FERMILAB-CONF-07-115-E

Microsecond Delays on Non-Real Time Operating Systems

R. Angstadt, J. Estrada, H.T. Diehl, B. Flaugher, M. Johnson,
Fermi National Accelerator Laboratory[1]
Batavia, IL. 60510, USA

Abstract

We have developed microsecond timing and profiling software that runs on standard Windows[1] and Linux based operating systems. This software is orders of magnitudes better than most of the standard native functions in wide use.

Our software libraries calibrate RDTSC in microseconds or seconds to provide two different types of delays: a "Guaranteed Minimum" and a precision "Long Delay", which releases to the kernel. Both return profiling information of the actual delay.

I. Introduction

Pentium II[2] and subsequent version processors as well as some AMD[3] compatible CPU's have a 64 bit register that counts the number of CPU clock cycles, or ticks of the CPU clock since boot time. This counter can be read with a non-privileged ring three or user mode ReaD Time Stamp Counter, RDTSC[4] instruction with which we have achieved very consistent timing results on desktop machines. Yes, even at >= 3 GHz: 1 count =1 tick. Laptops that vary the CPU clock may not be candidates for these techniques.

We have developed two similar software libraries: one for Windows is callable from C or VBA, Visual Basic for Applications and Excel[4], and another for Linux in C or Tcl via CRITCL[5] for Linux.

From C the RDTSC instruction is extremely low overhead with resolution that increases as a function of the clock to the CPU. The higher the CPU clock, the greater the resolution: on a ~1 GHz CPU there will be ~ 1,000 ticks per µs and on ~3 GHz CPU there will be ~3,000 ticks in one µs. On the latter machine, RDTSC, when put into an inline C function and profiled, takes ~500 clock ticks or less than two tenths of a µs. C called from Visual Basic for Applications, VBA and Excel, may take ~1.2-1.6 µs the first instance and only ~0.8 µs during subsequent instances on a 2.1 GHz machine. Tcl calls via CRITCL on Linux are typically a few tenths of a µs slower than the first instance on a ~3 GHz machine running Linux.

II. Overview

We are building the Dark Energy Camera (DECam) for the Dark Energy Survey (DES)[6]. The full focal plane will require 70 devices plus spares, and we expect to test about 200 Charge Coupled Devices (CCD), based on our current yield estimate. The testing facility is located in the Silicon Detector Laboratory at Fermilab. (Some testing may be performed by collaborating institutions). There are several testing stations and each station has a different CPU clock frequency. Many tests involve opening and closing a shutter to expose the CCD to a light source on the opposite side of the shutter. See Fig. 1.

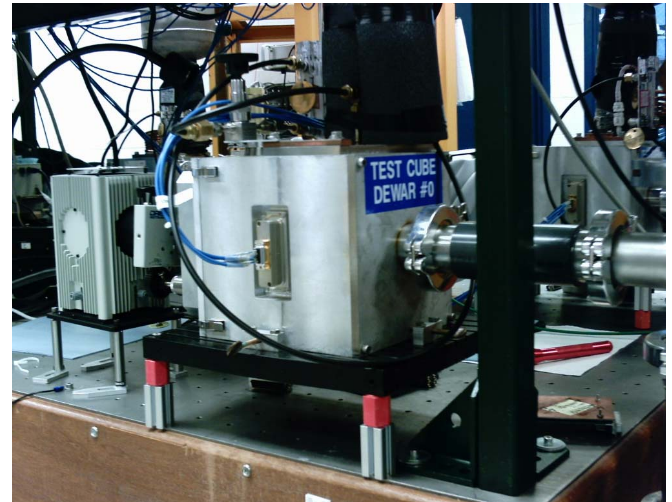

Fig. 1. CCD is in the "Test Cube". The shutter is the small (slightly cocked) rectangle to the left of the Cube and to the right of the "finned" halogen lamp.

Several other GPIB devices need to be coordinated along with the shutter. One example scenario is to open the shutter, wait half the exposure time, read light power via GPIB, profile how long the GPIB operation took with RDTSC and then subtract that time from the remaining shutter exposure time. (We could take out the Tcl calculation time but do not.) Next we make a socket call to close the shutter, which we also log, and that is on the order of about a millisecond. We are already at least ~100 times better than our environment as our log files show:

```
measured_exp_time_secB4SocClose#F=10.0000500391;
measured_exp_time_secB4SocClose#F=10.000048969;
measured_exp_time_secB4SocClose#F=14.0000616151;
……………………………………………………………
measured_exp_time_secB4SocClose#F=90.0000500799;
measured_exp_time_secB4SocClose#F=90.0000299969;
measured_exp_time_secB4SocClose#F=94.0000465351;
measured_exp_time_secB4SocClose#F=94.0000406329;
measured_exp_time_secB4SocClose#F=98.0000456423;
measured_exp_time_secB4SocClose#F=98.0000385692;
measured_exp_time_secB4SocClose#F=100.000043744;
```

[1] This work was done for Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

……………………………………………………………
measured_exp_time_secB4SocClose#F=4000.00002996;
……………………………………………………………
measured_exp_time_secB4SocClose#F=4000.00006041;
... and so on.

III. Profiling

Because of its low latency and high granularity, RDTSC is very good at profiling. One can use it to profile code directly without averaging. One must be aware of things like CPU caching that generally may make the first call longer than the subsequent (presumably) cache hit call. Also when doing many successive calls, some may be quite a bit longer than the rest if the profiled call was preempted. Due to the high granularity of RDTSC, one must be careful interpreting the results. If an average time is desired, one must remember to write explicit code to compute the average.

IV. Calibration

Calibrating the CPU clock in terms of time is essential for code portability between multiple test stations. We do not want to maintain thousands of lines of code in terms of clock ticks on multiple test stations all running with different CPU clock frequencies. For a cost of a second or two at program start up we can use the standard slower operating system time routines to calibrate the RDTSC counter for the machine we are running on and write our code in terms of seconds and/or µs.

On Linux we can use the accurate gettimeofday()[8] function combined with the usleep() function to calibrate the CPU frequency. If we are running on a Linux machines where the gettimeofday() function is synched to a time server then our CPU clock frequency will be referenced to that as well!

V. "Guaranteed Minimum" Microsecond Delay

With calibration complete, we need work only in seconds or µs. Latency of our timing routine determines our granularity. E.g., the Linux gettimeofday() function is reputed to be accurate to a µs[8]:

cpu freq= 3048592092 ticks per microsec=3048.2
gettimeofday took = 5.042897 µs
gettimeofday took = 3.674242 µs
gettimeofday took = 3.619128 µs

Note that the first gettimeofday() took a bit longer the subsequent instances, presumably because it was a CPU cache miss, while the next two obviously were cache hits.

However, the above latency cannot compete with RDTSC:

RDTSC took = 0.187564 µs

Although gettimeofday() is a candidate if we were to poll on it in a tight loop, we will have 20-30 times worse resolution than RDTSC.

So we poll on RDTSC for a calibrated time delay and get at most ~0.4 µs more than what we asked for. We note that this kernel, Fermi Scientific Linux, apparently does not preempt. The profiling indicates consistency to within a µs on both sides. On operating systems which preempt, such as Windows machines, the delay will be as long as requested, but may also go longer when preemption occurs. More details are presented later in this note.

We profile the standard Linux usleep() and compare the error with that obtained using RDTSC in the figure below.

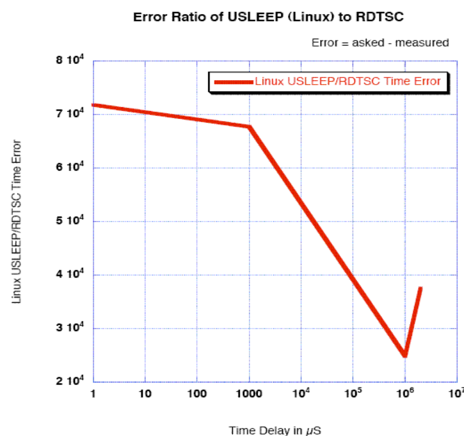

Fig. 2. Linux usleep() / RDTSC time error as a function of requested time delay. Ask usleep() for a 1 millisecond delay results in ~20 milliseconds. With usleep(), the DAQ rate is limited to ~50 hertz regardless of CPU clock speed.

For Windows the Sleep() function from winbase.h at least has their argument in milliseconds.

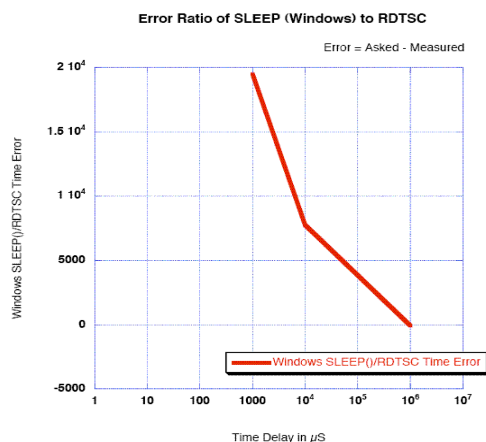

Fig. 3. Ratio of error for Windows Sleep() vs. RDTSC as a function of delay. Asking for a 1 millisecond delay results in ~15 millisecond delay. The DAQ rate is limited to ~66 hertz regardless of the CPU clock speed.

Clearly if high DAQ rates are desired one wants to use RDTSC for more precise delays. Certainly not usleep() or Sleep().

However, polling RDTSC can pretty well lock up the machine. In that way polling RDTSC is not the same as calling the usleep() or Sleep() functions, which release to the kernel and other tasks. If running Linux, another user may decide to reboot the machine. This happened to me once while gathering this data. If on Windows, polling for

RDTSC will also pretty well prevent other uses of the computer such as checking email.

If we interface[9] this delay code to some hardware that we can control directly, such as a standard Parallel Port, we can check our software on a scope as the next three figures show. These are all done on Microsoft XP service pack 2 from Excel via VBA to the Parallel Port.

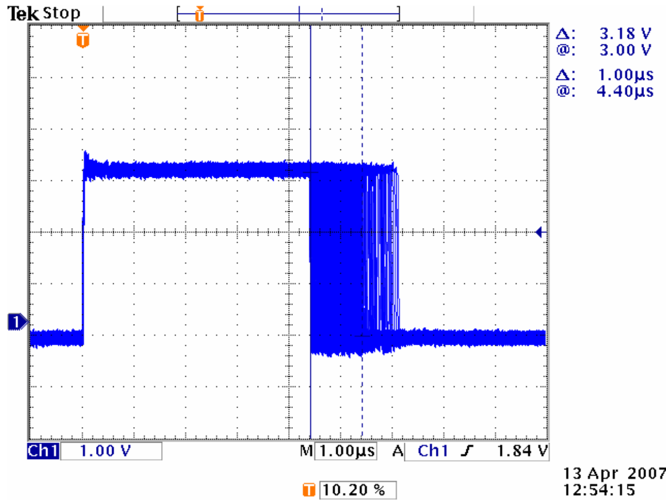

Fig. 4. Many "Guaranteed Minimum" Parallel Port Trigger Pulses with 2 µs delay. Note, no pulse ends before the 1st marker.

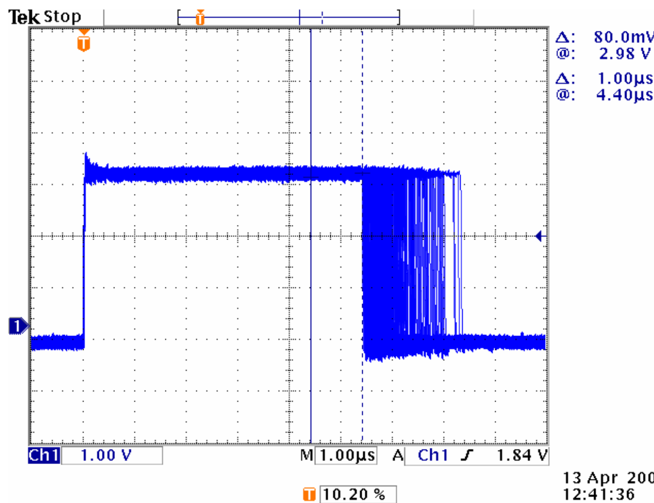

Fig. 5. Many similar Parallel Port Trigger Pulses with another µs added to the delay (3 µs delay requested). Note that no pulse ends before the 2nd marker and that the markers are 1 µs apart.

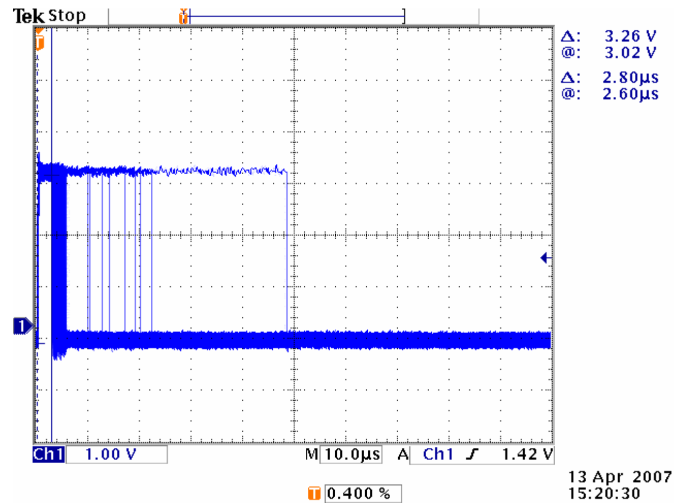

Fig. 6. If we change the scope time base to 10 µs per division and repeat the test many times, we see that preemption sometimes occurs even during our short 4-5 µs pulses on XP. However, RDTSC times reflect this, so we know for how long we have been preempted. If we pull the Ethernet cable out we may reduce the number of premptions but they never entirely go away.

VI. Precision "Long Delays" (Not Guaranteed Minimum)

With a second round of calibration, we gather profiling statistics on how long a given release to the kernel takes. For Windows, a routine like DoEvents in VBA will work. For Linux, we can use usleep() with some arbitrary time on the order of a tenth of a second or more. We are interested in the worst case time this takes and save the results somewhere, for instance, in a global.

After this last calibration, our new poll loop will consist of a coarse loop where we release to the kernel the same amount as our last calibration, and then dynamically check how much time remains until we get close to our target time. When we are within say 0.2 to about 0.5 seconds, we start the above section's non-releasing poll on RDTSC. Most people will tolerate the machine going away for short periods, but usually not seconds.

Because this is more code, with more converting to and from ticks, this routine's latency is much higher and causes loss of µs resolution. We can recover resolution by adding a tuning argument to subtract out this routine's overhead. Thus, we have moved from a "guaranteed minimum" to an average around a target. This is perfect for shutter control. After the first caching call, which is usually ~60-100 µs too long, and after tuning the delay on that machine, we can achieve the accuracy shown on the scope trace in the lower right.

A last tip is to trap negative polling requests to our delay routines that use RDTSC to prevent waiting for 64 bits of ticks (equivalent to ~150-190 years on our ~3 GHz machine).

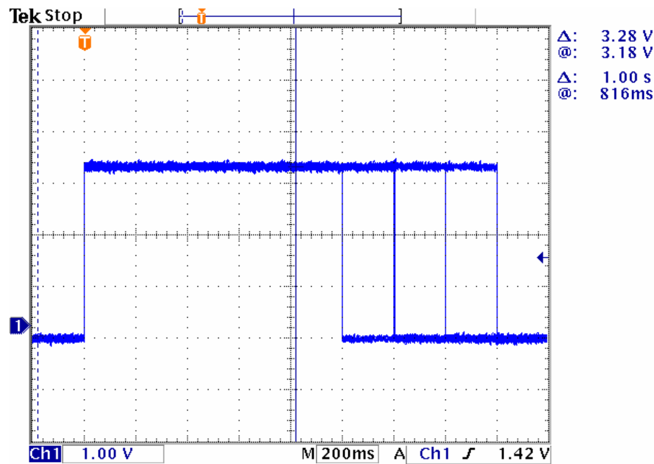

Fig. 7. ~10 each "Long Delays " each at 1.0, 1.2, 1.4 and 1.6 seconds shown at 200 millisecond/ division. From VBA & Excel to C on XP (service pack 2) out of the Parallel Port.

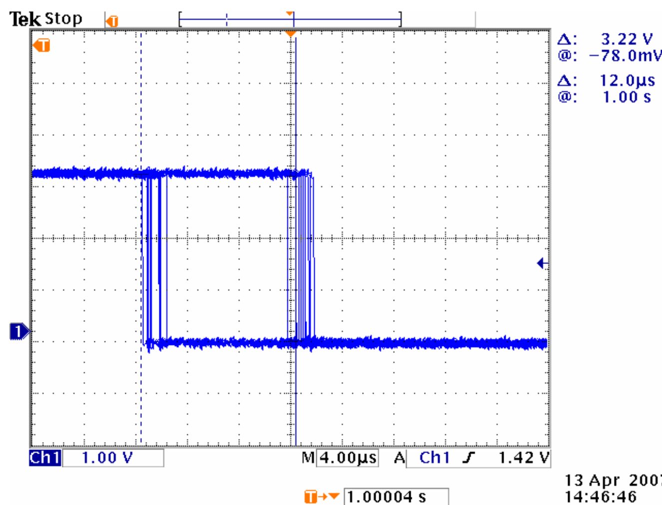

Fig. 8. End of ten "Long Delays" at 1.000004 seconds and ten more at 1.000016 seconds as before. This shows the excellent precision and repeatability. The time base is 4 µs per division and the space between markers is 12 µs.

The previous figure was obtained in the third try and is selected for. Nevertheless, it is twenty consecutive pulses with a delay change of 12 µs, after the first ten, on an operating system that we know can still preempt. Though results are not guaranteed, we know when we have been preempted because the RDTSC time stamp tells us. Similar results are obtained from profiling delays of from 1 to 60 seconds (and longer if desired, but the tests get tedious.) One can set the delay and then browse the web and have it come back within +/- a few µs of 60.000000 seconds inclusive. While CPU utilization can still be high, regardless of releasing to the kernel, the machine will still respond to other tasks.

VII. Conclusion

Once calibrated, RDTSC provides impressively low µs latency profiling and sub-microsecond timing granularity available on any Pentium II or above and/or compatible where the CPU clock is "constant" as on most desktops. The technique is independent of operating system. It is only dependent on processor type and compatibility. Although we have successfully developed software libraries on Windows and Linux, they could be developed on the Apple MacIntosh as well. The scope pictures are proof of our ability to calibrate the RDTSC counter rather well using the standard slow timing functions.

After calibration, precision (1 µs pictured) "guaranteed minimum" delays may be made. This delay may of course be longer due to normal operating system preemption.

The "Long Delay" method provides unexpectedly successful results. The algorithm may be unique or at least not common. The margin of error does not increase as a function of the delay time because any preemption, except in the last few µs, is accounted for. Additionally we always know the returned time delay regardless of preemption.

Delay routines return profiling so the caller may redo measurements if desired. The times agree well with scope traces. These techniques are obviously only applicable in non-life threatening applications. They also depend on a stable CPU clock so they may not be applicable to laptops.

References

[1] "Windows" Copyright Microsoft and here meaning any of their operating systems although the scope pictures here were with XP (all patches at the time of this writing, April 13, 2007).

[2] "Pentium" is copyright by Intel.

[3] AMD is Advanced Micro Devices.

[4] Excel is Microsoft's spreadsheet application that incorporates Visual Basic for Applications (VBA).

[5] http://en.wikipedia.org/wiki/RDTSC

[6] A lot of information about Dark Energy Survey: https://www.darkenergysurvey.org/

[7] Jean-Claude Wippler, "Critcl lets you easily embed C code in Tcl": http://www.equi4.com/starkit/critcl.html

[8] http://en.wikipedia.org/wiki/System_time claims Unix, POSIX (Linux) gettimeofday() resolution is 1 µs. Also that Microsft Windows GetSystemTime() is 1 millisecond and that their GetSystemTimeAsFileTime() is 100 nanoseconds. Although on one of the author's XP systems CompuWare's SoftICE (kernel level) debugger will cause the system time to loose minutes over the course of a debugging session. Obviously these must apply to some sort of "normal" system where no special (debugging) software is running.

[9] Dale Roberts, "DIRECT PORT I/O AND WINDOWS NT" Dr. Dobb's Journal (DDJ), May 1996. Source and binary available for Windows: ftp://66.77.27.238/sourcecode/ddj/1996/9605.zip (Can get to the parallel port directly using inp and outp from c with giveio.sys)