

Execution Time Profiling on a Pentium Processor

Shreepad Hardas



Execution Time Profiling on a Pentium Processor

Many a times, we require profiling our code for execution time characteristics. Most of us already know a few techniques of profiling like

1. Time stamping
2. Use of an explicit timer counter at a known freq.
3. Use of a free IO port for toggle and capture on DSO.

In this article we shall explore a technique similar to (1), (2) above.

The Pentium processor provides a dedicated 64 bit synchronous free running counter for clock cycle measurement. The counter shares the same clock as that of the processor core and hence provides accuracy down to individual clock cycle of the operating frequency. Thus if our desktop is running at 1GHz, we can get accurate timing measures of 1 nano-seconds!!

This register is called the "*Time Stamp Counter*" (TSC) and can be accessed using the assembly instruction Read Time Stamp Counter "*RDTSC*".

The 64 bit value is read into EDX:EAX. The higher order 32 bits in EDX and lower order 32 bits in EAX. This instruction can be executed at any privilege level. The *Time Stamp Disable* (TSD) flag in register CR4 restricts the use of the *RDTSC* instruction. When the TSD flag is clear, the *RDTSC* instruction can be executed at any privilege level. While if the flag is set, the instruction can only be executed at privilege level 0. The time-stamp counter can also be read with the *RDMSR* instruction at privilege level 0.

Being a non-serializing instruction, it does not necessarily wait until all previous instructions have been executed before reading the counter.

Since the TSC counts cycles, in order to convert to time we need to use the following equation:

$$\#seconds = \#cycles / (freq. Hz)$$

The difference can then be calculated as per requirement to get the time-measure. For a more accurate measure, the overhead of the instruction itself can be subtracted twice from the calculated difference. Another important consideration can be occurrence of interrupts etc. which would add to the execution time of the function being profiled. This time can in-turn be measured and subtracted from the actual difference.

The advantage of this method is its simplicity, accuracy with minimum resource and processing overhead. This method is well suited for profiling Interrupt Service Routines (ISRs).

One limitation is that it can only be used in sync with the program execution. Another limitation is that we need to disable any power management features of the system which can cause variation in the processor operating frequency. The technique has to be carefully used with multiprocessor systems executing parallel threads etc.

Following is the code that can be used for profiling under Linux. Similar yet specific code for the

windows platform too, is available.

```
#define OPERATING_FREQ    YYYYY /* your processor frequency in Khz */
```

```
unsigned long cpu_khz = OPERATING_FREQ;
```

```
#define CLOCK (cpu_khz/1000)
```

```
typedef struct
```

```
{  
    unsigned long hi;    //higher 32 bits  
    unsigned long lo;    //lower 32 bits
```

```
}time_586;
```

```
#define TimeStamp(x) \  
    __asm__ __volatile__ ("rdtsc" : "=d" (x.hi), "=a" (x.lo))
```

```
/*
```

```
* We cannot use floating point inside kernel, because FPU regs are  
* not saved on IRQ. So we use integer arithmetics, and get the  
* timings in clock ticks rather than usecs.
```

```
*/
```

```
static inline unsigned long long k_time_diff_ticks (time_586 b, time_586 a)
```

```
{  
    unsigned long long t2,t1;  
  
    /* convert to 64 bit */  
    t2 = (((unsigned long long)b.hi)<<32) + (unsigned long long)b.lo;  
  
    /* convert to 64 bit */  
    t1 = (((unsigned long long)a.hi)<<32) + (unsigned long long)a.lo;  
  
    return t2 - t1;    /* return difference in cycles */  
}
```

```
/*
```

```
* for user mode time difference we can use floating point  
* NOTE: this will be slower than the above routine
```

```
*/
```

```

static inline double u_time_diff(time_586 b, time_586 a)
{
    double t2,t1,res;

    /* convert to 64 bit */
    t2 = (double)b.hi*(double)(1<<16)*(double)(1<<16)+(double)b.lo;

    /* convert to 64 bit */
    t1 = (double)a.hi*(double)(1<<16)*(double)(1<<16)+(double)a.lo;

    if (t2 < t1)
    {
        /* take absolute difference & convert to usec */
        res = ((double)(1<<16) * (double)(1<<16) * (double)(1<<16) * (double)(1<<16)+t2-
        t1) / (double)CLOCK;
    }
    else
    {
        res = (t2-t1)/(double)CLOCK; /* return the time difference in usec */
    }

    return res;
}

```

```

/* example usage */
/*
* time_586 x1, x2;
* unsigned long long CycleCount = 0;
*
* TimeStamp(x1); /* time stamp just before function call */
*
* SomeFunctionToBeProfiled();
*
* TimeStamp(x2); /* time stamp immediately after function call */
*
* CycleCount = k_time_diff_ticks(x2, x1);
*
*/

```