rio=0;
      wio=0;
      aio=0
   }'
2009 Jun 15 13:11:39: reads :       0, writes :     578, async-ios :     465
2009 Jun 15 13:11:40: reads :       0, writes :     526, async-ios :     520
2009 Jun 15 13:11:41: reads :       0, writes :     529, async-ios :     528
2009 Jun 15 13:11:42: reads :       0, writes :     533, async-ios :     519
2009 Jun 15 13:11:43: reads :       0, writes :     779, async-ios :     622
2009 Jun 15 13:11:44: reads :       0, writes :     533, async-ios :     472
2009 Jun 15 13:11:45: reads :       0, writes :     511, async-ios :     513
2009 Jun 15 13:11:46: reads :       0, writes :     538, async-ios :     531
2009 Jun 15 13:11:47: reads :       0, writes :     620, async-ios :     503
2009 Jun 15 13:11:48: reads :       0, writes :     512, async-ios :     504
2009 Jun 15 13:11:49: reads :       0, writes :     533, async-ios :     526
2009 Jun 15 13:11:50: reads :       0, writes :     565, async-ios :     523
2009 Jun 15 13:11:51: reads :       0, writes :     662, async-ios :     539
2009 Jun 15 13:11:52: reads :       0, writes :     527, async-ios :     524
2009 Jun 15 13:11:53: reads :       0, writes :     501, async-ios :     503
2009 Jun 15 13:11:54: reads :       0, writes :     587, async-ios :     531
2009 Jun 15 13:11:55: reads :       0, writes :     585, async-ios :     532
```

The DTrace command used above has a few sections. BEGIN gets executed only once when the script starts to execute. The io:::start gets executed everytime when an I/O is initiated. tick-1s gets executed for every second.

The above output clearly shows that the system has been issuing a lot of asynchronous I/O to write data. It is easy to verify this behavior by running a truss command against one of those background Oracle processes. Since it's confirmed that only 'writes' are issued, not 'reads', the following commands will focus only on 'writes'.

We will now look at the histogram of write I/O sizes using DTrace aggregation feature. The aggregation function, quantize(), is used to generate histogram data. The quick summary of DTrace aggregation functions can be found here.

```
# dtrace -qn
   'io:::start /execname=="oracle" && args[0]->b_flags & B_WRITE/ {
       @[args[1]->dev_pathname] = quantize(args[0]->b_bcount);
    }
```

```
    tick-10s {
       exit(0)
    }'

  /devices/pci@3,700000/pci@0/scsi@8,1/sd@8,0:a
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@                               1324
           16384 |                                         0
           32768 |@@@@@@@@                                 1032
           65536 |@@@@@@@@@@@@@@@@@@                       2182
          131072 |@@@@                                     518
          262144 |                                         6
          524288 |                                         0
```

Before interpreting the output, let's examine the command used. The 2 new filters within forward slashes are used to a) enable DTrace only for executable name "oracle" and b) enable DTrace only if it's a write request. The quantize aggregation function creates multiple buckets for a range of I/O size and tracks it.

The DTrace output has a physical device name. It is easy to map it to the device that is used at the application layer.

```
# ls -l /dev/rdsk/c2t8d0s0
lrwxrwxrwx   1 root     root          54 Feb  8  2008
   /dev/rdsk/c2t8d0s0 -> ../../devices/pci@3,700000/pci@0/scsi@8,1/sd@8,0:a,raw
#
```

That's it. The device that is used in database is in fact a symbolic link to the device that showed up above in the DTrace output. The above output suggests that the size of more than half-of I/O requests are between 65536 (64KB) and 131071 (~128KB). Since the database software and the OS are capable of writing about upto 1MB per second, there is an opportunity to look closely the database configuration and OS kernel parameters to optimize it. This is one of the useful DTrace commands that can be used to observe how much throughupt a device can deliver. More importantly, it helps discover the I/O (write) pattern of an application.

We will now look at the latency histogram.

```
# dtrace -qn
   'io:::start /execname=="oracle" && args[0]->b_flags & B_WRITE/ {
       io_start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
    }
    io:::done / (io_start[args[0]->b_edev, args[0]->b_blkno]) && (args[0]->b_flags & B_WRITE) / {
       @[args[1]->dev_pathname, args[2]->fi_name] =
           quantize((timestamp - io_start[args[0]->b_edev, args[0]->b_blkno]) / 1000000);
       io_start[args[0]->b_edev, args[0]->b_blkno] = 0;
    }
    tick-10s {
       exit(0)
    }'

  /devices/pci@0,600000/pci@0/pci@8/pci@0/scsi@1/sd@1,0:h  orasystem
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
               8 |                                         0

  /devices/pci@0,600000/pci@0/pci@8/pci@0/scsi@1/sd@1,0:h  control_003
           value  ------------- Distribution ------------- count    0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      7
               2 |@@@@@                                    1
               4 |                                         0

  /devices/pci@0,600000/pci@0/pci@8/pci@0/scsi@1/sd@1,0:h  control_001
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |@@@@@@@@@@                               2
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      6
               8 |                                         0

  /devices/pci@0,600000/pci@0/pci@8/pci@0/scsi@1/sd@1,0:h  control_002
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 8
               8 |                                         0

  /devices/pci@0,600000/pci@0/pci@8/pci@0/scsi@1/sd@1,0:h  oraundo
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |                                         1
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  271
               8 |@                                        5
              16 |                                         2
              32 |                                         0
```

The preceding DTrace command now has more options, io:::start followed by io:::done. The io:::start gets executed every time when an I/O is initiated and io:::done gets executed when an I/O is completed. Since these 2 routines get executed every time when an I/O event occurs, we want to make sure an I/O that

is being tracked when it's initiated is the same I/O being tracked when it's completed. This is why the above command tracks each I/O by its device name and its block number. This, for most cases, gives us the result we want.

However, the elapsed time histogram output shows data for all files except the device that we are interested in. Why? I believe it's due to the effect of asynchronous I/O. (Expect a future blog about monitoring asynchronous I/O). The above output would have been much different if synchronous I/Os are issued to transfer data. The output is still interesting to see that there has been some 'writes' to other database files. For example, it shows the response-time of most 'writes' to various database files are well under 8 milliseconds.

Finally, let's look at disk blocks (lba) that are being accessed more frequently.

```
# dtrace -qn
   'io:::start /execname=="oracle" && args[0]->b_flags & B_WRITE/{
       @["block access count", args[1]->dev_pathname] = quantize(args[0]->b_blkno);
    }
    tick-10s {
       exit(0)
    }'

  block access count                                /devices/pci@3,700000/pci@0/scsi@8,1/sd@8,0:a
           value  ------------- Distribution ------------- count
         4194304 |                                         0
         8388608 |@@@                                      324
        16777216 |                                         8
        33554432 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4237
        67108864 |                                         0

#
```

We see that the range of blocks between 33554432 and 67108863 are being accessed much more frequently than any other block ranges listed above. The use of this command is rather occassional. It is more useful if an application provides any mapping info between the application block and the physical block, to figure out a hot block at the disk level and where it's located.

Conclusion:

We can observe how easy it is take a close look at the OS/physical layer when an application is busy with its disk-based workload. The DTrace collection can be widned or narrowed by using various filter options. For example, the read-intensive workloads can be observed by flipping a filter flag, from B_WRITE to B_READ.

Addtional references:

- DTrace I/O provider
- Sun's wiki website
- Solaris performance tools book

If you are interested in any specific topics similar to above experiment, feel free to drop a note in the comments section.

Category: Best Practices

Tags: dtrace opensolaris performance

Permanent link to this entry

« New CPU2006 Records:... | Main | Sun SSD Server Platf... »
Comments:

Excellent article!! Well presented.
We got the core purpose of the article and we can't use this in our Windows environment. If you have the same for Windows world, it would be the big help for windows users.

Good work.

~Venkat

Posted by **Venkat Paranjothi** on June 24, 2009 at 02:00 AM PDT #

Post a Comment:
Comments are closed for this entry.

About

BestPerf is the source of Oracle performance expertise. In this blog, Oracle's Strategic Applications Engineering group explores Oracle's performance results and shares best practices learned from working on Enterprise-wide Applications.

Index Pages

- Oracle Solaris 11 Benchmarks

- BestPerf Index 26 September 2013

Search

Enter search term:

Search only this blog

Recent Posts

- SPARC T5-2 Delivers World Record 2-Socket SPECvirt_sc2010 Benchmark
- SPARC T5-2 Produces SPECjbb2013-MultiJVM World Record for 2-Chip Systems
- SPARC M6-32 Delivers Oracle E-Business and PeopleSoft World Record Benchmarks, Linear Data Warehouse Scaling in a Virtualized Configuration
- SPARC T5-2 Delivers World Record 2-Socket Application Server for SPECjEnterprise2010 Benchmark
- World Record Single System TPC-H @10000GB Benchmark on SPARC T5-4
- SPARC M6-32 Delivers Oracle E-Business and PeopleSoft World Record Benchmarks, Linear Data Warehouse Scaling in a Virtualized Configuration
- SPARC T5-8 Delivers World Record Single Server SPECjEnterprise2010 Benchmark, Utilizes Virtualized Environment
- SPARC T5-2 Server Beats x86 Server on Oracle Database Transparent Data Encryption
- SPARC T5-8 Delivers World Record Oracle OLAP Perf Version 3 Benchmark Result on Oracle Database 12*c*
- SPARC T5 Encryption Performance Tops Intel E5-2600 v2 Processor

Top Tags

- $/perf
- amd
- application
- bandwidth
- cluster
- cmt
- database
- encryption
- flash
- fusion
- fusion-middleware
- hp
- hpc
- ibm
- intel
- java
- linux
- mcae
- middleware
- oltp
- oow
- opensolaris
- openstorage
- oracle
- peoplesoft
- performance
- response-time
- sap-sd
- scalability
- solaris
- sparc
- sparc64
- spec
- speccpu
- specfp
- specint
- ssd
- storage
- suse
- t4
- t4-2
- t4-4
- t5
- virtualization

- web
- world-record
- x64
- x86
- zfs
- zones

Categories

- Benchmark
- Best Practices
- General Information
- Index
- Oracle

Archives

« March 2014

Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Today

Bookmarks

- blogs.oracle.com
- henk (SAE Eng)

Important Links

- java.com
- java.net
- opensolaris.org
- oracle.com
- otn

Menu

- Blogs Home
- Weblog
- Login

Feeds

RSS

- All
- /Benchmark
- /Best Practices
- /General Information
- /Index
- /Oracle
- Comments

Atom

- All
- /Benchmark
- /Best Practices
- /General Information
- /Index
- /Oracle
- Comments