Kathleen Locher

Dr. Foxwell

CS 671

March 27, 2008

Tracing ZFS on OS X

Abstract

DTrace is a tool which can be used to observe behavior in the kernel, including OpenSolaris's Zettabyte File System (ZFS), and pinpoint problem areas. Both DTrace and ZFS have been ported in some measure to Apple's OS X in version 10.5. Unfortunately DTrace cannot be used to observe ZFS on OS X. This paper describes the sources of this problem. The paper then goes on to describe ways of using DTrace to observe the layers of OS X around ZFS in order to gain some understanding of how ZFS-formatted volumes behave on OS X.

Introduction

Apple created a lot of excitement over the possibility of porting two Solaris features to Mac OS X 10.5, code named Leopard. The first feature was DTrace, the dynamic observability tool which instruments operating systems and computer applications from the kernel up. OS X Leopard includes a port of DTrace. The other potential port from Solaris was ZFS. ZFS is the Zettabyte File System, a next-generation file system which makes many older file system assumptions obsolete. The OpenSolaris community describes ZFS as “a new kind of file system that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability” (“OpenSolaris Community: ZFS”).

ZFS only partially made it into OS X Leopard. ZFS is not the default file system for Leopard. The installed versions of the ZFS commands are read-only. This means that only commands which read ZFS data can be executed on Leopard. Mac OS Forge hosts a project which is working on a read-write version of ZFS for OS X. OS Forge is a site for open source projects on Darwin, the current Mac OS X kernel. Downloading the binaries from this site allows OS X Leopard users access to a beta read-write version of ZFS. While this beta version cannot be used as the root file system for OS X, ZFS-formated file systems can be created, written to, read from, and deleted.

The Virtual File System Layer

The operating system accesses ZFS is through the virtual file system (VFS). The VFS sits between the file-system dependent code and the rest of the kernel. All code which needs to use a file system talks to the VFS. This layer abstracts file system implementation details from the rest of the kernel. According to the book Mac OS X Internals: A Systems Approach, the concept of a VFS was originally created by Sun Microsystems. In a VFS, a virtual node (vnode) provides the representation of a file while the VFS itself is responsible for representing the actual file system (Singh Section 11.6). Multiple file system support, such as OS X's support for HFS, HFS+, UFS, NFS, ZFS, and many others, is enabled through a VFS layer.

DTrace and Kernel Extensions on OS X

Because the read-write ZFS project is not part of any Darwin release, ZFS cannot be built into the kernel itself. On OS X ZFS is a kernel extension (KEXT). KEXTs have access to kernel

programming interfaces (KPIs) and run with kernel privileges. KEXTs must also conform to kernel restrictions. OS X offers the ability to load and unload KEXTs at runtime.

One KPI enables programming to the virtual file system (VFS) layer of Darwin. Apple's documentation explains that “the ability to dynamically add a new file-system implementation is based on VFS KEXTs” (“Kernel Extension Programming Topics”). In order to use ZFS, the appropriate KEXT is loaded when a ZFS-related command is first run.

According to Noël Dellofano, the lead developer on the OS X ZFS port, DTrace on Leopard does not currently support instrumentation of KEXTs. Mr. Dellofano explained this on the zfs-discuss list for the OS X Forge project (“The zfs-discuss March 2008 Archive by Thread”) more than once in his discussions with both James Snyder and the author. This claim has been confirmed by James McIlree of Apple on the dtrace-discuss list of OpenSolaris. The DTrace team has simply not yet implemented this functionality (“Thread: Using DTrace on Kernel Extensions in OS X”).

In the source for the beta of ZFS, probes are currently not correctly implemented. This lack of probes makes sense, as DTrace on OS X would not be able to load them. Some of the statically defined DTrace probes that are present in the latest OpenSolaris revision of ZFS are present in the OS X revision as well. However, adding statically-defined probes in OS X is different than adding them in OpenSolaris. This difference is described at the end of the OS X DTrace man page (“Mac OS X Manual Page for dtrace(1m)”). In the OS X code, the static probes which are present are written to conform to OpenSolaris interfaces rather than OS X interfaces. Because of the differences between static probe definition on OS X and OpenSolaris, these probes are not added to any provider. No additional code appears to exist to either add an

additional module to an existing provider or add any OS X-style statically defined probes. Because of these restrictions, it is impossible to trace ZFS calls on OS X without significant modification to the ZFS and DTrace project code.

On OpenSolaris, the equivalent of KEXTs are loadable kernel modules. DTrace on OpenSolaris has the ability to instrument loadable kernel modules. KEXTs should certainly be traceable at some point in the future of OS X. Until the needed changes for instrumenting ZFS on OS X are made, systems administrators and developers who wish to instrument ZFS on OS X cannot expect any probes in the ZFS code itself.

Observation Utilities Provided by ZFS

What can inquiring DTrace users do now in order to investigate the behavior of ZFS on OS X? Even without DTrace, the zfs and zpool commands themselves provide a few built-in tools which provide helpful information. These tools are a useful place to begin debugging even on OpenSolaris where DTrace can instrument ZFS.

A ZFS file system is not mounted directly. Instead, ZFS file systems are mounted within zpools. A zpool is a grouping of storage devices. The space and I/O bandwidth of all devices can be open to many ZFS file systems all mounted on the zpool at once. The concepts of volumes and partitions as well as the associated maintenance headaches are all removed by the pooled storage model provided by ZFS.

The main command to use for investigating the health zpools, the command `zpool status` provides information on the health of a specific pool. Sample output is provided below:

```
 pool: cs671
 state: ONLINE
status: The pool is formatted using an older on-disk format. The pool
```

```
   can still be used, but some features are unavailable.
   action: Upgrade the pool using 'zpool upgrade'. Once this is
   done, the pool will no longer be accessible on older software
  versions.
 scrub: none requested
config:
      NAME    STATE   READ WRITE CKSUM
      cs671    ONLINE    0   0   0
       disk1s1  ONLINE    0   0   0
```

In order to look at raw I/O on a zpool, the command `zpool iostat` can be used. This command shows IO activity on zpools. Such an overview of IO activity can provide a starting point for pinpointing application and operating system bottlenecks.

Using DTrace to Investigate ZFS with Existing Probes

Despite the crippling lack of instrumentation at the ZFS layer itself, pieces of the kernel which ZFS interacts with can be instrumented by DTrace. Several VFS and vnode related probes are available in the fbt (function boundary tracing) provider in Darwin. Examining the VFS layer while mounting a ZFS volume or working with a ZFS-formatted file system provides some idea of calls that are made to or by ZFS itself. This examination can be conducted by writing fbt probes and using them while running ZFS commands or performing file system operations with a ZFS-formatted device. The input/output (I/O) layer can still be instrumented with the io provider as well. Unfortunately ZFS itself often makes instrumentation of the I/O layer with the io provider difficult. The lack of instrumentation of the ZFS layer itself is definitely an issue which should be addressed. However, some tracing can be done until that point.

A small D script is sufficient to illustrate that the interactions between ZFS and the VFS layer can be traced. The script is shown below. Note that this script, because it relies on the fbt provider, is not portable. The fbt provider is considered not stable according to the stability tables

provided by OpenSolaris. Changes to the architecture of the kernel will change the fbt provider's contents. This particular script was written against OS X 10.5.2.

```
!/usr/sbin/dtrace -s
#pragma D option quiet
self int trace;
dtrace:::BEGIN
{
    printf("%3s%12s%8s\n","PID","Execname","Type");
}
fbt::VFS_MOUNT:entry
{
    printf("%i%12s%8s\n",$pid,execname,"MOUNT");
}
fbt::VFS_UNMOUNT:entry
{
    printf("%i%12s%8s\n",$pid,execname,"UNMOUNT");
}
```

The above script was run during the creation and deletion of a zpool. The pool, named cs671, was first created with the command `zpool create cs671`. This caused the mount line in the output shown below. Next, the pool was destroyed with the command `zpool destroy -f cs671`. The -f option is required to force the pool to be unmounted. The `zpool destroy` command does not yet have the ability to unmount the USB device from the system.

```
PID  Execname  Type
936    zpool  MOUNT
936    zpool UNMOUNT
```

A similar interaction takes place when running the script while creating an actual file system. A ZFS file system is created by calling `zfs create` with the pool, a slash, and then the file system name as arguments. For example, to create a file system named data on the pool cs671, use the command `zfs create cs671/data`. In the case of creating the zpool and then

creating the file system, the following data is produced:

```
PID  Execname  Type
950    zpool  MOUNT
950     zfs  MOUNT
```

The I/O usage of the zpool and zfs creation can also be measured. Apple has ported some of Brendan Gregg's DTaceToolkit scripts to support Darwin's kernel. These ported scripts are available in the open source DTrace project available at the Apple Developer Connection. One script from this collection which can be used to measure I/O is the iofileb.d. Running the `zpool create` and then `zfs create` commands produces the following output when instrumented with iofileb.d.

```
  PID CMD        KB FILE
15447 zpool       4 ??/zfs/zpool.cache
15465 zfs        24 ??/lib/libzfs.dylib
15447 zpool      36 ??/Resources/VolumeIcon.icns
15447 zpool      40 ??/lib/libzfs.dylib
15465 zfs        56 ??/sbin/zfs
```

Actions on individual vnodes in the VFS layer can also be probed with the fbt provider. Another ported DTraceToolkit script is rwbytype.d, which shows reads and writes by type. This D script uses the syscall and fbt providers to look at the read and write system calls as well as the read and write methods in the VFS. The script lists process Ids, program names, vnode operating types, whether the operation is a read (R) or a write (W), and the number of bytes read or written. Running this script during the creation of a new ZFS file system shows that ZFS is interacting with the vnodes in the VFS layer. Running rwbytype during the creation of the ZFS file system cs671/data produced the following output for the zfs command:

```
PID  CMD         VTYPE DIR   BYTES
724  zfs          chr  R    32
```

Not only can actions by the `zfs` command itself be instrumented, but reads and writes of files on a ZFS-formatted drive are also shown.. The following small Java program creates the same file repeatedly on the ZFS file system

```
package edu.gmu.cs671.klocher;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class FileCreator {
     public static void main(String[] args) {
          Runnable runnable = new Runnable() {
               public void run() {
                    try {
                         // the argument here is a path to
                         // a ZFS file system
                         BufferedWriter writer =
                              new BufferedWriter(
                              new FileWriter(
                              "/Volumes/cs671/data/test.txt"));
                         for(int i = 0; i < 100; i++)
                         {
                              writer.write("This is a file");
                         }
                         writer.flush();
                         writer.close();
                    } catch (IOException ex) {
                         ex.printStackTrace();
                    }
               }
          }
          ScheduledThreadPoolExecutor exec = new
ScheduledThreadPoolExecutor(1);
          exec.scheduleAtFixedRate(runnable, 0, 2,
TimeUnit.SECONDS);
     }
}
```

ZFS on OS X requires all actions to be performed as root. There are strong possibilities of a kernel panic when performing ZFS I/O with the beta version, so this restriction is very useful as a sanity check. However, for the small Java program to run correctly the java command must be invoked by root or the file system owner must be changed. Once the program has the correct permissions to access the ZFS mount, the program runs smoothly. Instrumenting this

program with rwbytype.d produces the following output:

```
PID  CMD        VTYPE DIR   BYTES
767  java        reg  W   7000
```

As expected, java has generated several bytes of output. The exciting thing is that the file was being written on the ZFS file system cs671/data. DTrace was not able to instrument the ZFS specific calls, but it was able to instrument the VFS calls and system calls required to do this work, allowing the user to at least see the logical reads and writes being performed.

An interesting question for the cs671/data file system created on the USB drive is how much time is spent waiting for or executing I/O operations. Unfortunately, the io provider is not useful for reads and writes done with ZFS. When run against the same Java program which shows read and write output at the system call and VFS level above, iofileb.d did not show any output for the java command or any obviously zfs related commands. There is no way to correlate such any other processes with ZFS for sure. The io provider could be used to instrument the zpool and zfs commands themselves because they were run on an HFS+ formatted volume, the default for OS X Leopard. However, reads and writes to a ZFS-formatted file system are conspicuously absent from such DTraceToolkit-based scripts as iofileb.d and iofile.d. The lack of information about disk I/O for ZFS-formatted devices is truly a loss for ZFS on OS X.

Conclusion

The prohibition against tracing KEXTs in the current implementation of ZFS on DTrace provides a limitation in its observability that must be fixed. However, ZFS itself provides some monitor tools as part. Instrumentation at the system call , VFS, and I/O layers provide some attempts at bypassing this problem, though data from this instrumentation may be inadequate in

some cases. OS X users can still do some work with DTrace against ZFS. Unfortunately, the internals of ZFS will remain opaque until OS X allows instrumentation of KEXTs with DTrace.

Works Cited

"Kernel Extension Programming Topics". Online. Internet. 13 October 2007. Apple. 19 March 2008. <http://developer.apple.com/documentation/Darwin/Conceptual/KEXTConcept/index.html>

"Mac OS X Manual Page for dtrace(1m)". Online. Internet. July 2006. Apple. 22 March 2008. <http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/dtrace.1.html>

"OpenSolaris Community: ZFS". Online. Internet. 26 June 2007. Sun Microsystems. 22 March 2008. <http://www.opensolaris.org/os/community/zfs/>

Singh, Amit. Mac OS X Internals: A Systems Approach. Online. Internet. 19 June 2006. Amit Singh. 22 March 2008. <http://safari.oreilly.com/0321278542>

"The zfs-discuss March 2008 Archive by Thread". Online. Internet. 24 March 2008. <http://lists.macosforge.org/pipermail/zfs-discuss/2008-March/thread.html>

"Thread: Using DTrace on Kernel Extensions in OS X" Online. Internet. 23 October 2007. Sun Microsystems. 24 March 2008. <http://www.opensolaris.org/jive/thread.jspa?messageID=218276>