SystemTap For Runtime Analysis of Kernel Modules such as AFS

Yadavendra Yadav
Prashant Sodhiya

IBM
Agenda

✓ Introduction
✓ Architecture
✓ Advantages of SystemTap
✓ Using SystemTap
✓ SystemTap Language
✓ Introduction & Examples of Tapsets
✓ Tapsets for OpenAFS
✓ SystemTap Usage in AFS
✓ Case Studies
✓ Performance Measurement
SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system.

It is based on kprobes / kretprobe.

Eliminates the tedious and disruptive process of instrumentation, recompile, install, and reboot sequence that may be otherwise required to collect data.

Provides a simple command line interface and scripting language for writing instrumentation for a live running kernel.
SystemTap Target Audience:

- **Kernel Developer**: I wish I could add debug statements easily without going through the insert / build / reboot cycle.

- **Technical Support**: How can I get additional data out of a customer's kernel easily and safely?

- **System Admin**: Occasionally jobs take significantly longer than usual to complete, or do not complete. Why?

- **Student**: How can I learn more about the call flow of a kernel subsystem?
o System tap uses Kprobes / Kretprobe for dynamic probing.

o Kprobes requires that you:
  ✓ Write a kernel module.
  ✓ Specify an address and handler for each probe point.
  ✓ Be careful! Mistakes can crash the system.
  ✓ Powerful, but cumbersome to use.
SystemTap Processing Steps:

- top-syscalls.stp
- stap_<random#>.c
- stap_<random#>.ko

**STAP**
- parse
- elaborate
- translate
- build

**STAPRUN**
- load/run
- store output
- stop/unload

**SYSTEMTAP**

**DEBUG SYMBOLS**
- debuginfo

**TAPSETS**
- script library

**RUNTIME**
- support functions

**KERNEL**
- kprobes, relays

**output**
Advantages of SystemTap

✓ No module writing required. Create and insert probes quickly and easily using a simple scripting language.

✓ No kprobes knowledge required.

✓ No kernel addresses required. Automates gathering of symbol information.

✓ Provides pre-written probes for common kernel areas.

✓ Growing set of pre-written scripts.

✓ Powerful and simple to use.
Using SystemTap

**Installation & Setup**

To deploy SystemTap, install the following RPMs

- Systemtap
- Systemtap-runtime

Installing requires Kernel Information RPM

- Kernel-debuginfo
- kernel-debuginfo-common-arch
- kernel-devel
- For probing OpenAFS we need to install openafs-debuginfo package.
Using SystemTap (cont…)

Verifying Installation

stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'.

Pass 1: parsed user script and 45 library script(s) in 340usr/0sys/358real ms.
Pass 2: analyzed script: 1 probe(s), 1 function(s), 0 embed(s), 0 global(s) in 290usr/260sys
/568real ms.
Pass 3: translated to C into "/tmp/stapiArgLX/stap_e5886fa50499994e6a87aadcd43cd392_399.c"
in 490usr/430sys/938real ms.
Pass 4: compiled C into "stap_e5886fa50499994e6a87aadcd43cd392_399.ko" in 3310usr/430sys
/3714real ms.
Pass 5: starting run.
read performed
Pass 5: run completed in 10usr/40sys/73real ms.
### System Tap Command

`stap [options] script.stp`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v</td>
<td>Increase verbosity</td>
</tr>
<tr>
<td>-g</td>
<td>Guru mode, embedded C allowed</td>
</tr>
<tr>
<td>-k</td>
<td>Keep temporary directory</td>
</tr>
<tr>
<td>-m</td>
<td>Set probe module name</td>
</tr>
<tr>
<td>-x</td>
<td>Sets target() to PID</td>
</tr>
<tr>
<td>-c</td>
<td>Start probes, run command, exit when it finishes</td>
</tr>
<tr>
<td>-r</td>
<td>Cross-compile to kernel RELEASE</td>
</tr>
</tbody>
</table>

* See `stap(5)` man page for complete list and details
Cross Instrumentation

- Production environment will not have development & debuginfo packages. So how to run systemtap there?

- All kernel development & debuginfo packages can be installed on a single host machine.

- On host machine below command will provide kernel module (e.g. file_op.ko)
  
  ```bash
  stap -r `uname -r` file_op.stp -m file_op -p4
  ```

- Target system only one RPM needs to be installed i.e. systemtap-runtime.

- On target systems run
  
  ```bash
  staprun <kernel module>
  ```
Required Privileges

✓ Running stap and staprun requires elevated privileges to the system

✓ To allow ordinary users to run SystemTap without root access, add them to both of these user groups

1. stapdev
   Members of this group can use stap to run SystemTap scripts, or staprun to run SystemTap instrumentation modules.

2. stapusr
   Members of this group can only use staprun to run SystemTap instrumentation modules
SystemTap Language

- Probes & Probe Aliases – function entry & exit, source line
- kernel address, timer, begin/end
- Wildcarding
- Functions
- Types – string, 64-bit long, associative array, aggregation
- Comparison – if else & ternary operators
- Looping - while, for, foreach
- Usual binary & numeric operators
- String manipulation – sprint, sprintf, . & .= operators
- Output – log, print, printf
- Target variables – accessible with ‘$’ prefix
- Embedded C – raw C code, not covered by safety checks
Introduction of Tapsets

- Probe set that encapsulates kernel subsystem knowledge. Defines probes, data, auxiliary functions.

- Abstracts away subsystem implementation details.

- Probes are usable and extendable by other scripts.

- Tested and packaged with SystemTap.

- Located in either:
  - /usr/local/share/systemtap/tapset if installed from source
  - /usr/share/systemtap/tapset if installed from rpm
Example of Tapsets

VFS tapset

```c
probe generic.fop.open = kernel.function("generic_file_open")
{
    dev = __file_dev($filp)
    devname = __find_bdevname(dev, __file_bdev($filp))
    ino = $inode->i_ino
    file = $filp

    filename = __file_filename($filp)
    flag = $filp->f_flags
    size = $inode->i_size

    name = "generic_file_open"
    argstr = sprintf("%d, %d, %s", $inode->i_ino, $filp->f_flags, filename)
}
probe generic.fop.open.return = kernel.function("generic_file_open").return
{
    name = "generic_file_open"
    retstr = sprintf("%d", $return)
}
```
Sample Tapset Routines:

- AfsLockInfo
- GetVFid
- PrintVcache
- PrintDcache
- ...

```c
function PrintVcache:long (vcache:long)
{
    __fvcache = @cast(vcache,"vcache","kernel:openafs")->f
    __afsllock = @cast(vcache,"vcache","kernel:openafs")->lock

    fcache_info = Getfvcache(__fvcache)
    printf("\t\tVcache Information [%p] : \n",vcache);
    printf("\t\tFcache Information [%p] : [%s]\n",__fvcache,fcache_info
    printf("\t LockInformation : \n %s\n",AfsLockInfo(__afsllock))
}
```
Tapset Routines

```c
function PrintDcache:string (dcache:long)
{
  __lock = @cast(dcache,"dcache","kernel:openafs")\rightarrow lock
  __tlock = @cast(dcache,"dcache","kernel:openafs")\rightarrow tlock
  __mflock = @cast(dcache,"dcache","kernel:openafs")\rightarrow mflock
  validPos = @cast(dcache,"dcache","kernel:openafs")\rightarrow validPos
  index = @cast(dcache,"dcache","kernel:openafs")\rightarrow index
  refCount = @cast(dcache,"dcache","kernel:openafs")\rightarrow refCount
  dflags = @cast(dcache,"dcache","kernel:openafs")\rightarrow dflags
  mflags = @cast(dcache,"dcache","kernel:openafs")\rightarrow mflags
  __fcache = @cast(dcache,"dcache","kernel:openafs")\rightarrow f
  fcache_str = GetFcache(__fcache)
  lock_str = AfsLockInfo(__lock)
  tlock_str = AfsLockInfo(__tlock)
  mflock_str = AfsLockInfo(__mflock)

Lock [%s] Tlock [%s] mflock [%s]
\n",dcache,index,validPos,refCount,dflags,mflags,fcache_str,lock_str,tlock_str,mflock_str)
}
```
Tapsets for OpenAFS (cont...)  

Tapset Routines

```c
probe openafs.aops.writepage = module("openafs").function("afs_linux_writepage")
{

    __page = $pp
    dev = __page_dev(__page)
    ino = __page_ino(__page)
    for_reclaim = "N/A"
    for_kupdate = "N/A"

    if (@defined($wbp)) {
        for_reclaim = $wbc->for_reclaim
        for_kupdate = $wbc->for_kupdate
    }

    __inode = __address_inode(__page)
    __vcache = VTOAFS(__inode)

    name = "openafs.aop.writepage"
    page_index = $pp->index

}

probe openafs.aop.writepage.return = module("openafs").function("afs_linux_writepage").return
{
    name = "openafs.aop.writepage.return"
    retstr = sprintf("Return Value : %d",$return)
}
```
SystemTap Usage in AFS

- Defect Analysis and simulation
- Defect Testing
- Fault Injection
- Performance
- Tapset
**Problem Statement:** User application returned EIO error during msync operation.

**Initial Analysis:** Need to find which AFS function is failing & how EIO is returned back to a application.

**STEP 1:** Find which AFS function is failing
- **Script**

```c
global Agg
global RetAgg
probe begin {
    printf("Started probe\n")
}
probe module("openafs").function("+").call {
    if (tid() == target()) {
        Agg[probefunc()]++;
    }
}
probe module("openafs").function("-").return {
    if ((tid() == target()) & & @defined(Sreturn)) {
        RetAgg[probefunc()] << Sreturn
    }
}
probe end {
    foreach (Iter in Agg) {
        printf("Function Called %s : Count %d\n", Iter, Agg[Iter])
    }
    foreach (Iter1 in RetAgg) {
        printf("Func %s::", Iter1)
        printf($hist_linear(RetAgg[Iter1], 0, 10, 1))
    }
}
```
Case Study-1 (cont…)

Statistics of Functions Return Value

- Output

```
Func afs_EvalFakeStat::value |----------------------------------------------- count
  0 |.................................................. 28
  1 |
  2 |

Func afs_linux_permission::value |----------------------------------------------- count
  0 |.................................................. 18
  1 |
  2 |

Func afs_linux_writepage::value |----------------------------------------------- count
  0 |.................................................. 34086
  1 |
  2 |
  ~ |
  9 |
 10 |
>10 |.................................................. 9141
```
STEP 2: Stack Trace of the failing AFS function

- **Script**

```c
probe openafs.aop.writepage.return {
    if ((tid()) == target()) && @defined($return) && $return) {
        printf("%s <- %s\n", name, retstr)
        print_backtrace()
    }
}
```

- **Output**

```
Returning from: 0xfffffffffa04b23b0 : afs_linux_writepage+0x17/0x40 [kernel]
Returning to : 0xffffffff8112caa7 : __writepage+0x17/0x40 [kernel]
0xffffffff8112cbb9 : write_cache_pages+0x1c9/0x4b0 [kernel]
0xffffffff8112c0c4 : generic_writepages+0x24/0x30 [kernel]
0xffffffff8112e105 : do_writepages+0x35/0x40 [kernel]
0xffffffff8111a4bb : __filemap_fdatawrite_range+0x5b/0x60 [kernel]
0xffffffff8111a51a : filemap_write_and_wait_range+0x5a/0x90 [kernel]
0xffffffff8111b0ae : vfs_fsync_range+0x7e/0xe0 [kernel]
0xffffffff8111b1add : vfs_fsync+0x1d/0x20 [kernel]
0xffffffff8111c951 : sys_msync+0x151/0x1e0 [kernel]
0xffffffff8100b072 : system_call_fastpath+0x16/0x1b [kernel]
```
STEP 3: Which function is returning EIO error

- **Script**

```c
probe kernel.function("do_writepages").return {
    if ((tid() == target()) && @defined($return))
        printf("%s Returned %d\n", probefunc(), $return)
}

probe kernel.function("__filemap_fdatawrite_range").return {
    if ((tid() == target()) && @defined($return))
        printf("%s Returned %d\n", probefunc(), $return)
}

probe kernel.function("filemap_write_and_wait_range").return {
    if ((tid() == target()) && @defined($return))
        printf("%s Returned %d\n", probefunc(), $return)
}
```

- **Output**

```
wait_on_page_writeback_range Returned -5
filemap_write_and_wait_range Returned -5
write_cache_pages Returned 0
generic_writepages Returned 0
do_writepages Returned 0
__filemap_fdatawrite_range Returned 0
__filemap_fdatawrite_range Returned 0
wait_on_page_writeback_range Returned 0
vfs_fsync_range Returned -5
vfs_fsync Returned -5
sys_msync Returned -5
```
**Problem Statement**: Dcache readlock leak in case “afs_dir_GetVerifiedBlob” fails inside “afs_linux readdir”.

- Mainly afs_PutDcache was called without releasing a readlock

**Simulation**: To simulate this defect “afs_dir_GetVerifiedBlob” should fail. For this we used SystemTap as a fault injection mechanism.

- Script

```bash
probe module("openafs").function("afs_dir_GetVerifiedBlob").return {
    printf("Ret value [%d] going to change to non-zero\n",$return);
    $return = -1;
    printf("Ret value changed to[%d] \n",$return);
}
```
With above fault-injection we were able to test the fix.

To verify that there is no such lock leak in other places, we added probe during return of “afs_PutDCache” which checks for lock leak.

```c
probe module("openafs").function("afs_PutDCache").return
{
  __dcache = $adc
  __lock = &@cast(__dcache,"dcache","kernel:openafs")->lock
  __mlock = &@cast(__dcache,"dcache","kernel:openafs")->mlock
  __tlock = &@cast(__dcache,"dcache","kernel:openafs")->tlock
  if ((CheckForDcacheLockLeak(__lock,__dcache) ||
       CheckForDcacheLockLeak(__mlock,__dcache) ||
       CheckForDcacheLockLeak(__tlock,__dcache)))
  {
    printf("ALERT::Lock Leak Detected\n")
    print_backtrace()
  }
}
```
SystemTap can be used to gather performance statistics

- **Script**

```c
#include <stdio.h>

int main()
{
    int i, j;
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
            printf("%d %d\n", i, j);
    return 0;
}
```

- **Output**

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Min</th>
<th>Avg</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>afs_GetDCache</td>
<td>271</td>
<td>0</td>
<td>200</td>
<td>52396</td>
</tr>
</tbody>
</table>
References

- [http://sourceware.org/systemtap/examples](http://sourceware.org/systemtap/examples)


- [http://sourceware.org/systemtap/wiki/](http://sourceware.org/systemtap/wiki/)
Thank You