Recognizing and Resolving CPU Load in the Kernel

Fluctuating Processors

Most CPU load monitors, such as *top* and *xosview* access the Linux kernel's */proc* filesystem to try and determine the correct value. However, this interface can deliver incorrect values if certain conditions occur. We explain why and the patch described in this article resolves this issue.

BY ARNE WIEBALCK, TIMM M. STEINBECK AND VOLKER LINDENSTRUTH

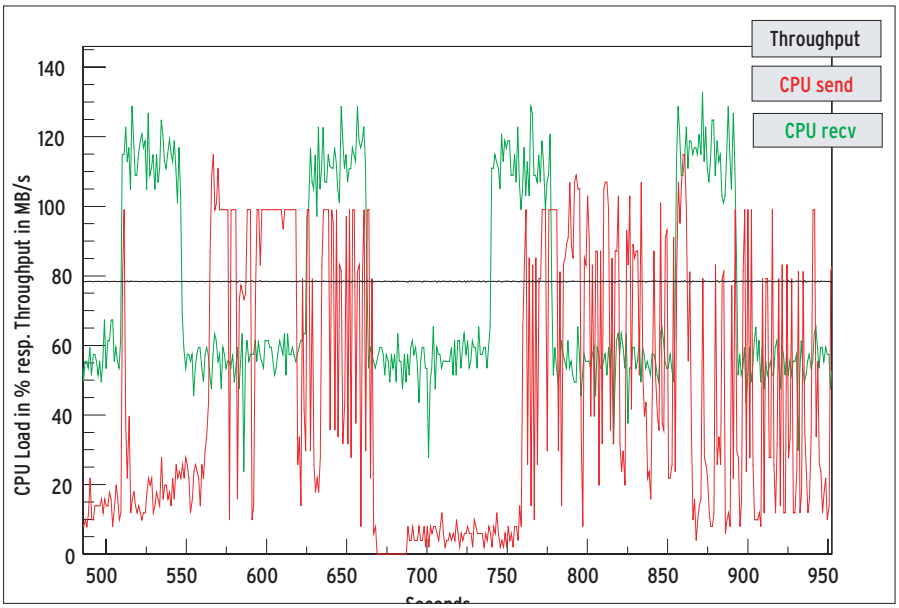

Figure 1: Inexplicable fluctuations of processor load and throughput while transferring data across the wire between two SMP systems

The Department of Technical Computer Science at the University of Heidelberg develops data parsing systems for planned large scale applied research into elementary particles and heavy ions, where Linux clusters can comprise 1000 nodes.

Within this framework one working group has been looking into efficient network communication mechanisms, to avoid processor load during data transfer as far as possible. The group was particularly interested in minimal network transfer overhead using unmodified network card drivers. With this aim in mind, the authors of this article developed a small footprint kernel module capable of transferring data from a program across a network without using a protocol like TCP/IP.

Inexplicable Fluctuations

This module was intended to control the CPU load for multiple data transfer rates. But our measurements showed an interesting phenomenon on a dual processor system: despite constant transfer rates system load tended to deviate between 0 per cent and an upper limit proportional to the transfer rate. This phenomenon affected both the sender and the receiver of the data.

These fluctuations occurred periodically within a timespace in the region of several hundred seconds. Figure 1 shows an example of the phenomenon. This effect only occurred when the *usleep* function was called to limit the transfer rate, but not if the total network bandwidth was used.

A small test program running outside of the kernel and using *memcpy* to copy data to memory showed a similar reaction. The rate at which this program copies is also configurable via *usleep*. Calling the same program on a system with a single processor kernel showed a minimal CPU load (slightly above 0 per cent) despite copying 80 Mbyte/sec…

Where's Harvey?

This led us to suspect that our program's processes might have escaped monitoring by, and might be partly invisible to, the kernel. To verify this hypothesis we

The Scheduler

Linux allows processes to run more or less simultaneously on a CPU by dividing the processing time into timeslots. The decision as to which process receives which timeslot when is made by the scheduler. The scheduler applies a scheduling algorithm to select an executable process from a list and allots the process one timeslots worth of CPU cycles.

A timeslot is normally 10 ms for Linux. The length of the timeslot is important to a system's performance: if it is too short, overhead caused by the scheduler making a decision and switching from one process (context) to another increases. If the timeslot is too long, the processes will no longer appear to be running simultaneously.

When a process terminates or needs to wait for an external event, it no longer requires the remainder of its timeslot, and thus releases it prematurely, by calling the *usleep* function, for example.

In this case, an exceptional call to the scheduler occurs and the scheduler passes the remainder of the timeslot to the next regular process. Independently of this, the scheduler is again called after the timeslot has elapsed.

wrote a program that works and sleeps alternately in an infinite loop.

On account of its ability to hide from the kernel's standard process accounting functionality, we called the program "Harvey" (see Listing 1). The infinite loop that starts in line 13, contains the two functional blocks that perform calculations and sleep. While working Harvey repeatedly calls *gettimeofday* to ascertain the elapsed time for a block.

When the time specified in *RUNTIME* elapses, Harvey calls *usleep(0)* to release the remainder of its timeslot (see "The Scheduler" boxout). Figure 2 shows a screenshot of *top*, with Harvey running. Obviously Harvey is not creating any load on the system and thus demonstrates the same behavior as the other programs.

As it is improbable that a process that spends 90 per cent of its runtime in a loop will create any noticeable CPU load, we used the CPUmeter program shown as Listing 2 to measure the load. The program's infinite while loop calls the *get_iterations* function to measure the number of loops per second. If the program is launched with lowest possible priority, the scheduler assigns it less CPU cycles, assuming that at least one other process is running on the system.

Thus, the number of iterations performed per second is a measure of the genuine load on the processor: the less iterations, the more load there is. Looking at CPUmeter shows that Harvey creates a load of about 90 per cent on our system – as one would expect from viewing the listings.

Watching the Kernel

The interface used by *top* and co. */proc/stat* yields the CPU load measured in timeslots only. Thus, one would suspect that the kernel also applies the same level of granularity. In the kernel sources, *fs/proc/proc_misc.c* is responsible for outputting the */proc/stat* pseudo-file, as specified in [1].

The structure used at this point, *kstat*, contains the data on the timeslots used by each CPU. Each entry is placed in one of three categories: *user*, *nice*, and *sys*. The global counter, *jiffies*, is used to output non-utilized timeslots, that are either used up by the kernel's idle task, or not used at all.

The structure elements are written to the *kernel/timer.c* file by the *update_process_times* function, that calls timer interrupt routines every time a timeslot elapses. Update_process_times checks which process is currently active and, if it is not the idle process, increments one of the three counters: *user*, *nice*, or *sys*. At the same time it decides which process to attribute the elapsed timeslot to. A timeslot is always attributed to an active process at the point where it *elapses*. If there is no active process at this point, the kernel marks the timeslot as unused.

Listing 1: *Harvey*

```
#include <unistd.h>
#include <sys/time.h>

/* runtime in microseconds */
#define RUNTIME 9000

int main( int argc, char** argv )
    {
    unsigned long n=0;
    unsigned long t;
    struct timeval s, e;

    while ( 1 )
        {

        /* work */
        gettimeofday( &s, NULL );
        do
            {
            n++;
            gettimeofday( &e,
NULL );
            t = (e.tv_sec-
s.tv_sec)*1000000
                +(e.tv_usec-
s.tv_usec);
            }
        while ( t < RUNTIME );

        /* sleep */
        usleep( 0 );
        }
    }
```

Table 1: Context Change Latency figures

Process size	Latency unpatched	Latency patched
0 kByte	0.89 µs	0.97 µs
4 kByte	1.02 µs	1.11 µs
16 kByte	4.31 µs	4.44 µs

Based on this background knowledge, we can now better understand the behavior shown by Harvey and the other programs. Dropping the timeslot by calling *usleep* means that there is no active process in *update_process_times*. Thus the routine records the timeslot as unused, although in fact it was partly used.

Phenomenon Observed Previously

A Google search for this problem showed that this issue is not unknown. Early in 2000 Jan Astalos produced a patch for

Listing 2: *CPUmeter*

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

#define TIME 1000000

unsigned long long
get_iterations( unsigned long
t_musec )
    {
    unsigned long long n = 0;
    struct timeval s, e;
    unsigned long tdiff;
    gettimeofday( &s, NULL );
    while ( 1 )
        {
        n++;
        gettimeofday( &e, NULL );
        tdiff = (e.tv_sec-
s.tv_sec)*1000000+(e.tv_usec-
s.tv_usec);
        if ( tdiff >= t_musec )
            break;
        }
    return n;
    }

int main( int argc, char** argv )
    {
    unsigned long long cur;

    while ( 1 )
        {
        cur = get_iterations(
TIME );
        printf( "     %20Lu
iter./s\n", cur );
        }
    return 0;
    }
```

single-processor systems running the 2.2.14 kernel [2]. This patch uses the timestamp counter (TSC) available on more modern processors to count the number of CPU cycles consumed by each process.

When asked for a later, SMP capable version, Jan sent us a version for 2.4.0, that was the basis for the port to the later kernel we were using. We additionally added the ability to collate the number of CPU cycles consumed either globally or by CPU, and to detect cycles used by interrupts and soft IRQs to the patch. The latter was particularly important for the kind of network measurements our department needed to perform.

In contrast to the process accounting performed by the standard kernel, that occurs only at the end of each timeslot, as previously discussed, we called the function shown as Listing 3, *update_process_cycles*, shortly *before* the scheduler assigns a new process to the CPU. This is why the function is also called in the *kernel/sched.c* file.

In lines 3 through 5 *update_process_cycles* first ascertains the current process, the active CPU and the current value of the TSC. Line 7 updates the array, *cycles*, added to the process structure. The *last_cycles* array used for this purpose contains the value of the TSC for the CPU in question at the point when *update_process_cycles* was last called (line 10).

Lines 8 and 9 set the global counter that contains the used cycles per CPU. This only happens if the current process is not the idle process, which has a process ID (PID) of zero.

In a similar fashion kernel functions count the number of CPU cycles used to handle interrupts and soft IRQs.

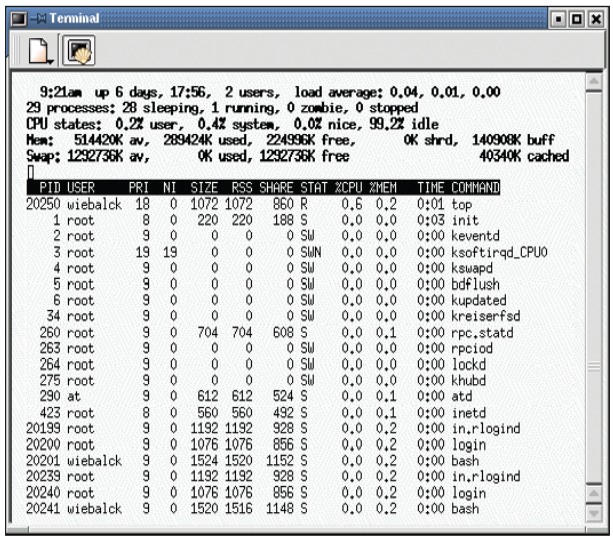

Figure 2: *top* with Harvey running

Output via */proc/stat*

The values ascertained here are output via standard process accounting facilities, that is, the */proc* pseudo-filesystem for the kernel. */proc/stat* lists the number of processes, interrupts, and soft IRQs used, and not used, as well as the total number of cancelled cycles. Each of these values are displayed as a total for all CPUs and for each individual CPU.

The CPU cycles used by each process is displayed in */proc/*PID*/stat* – again on a per CPU basis and in total. */proc/interrupts_cycles* contains a more detailed breakdown of the cycles used by interrupts and soft IRQs. Listing 4 shows the output for the pseudo-file: the lines starting with numbers contain the cycles used by the corresponding interrupts, and the last four lines show the four different soft IRQ types.

The system load values caused by our program were ascertained for the patched kernel, and correlate to the values indicated by the CPUmeter program (and make sense in programming terms).

Our example in Figure 3 shows a comparison between the load generated by Harvey as measured for the standard kernel and the patched version. As you can see, Harvey successfully hides from the normal kernel, whereas the patched

Listing 3: Read CPU cycle statistics for the kernel

```
void update_process_cycles(void)
{
        struct task_struct *p = current;
        int cpu = smp_processor_id();
        cycles_t t = get_cycles();

        p->cycles[cpu] += t - last_cycles[cpu];
        if ( p->pid )
            kstat.used_cycles[cpu] += t - last_cycles[cpu];
        last_cycles[cpu] = t;
}
```

Listing 4: */proc/interrupts_cycles*

```
                CPU0            CPU1
   0:    20242854219     16586735080    IO-APIC-edge  timer
   1:        1636655         1225320    IO-APIC-edge  keyboard
   2:              0               0           XT-PIC  cascade
  10:              0               0   IO-APIC-level  usb-ohci
  14:      251664916       263921645    IO-APIC-edge  ide0
  23:     5601393923      5431463426   IO-APIC-level  eth0
HI_SOFTIRQ:  3924680280  3191840981
NET_TX_SOFTIRQ:  1115199064  1274524965
NET_RX_SOFTIRQ:  5157803275  4835883365
TASKLET_SOFTIRQ:  90516230  90298686
```

INFO

[1] Linux Cross-Reference: *http://lxr.linux.no/*

[2] 2.2.14 Precise Accounting Patch Posting: *http://www.beowulf.org/pipermail/beowulf/2000-February/008415.html*

[3] Lmbench homepage: *http://www.bitmover.com/lmbench*

[4] Precise Accounting Patch: *http://www.ti.uni-hd.de/HLT/documentation/software-and-documentation.html#kernel*

kernel knows exactly what he is up to. The patched kernel also displays correct values for the load generated by programs running on SMP systems.

To discover the effect this patch had on the scheduler's performance, we used the *LMbench* [3] benchmark suite on both the patched and unpatched single-processor kernel. The values the benchmark returned for context change latency are shown in Table 1. More exact accounting figures cost about a 10 per cent increase in latency for context changes.

Conclusion

The process accounting implementation provided by the Linux kernel can return incorrect values for the system load under specific circumstances. This is caused by the timeslot based granularity that the kernel applies to measure CPU cycle use.

The two test programs discussed here, Harvey and CPUmeter, paint a clear picture of this issue. The kernel patch discussed in this article, which is available at [4], implements a process accounting method based on the CPUs timestamp counter registers, and returned reliable results in our lab environment for the system load. ■

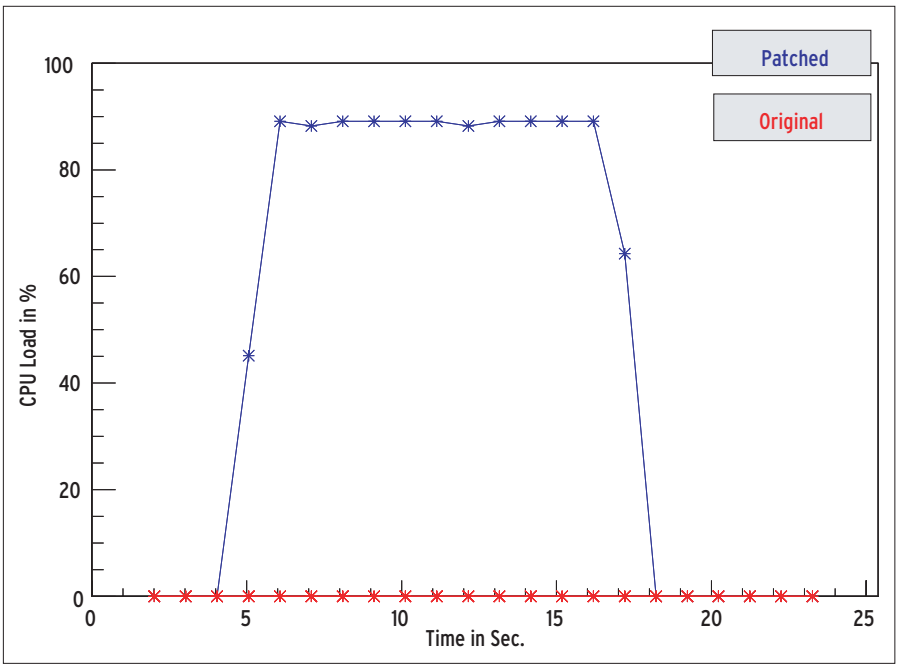

Figure 3: Harvey exposed