Improving the Approach to Linux Performance Analysis

An analyst point of view

Jose Santos
IBM's Linux Technology Center
`jrs@us.ibm.com`

Guanglei Li
IBM's China Development Lab
`guanglei@cn.ibm.com`

Abstract

In-depth Linux kernel performance analysis and debugging historically has been a focus which required resident kernel hackers. This effort not only required deep kernel knowledge to analyze the code, but also required programming skills in order to modify the code, re-build the kernel, re-boot and extract and format the information from the system.

A variety of powerful Linux system tools are emerging which provide significantly more flexibility for the analyst and the system owner. This paper highlights an example set of performance problems which are often seen on a system, and focuses on the methodology, approach, and steps that can be used to address each problem.

Examples of a set of pre-defined SystemTAP "tapsets" are provided which make it easier for the non-programmer to extract information about kernel events from a running system, effectively tracing some of the kernel behavior. Some examples of performance problems include I/O problems, workload scalability issues, and efficient system utilization. Available tools such as SystemTAP, kprobes, oprofile, and trace tools are compared and contrasted to provide the system owner with real-life examples which can be used on their systems today.

1 Introduction

As the complexity of the Linux kernel increases, so does the complexity of the problems that impact performance on a production system. In addition, the hardware systems on which Linux runs today is also increasing in complexity, scale, and size. Customers are also running more robust and challenging workloads in production environments and are expecting more reliability, stability, and performance from the underlying operating system and hardware platform. The end result of all this complexity is that new performance barriers continue to emerge which in many cases are increasingly difficult to analyze and understand.

In the past, many of these problems required the expertise of kernel developers to create specialized one-off tools that were specific to the problem at hand and are of little use to other problems. While this remains one of the more powerful ways to do analysis of the kernel, it is limited to people with deep understanding of the kernel subsystems. To some extent, this prevents many users who are running on hardware systems and software environments to which developers may not have access, from doing their own initial performance problem determination and assessments.

With the continued evolution of tools like Oprofile and the development of new tools such as SystemTap, there are now easier and more consistent ways for users and developers to tap into the performance problems present in specialized environment. By making these tools less complicated to use, it allows for more users to provide better information to developers which in turn enhances communications in the community and functions as a learning tool for the aspiring kernel hacker.

2 Identifying performance problems

2.1 Types of problems

Performance related problems can be split into two general categories:

1. CPU bound problems
2. Non-CPU bound problems

CPU bound problems are caused when System Under Test (SUT) CPU resources are completely utilized, which means that the system will not be able to process more information faster. Due to their nature, CPU bound problems are generally easier to identify since there are various tools available to determine the utilization of these resource. For these kinds of problems, a profiler is typically the best tool for the job.

Problems that are not CPU bound are usually caused by a lack of some other resource on the SUT or other problems in the code that do not allow for the various resources of the system be fully utilized. These are a bit trickier to figure out because there are numerous resources that can be utilized in any given environment and some investigation is required in order to identify what resource is causing any given performance problem. A typical tool used to do initial forensics of this problem is a system trace.

2.2 Basic analysis

Before the first email is sent or a kernel recompiled, there is some initial data that needs to be gathered to determine the starting point of the analysis process. Programs such as vmstat and the tools from the sysstat package provide valuable data at this stage. The data gathered by these tools can give insights into possible causes of the problems and act as a stating point for the analysis process.

Another important information that needs to be gathered is accurate and verbose system configuration information as some problems can be traced back to hardware or known device driver issues. Its also important to know the limitations of the hardware before assessing that there is a performance problem in the first place.

Once this information is gathered, analysis of vmstat output can provide usage activites from memory, processes, CPU components that can narrow down the scope of the analysis process. If the system show very low activity when more activity is expected the use of other programs like iostat or sar can be use obtain a system activity report. While iostat concentrates on disk IO activity, sar can get information from various different components of the running system, including detail interrupt information, networking cpu activity and more.

There are several books available on the market that discuses some of these techniques in more detail and are a valuable reference for analysis work.

3 Oprofile

Oprofile is a system-wide profiler that can utilize the performance counters available in a variety of processors in order to create summary reports of the activities that happes within the system. One of Oprofiles greatest strengths is its simplicity. A basic session consists of three commands:

1. `$opcontrol --setup --vmlinux=/boot/vmlinux`
2. `$opcontrol --start`
3. `$opreport -l -p /lib/modules/uname -r`

This generates the output as shown in Figure 1.

By default Oprofile is configured to use the CPU cycles performance counter as the trigger for a profile event. Since the the tool relies on performance counters as the trigger for collecting data, the tool is most suited for analyzing CPU bound problems.

While more advanced uses of Oprofile are beyond the scope of this paper[1] the basic information obtained though opreport can be examined to determine the cause of a performance issue by viewing the frequency that a kernel or user function spends on a given performance counter event. A good guideline to follow is that if the kernel is spending more than 5% of its time in a single function, then this function is a good candidate for further analysis.

Once a kernel function that causes the performance abnormality has been identified, further analysis needs to be done at the source code level to figure out the root cause of the problem. There is a tool on the Oprofile package called opannotate can help determine where within a function Oprofile is receiving the greatest hits. One drawback is that opannotate does little to help figure out problems that are making such a function show high utilization in Oprofile reports. One example of this type of scenario is when large amount of time is spent in spinlock code. This is typically not a problem with the spinlock code itself, but rather the code that calls a particular spinlock. The tool does however provide a good amount of information as a starting point for the next tool described on the paper.

[1]More examples of the capabilities of Oprofile can be found at `http://oprofile.sourceforge.net/examples/`

4 SystemTap

Inspired by the work of Sun's DTrace, engineers from Red Hat with the help from other companies created a new tool called SystemTap. This tool, while still in its infancy, provides a wealth of opportunity not only to new kernel programmers, but also for the veteran kernel hacker. SystemTap provides a simplistic language that is built to talk to a live kernel. This simple language provides the user with a way to interface with the kernel without the complexities details that similar functionality using kprobes and C code requires. This allows for the creation of very detailed tools with minimal amounts of code and helps the analyst focus on the core problem. This is a big contrast to using the dynamic probe infrastructure available in the Linux kernel today as these require the user to design a fully functional kernel module. While this approach can provide some benefits in speed as well as the flexibility to instrument the kernel, it does so at the expense simplicity.

SystemTap relies on predefined functions called tapsets in order to extract certain information from the kernel. These tapsets are designed to provide a set of predefined functions

```
CPU: P4 / Xeon with 2 hyper-threads, speed 3002.82 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped)
with a unit mask of 0x01 (mandatory) count 100000
samples  %        app name                 symbol name
2270717  73.6201  vmlinux-2.6.5-7.244-smp  .text.lock.rwsem_spinlock
218144    7.0726  vmlinux-2.6.5-7.244-smp  __down_read
201530    6.5339  vmlinux-2.6.5-7.244-smp  __up_read
18968     0.6150  oprofile                 (no symbols)
11730     0.3803  tkrlog                   tkfind
10174     0.3299  libstdc++.so.5.0.6       std::string::compare(std::string const&) const
```

Figure 1: Sample Oprofile output

that make it simple to gather information from the kernel. In situations where the tapsets do not provide sufficient functionality needed to analyze a problem, extra functionality can be added by embedding C code into the script. While this provides experienced users with the ability to customize their instrumentation to their need, the C code will run as-is, and if the user is not cautious, a system crash may occur. For this reason, embedded C code can only be run in 'GURU Mode' which restricts the use of SystemTap to users with privileged access. Even with this restriction, the user should take care of the fact that badly generated code may cause his system to fail. Since SystemTap relies on kprobes for inserting probe points into the kernel, it is also important to mention that kprobe inserted in certain areas may lead to a system crash. While a lot of these functions have been black-listed to prevent users from inserting probes in dangerous locations, some probes can, under some circumstances, cause system stability issues. As SystemTap matures, the tools will be more robust and take precaution to not insert probe points into such places in the kernel.

One of the biggest advantages of SystemTap is the ability to export kernel information to the user. One of the traditional ways to export kernel information to the user is using printk and analyzing the output. While this works well for debugging functionality problems, printk is a costly operation and can change the perfor-

```
#!/usr/local/bin/stap
global trace

probe kernel.inline("idle_balance") {
     trace[backtrace()] <<< 1
}

probe begin {
     print("Starting IDLE backtace")
}

probe end {
     foreach( stack in trace ) {
          print("==================\n")
          printf("Count: %d\n",
               @count(trace[stack]))
          print_stack (stack)
     }
}
```

Figure 2: Backtrace accounting when entering the idle loop

mance characteristics of the problem at hand. Although a developer can always create a more complex code to store and export that data to user space, this is a level of complexity that not only requires deep kernel knowledge, but also takes valuable time away from the problem solving.

In the code sample shown in Figure 2, the goal is show how to analyze one of the most common performance problems that can be seen in the field, the inability to drive a system to full CPU capacity. There are many causes of not being able to drive a system to full capacity; examples includes IO limitations or semaphores restricting the scalability of the system. This script instruments the `idle_balance()` so that every time that the system is about to go into the idle loop, a backtrace of the sequence of events that caused us to go idle is shown. The backtrace is then use as a key for an array that increments each time the same backtrace is hit. The end result is a count of backtraces that can show the sequence of events that led the machine to go idle. With this sequence of events, tuning of the system or code changes may be utilize to fix the problem that prevents the system for doing more work.

In the previous example, the tapsets provided by SystemTap were sufficient to analyze the given problem. In code sample in Figure 3, embedded C code is use in order to create a report that shows how processes access memory across NUMA nodes. The purpose of this script is to assist the analysis of problems cause application accessing memory across NUMA nodes. This is typical in HPC (High Performance Computing) environments were lots of remote memory can cause huge stall in the processor reducing the performance of the application. To obtain memory access information from the running system, a probe is inserted into the `__handle_mm_fault()` in which the address of the page fault can be extracted. In order to translate kernel addresses to NUMA node information, the function `addr_to_node()` was created in embedded C code to fulfill this functionality. All the information is later inserted into arrays that use the pid number, write access, and numa nodes as keys for the counters.

5 Tracing

While tools such as SystemTap provide the means to instrument the kernel in new exciting ways, it still relies on the expertise of knowledgeable developers or analysts to come up with proper ways to use the tool in order to resolve a problem. Since the tool is so focused in its approach, it is not the best tool to use in situations where the problem has not been narrowed down to a specific component of the kernel. In these situations, getting a system trace is often one of the best tools for getting information that other tools such as Oprofile have failed to uncover. A system trace consist of predefined probe points called trace hooks that are inserted key places located within the kernel. These trace hooks are designed to give the user a detailed activity histogram of a running system. These system activity histograms can then be analyzed in user space using scripts that generate detail reports or using visualization tools that allow the user to view time spent on system activities.

One of the advantages of using this method of system analysis is that the work of determining where probe points should be inserted in the kernel has already done by the developers of the tool. This is very appealing tool for developers doing analysis in customer environments where the developer has restricted access to the production environment. With a trace tool, the developer simply needs to provide the customer with the right set of instructions for client to run the tool. The the data extracted from the system is then passed on to the developer for further analysis.

There are several tools currently in development that provide trace functionality to the Linux kernel. There are offerings like LTTng (Linux Trace Toolkit Next Generation) and LKST (Linux Kernel State Tracer) which require kernel patching but offer superior performance and as well as tools like LKET(Linux Kernel Event Trace) which are being implemented on top SystemTap for user convenience but at expense of some performance.

```
#!/usr/local/bin/stap -g
global execnames, page_faults, node_faults

function addr_to_node:long(addr:long)
%{
        int nid;
        int pfn = __pa(THIS->addr) >> PAGE_SHIFT;
        for_each_online_node(nid)
                if ( node_start_pfn(nid) <= pfn &&
                        pfn < (node_start_pfn(nid) +
                        NODE_DATA(nid)->node_spanned_pages) )
                {
                        THIS->__retvalue = nid;
                        break;
                }

%}

probe kernel.function("__handle_mm_fault") {
        execnames[pid()] = execname()
        page_faults [pid(), $write_access ? 1 : 0] ++
        node_faults [pid(), addr_to_node($address)] ++

}

function print_pf () {
        print ("            Execname\t     PID\tRead Faults\tWrite Faults\n")
        print ("====================\t========\t===========\t============\n")
        foreach (pid in execnames) {
                printf ("%20s\t%8d\t%11d\t%12d\t", execnames[pid], pid,
                        page_faults[pid,0], page_faults[pid,1])

                foreach ([pid2,node+] in node_faults) {
                        if (pid2 == pid)
                                printf ("Node[%d]=%d\t", node,
                                        node_faults[pid2, node])
                }
                print ("\n")

        }
}

probe begin {
  print ("Starting pagefault counters \n")
}

probe end {
  print ("Printing counters: \n")
  print_pf ()
  print ("Done\n")
}
```

Figure 3: Numa page fault accounting

5.1 LTTng—Linux Trace Toolkit Next Generation

LTTng is a replacement of the original LTT (Linux Trace Toolkit). It is an enhanced version of the existing LTT instrumentation and uses RelayFS to export the data to user space. It is designed to be fully reentrant, scalable, extensible, modular, precise, (declared to be around 100ns time accurate) and has low overhead, low system disturbance, and architecture independence.

LTTV (Linux Trace Toolkit Viewer) is a visualization tool that complements LTTng by performing analysis of the trace data generated by LTTng and showing the results in text or in a graphical display interface. It has a modular architecture based on plug-ins which means that you can use various text and graphical plugs to handle different trace data. Multiple plug-ins can interact with each other to further enhance the analysis capabilities.

LTTng also has a code generator named genevent which will parse user customized event descriptions and generate the necessary codes to record events in the kernel. With the added plug-in mechanism of LTTV, it makes it easier to trace and analyze a new event inside the kernel.

One of LTTng biggest weakness is that it requires kernel patches. This limits its use to environments were kernel recompiles and lost of downtime are allowed.

5.2 LKST—Linux Kernel State Tracer

LKST is another kind of kernel trace tool developed by Hitachi and Fujitsu. Like SystemTap, it enables developers to investigate problems in the Linux Kernel without stopping the machine. It will record kernel events such as process context switching, exception, memory allocation as trace data, and provides a log analyzer tool to do some post-processing work. Users can use LKST to analyze the errors happened inside a running kernel, and it could also be used to do the performance analyzing work. And what's more, it is possible to change dynamically which events will be recorded, so that developers can obtain information about the events which they concern only. It is also possible to change the handler associated with each event.

Like LTTng, LKST requires patches to the Kernel. Unlike SystemTap, LKST will add static hook check points into various locations of Linux Kernel, and the registered event handler will be executed if the user chooses to probe that event.

5.3 LKET—Linux Kernel Event Trace

LKET is an extension to the tapsets library available on SystemTap. It was born out of the necessity to gather trace information from environments were recompiling kernels is not allowed. Its goal is to utilize the dynamic probing capabilities provided through SystemTap to create a set of standard hooks that probe predefined places in the kernel. This allows both experienced kernel programmers, analysts and customers to gather important information that can be used as a starting point to analyze a performance problem in their system.

The LKET tapset is designed to only gather the trace hook events selected by the user. This allow the tool to be customize depending on the nature of the problem being analyzed. Once the data has been collected, it is then post-processed according to the need of the user. This provides a significant advantage over just running a simple SystemTap scripts since the data there is some what static. On the other

Hook Family	Hooks	Description
addevent.syscall	addevent.syscall.entry addevent.syscall.return	Entry and exit of System Call events
addevent.process	addevent.process.fork addevent.process.execve	Process Creation events
addevent.ioscheduler	addevent.ioscheduler.elv_next_request addevent.ioscheduler.elv_completed_request addevent.ioscheduler.elv_add_request	IO Scheduler activity events
addevent.tskdispatch	addevent.tskdispatch.ctxswitch addevent.tskdispatch.cpuidle	Task scheduling events
addevent.scsi	addevent.scsi.ioentry addevent.scsi.iodispatching addevent.scsi.iodone addevent.scsi.iocompleted	SCSI layer activity events

Table 1: Supported LKET trace hooks

hand, trace data can be process in various different ways to generate from simple to complex reports. Detailed information is necessary in order to create complex reports. That is why each event hook contains common data such as time stamp, processes ID information and CPU information as well as some data that is specific to the trace hook.

The trace hook event utilizes the aliasing functionality of SystemTap. This allows for grouping of event base component of the kernel being probed. Different aliases(addevent.eventName) are defined to trace different kinds of events. As of this writing, Table 1 show the current event hooks provided by LKET. More event hooks are scheduled to be implemented as development continues.

Simplicity of use is one of the design goals of LKET and SystemTap plays a big role in achieving this goal. In order to enable all the trace hooks available in LKET, a simple SystemTap script containing "`probe addevent.* { }`" is all that is needed. If a more selected set of trace hooks is desired, one can add individual trace hooks or trace hook families to as described in Table 1.

To show an example of using LKET to trace system calls of "updatedb" and do simple post-processing we first SystemTap's LKET tapset to generate the trace data:

```
$ stap -e "probe addevent.syscall {}" \
-c "updatedb" -D ASCII_TRACE \
-I LKET_TAPSETS > probe.out
```

The generated trace data looks like:

```
1|1143485073|422541|8378|8368|8378|0|sys_mmap
2|1143485073|422550|8378|8368|8378|0|sys_mmap
1|1143485073|422556|8378|8368|8378|0|sys_close
2|1143485073|422562|8378|8368|8378|0|sys_close
1|1143485073|422602|8378|8368|8378|0|sys_read
2|1143485073|422611|8378|8368|8378|0|sys_read
```

To make this example simpler, we let LKET log trace data in ASCII format instead of the default binary format. The ASCII trace format uses "|" as the delimiter and its described in Figure 5.3. The HookID's for the system call trace hooks are "1" for syscall entry and "2" for syscall return. The syscall hooks have a single "Hook data" field which in this case is the name of the syscall.

After the data has been gathered, analysis of the data can be delegated to scripts like the one

HookID	tv.sec	tv.u_sec	pid	ppid	tid	cpuid	Hook data	...

Figure 4: ASCII trace format

```
#!/bin/awk -f
BEGIN {
  FS="|";
}

{
  if($1 == 1) {
    start[$8,$6] = $2*1000 +$3/1000
  } else {
    stop[$8,$6] = $2*1000 +$3/1000
    elapsed[$8]=stop[$8,$6]-start[$8,$6]
    if(elapsed[$8] > max[$8])
      max[$8]=elapsed[$8]
    cnt[$8]++
    total[$8] += elapsed[$8]
  }
}

END {
  printf "%-22s%-12s%-12s%-6s%-12s\n",
     "name","max","average","count","total"
  for(x in cnt) {
    printf "%-22s%-12s%-12s%-6s%-12s\n", x,
     max[x],total[x]/cnt[x],cnt[x],total[x]
  }
}
```

Figure 5: AWK script for post-processing System Call

in Figure 5.3 to generate a report of the top 10 most costly system call during the execution of updatedb.

```
$ awk -f post-processing.awk probe.out | sort
-nr -k 5 | head -n 10
```

The output looks like:

```
sys_getdents64 1.92896 0.021046 29728 625.651
sys_fstat64    5.03613 0.007144 43785 312.803
compat_sys     2.17603 0.020722 14600 302.534
  _fcntl64
sys_close      3.37598 0.008320 29213 243.061
sys_fchdir     1.75    0.007192 29184 209.896
sys_fcntl      2.16382 0.007017 14600 102.444
sys_write      0.24512 0.032507 556   18.074
sys_rename     1.80811 1.80811  1     1.80811
sys_brk        0.26709 0.024827 65    1.61377
sys32_execve   0.43408 0.357544 2     0.71509
```

From left to right, the data are syscall name, max time, average time, number of times being called,, and total time of each system call. This just illustrates one of the ways to analyze the trace output data. Further analysis of the data can be done by customizing the scripts to generate more complex results without the need to gather the data again.

5.3.1 LKET Limitations

One of the current limitations of LKET is that the overhead of kprobes added to the overhead of the hook it self can cause some workload to slow down significantly. New improvements to the Kprobe subsystem plus enhancements to SystemTap such as static probes and the binary tracing mechanism will solve most of the performance issues though.

Another short coming of the LKET tool is that since the project has been in active development for a short period of time, there lack of availability of good post-processing tools. The development team is working to create a post-processing infrastructure for LKET so that the tool can be more useful for first time users.

6 Conclusion

Gone are the days were developers needed to resort to hack in order to analyze kernel performance. Tools like Oprofile and SystemTap are opening the doors to people new to Linux kernel analysis. While Oprofile has shown its powerful usefulness for the past couple of years, SystemTap shows its flexibility by providing an infrastructure were new tools can be developed.

7 Legal Statement

References

[1] Frank Ch. Eigler, Systemtap tutorial, March 27, 2006, `http://sourceware.org/systemtap/tutorial/`

[2] Steve Best, Linux Debugging and Performance Tuning, Prentice Hall, October 14, 2005

[3] Phillip G. Ezolt, Optimizing Linux Performance, Prentice Hall, 2005

[4] SystemTap Mailing List Archives, `http://sources.redhat.com/ml/systemtap/`

[5] SystemTap Web Site, `http://sourceware.org/systemtap`

[6] Oprofile Website, `http://oprofile.sourceforge.net`

Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada