Time Stamp Counters Library - Measurements with Nano Seconds Resolution

`Library Version: 2.1`

Yoav Etsion and Dror G. Feitelson

July 22, 2001

School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
{etsman,feit}@cs.huji.ac.il

Abstract

As current computers become faster, and more applications require very tight time constraints and optimizations, there is a growing need for a simple and robust manner to evaluate performance with extremely low overhead. The best way to this is to directly measure the CPU cycles. The standard method of evaluation is by extracting the system time using operating system calls. This method incures a very substantial overhead since it implicitly includes two context switches - to the system context and back - and thus can only be used in a very coarse grained methodolgy. Intel, makers of the most popular workstation CPUs, has incorporated into its P5 processors family (Pentium and up) an opcode which allows to read the CPU's cycle counter from user level. Our library enables the user to use this opcode and to take these coveted cycle measurements, that can be used for fine grained performance evaluation. These cycles measurements can be easily converted, using the CPU speed, into nanosecond resolution time measurements. Also, we designed the library to be very portable across most of the common operating systems that are used with the P5 processor family.

1 Overview

Starting with the Pentium processor family, Intel introduced a new opcode, "*rdtsc*" [3], which enables a program running at user level to read the processor's Time Stamp Counter — the number of cycles since the processor was last reset. This feature enables a user level program to take cycles measurements with an extremely low overhead. Furthermore, since this cycle count can be easily converted into nano seconds (using the CPU speed), using this library

one can take time measurements with only a fraction of a system call overhead (section 3). Note that even though most operating systems mesaure times using the CPU cycle counter, there is usually no way to get the cycle count, only time measurements with limited resolution. Thus our library supplies the programmer with a powerful tool to evaluate performance.
Even though there are a few other libraries out there with the same functionality, we experimented with one or two, only to find out that they they are overly complicated for our needs, and are difficult to learn and use. This motivated us to write another library with the intention to supply the programmer with an interface that is as simple as possible.

1.1 Design

Our two main goals designing the interface were performance and portability, and we hope to have acheived both. The design was greatly influenced by both goals: the interface offers three types to measure with (section 2.1), two of them are special, library specific types. Also, most of the library is written inlined, and the implementation was detached from compiler specific syntax (such as assembly code syntax and non-standard integral types), using a special low level Abstract System Interface (ASI), which offers easy porting to new operating systems and compilers. This low level interface does not degrade performance, because its time critical sections are inline functions and macro definitions. This interface is further discussed in the porting section (section 6).

1.2 Supported systems

Since the library uses a Pentium - specific opcode, it is desgined for Intel platforms, and currently cannot be used on other platforms.

However, since the library uses a few operating system services (time measurement and sleep functions), and use some compiler syntax which is not standard (assembly code, non-standard types), we detached the implementation from the system using an Abstract System Interface.

We implemented the library using the ASI for several development environments:

- GNU compiler set, tested on Linux, BSDI and Solaris. Since the GNU compiler set is implemented on other platforms, this library can probably be compiled for them, but it was not tested. If you have compiled it for new systems, please send email to the authors, describing the compilation, system and any necessary patches.

- Visual C++ which was tested on Windows NT4. Again, it was not tested on other windows systems (95/98, 2000, and ME), but since all these platforms use the same compiler set, there shouldn't be any problem using the library on those systems.

Table 1: Compiler optimizations flags

Compiler	Optimization flag
GNU compiler collection	-O and up (-O2, -O3)
Visual C++	/O2
Watcom compiler (QNX RTOS)	-Oe

- Watcom compiler, tested on QNX Real-Time OS (RTOS). The Watcom compiler set is also used with DOS system, but again, we did not test it for DOS.

1.3 Compilation

The library code is highly optimized. However, some of the optimizations might not take effect with most compilers if the code is compiled without a compiler optimization flag. Since most of the code is inlined, it is not compiled into the library, but rather straight into the user code when using the library. Because of this reason, always make sure when using the library to compile your code at least at the minimal compiler optimization level. On the compilers we used, these flags are shown in table 1. If you port the library to a new platform/compiler, please send us compiler optimization flag required, along with the ASI implementation.

2 Interface

When designing the interface our goals were to give the user both performance and convenience. When those conflicted, we tried to offer a choise.

2.1 Types

The hardware's time stamp counter is a 64 bit integral register. Currently, the biggest standard integral type that is supported by the Intel hardware is only 32 bits in size. Some compilers support 64 bit integral in software, but such types are not standard so even the compilers that do support it name it in various ways, such as *"long long"* or *"int64"*. Such support is less than optimal since all operation are simulated in software using 32 bit integers. This problem forced us to offer the user three possible type to work with:

1. *tsc_counter_t*
   The basic storage type of the clock cycle counter. Since some systems do not offer 64 bit integral support, the only type that can hold such big numbers is double. However, since a double precision type cannot represent all 64 bit integer values, and since conversion to and from it incur unnecessary overhread, we found it necessary to keep the raw data (two 32 bit integers) in an internal data structure, for later processing. Using this

type in measurements gives the fastest and most accurate measurements, but is more complicated to manipulate. Such measurements have to be saved and later converted into one of the other two types, for manipulation.

2. *tsc_counter_int_t*
   The integral type, or actually the best atomic type available by the system that is large enough to hold a 64-bit unsigned integral value. Basically, the only assurances on this type is that is supports mathematical manipulations, and it's efficeincy is no better than *tsc_counter_t* and no worse than *double*. We give the user an abstraction of the actual type, because not all systems support such integral types, and sometimes we have to emulate using *double* values, and those that do support such large integers does not have a standard name for it. The only assurance the library gives a user is that this type can be used with all mathemtical operators (+,-,*,/, etc.) to calculate time differences. The library also provides the user with a function that translates the integral value into a character array. On platforms that supports a 64 bit integral type, there is no advanage for using the *tsc_counter_t* type, and this type could be used instead. This can be determined by the presence of the macro *TSC_UINT64* is defined (see section 2.2).

3. *double*
   The trivial data type known by all. This is not the best type to measure with — manipulating *double* variables suffers a bigger overhead than the other two types (see section 3), but it is still much faster than a system call. Also, using *doubles* may be less accurate because not all integers can be accurately represented by it. Using *double*s may incur an inaccuracy overhead of some units (either nanoseconds or cycles). This type is best for users who do not measure with nanosecond resolution, but rather prefer a more convenient type to work with over fastest possible measurements.

2.2 Macros

The macro API offer the programmer a some special values, that are used as special parameters and return values. Also, it gives the programmer a way of knowing what features are supported by the system, which the library uses. This simply helps knowing how optimized the library can be on the current system.

2.2.1 Hardware Related Macros

This section describes the macros that are used to describe the features supported by the compiler and used by the library.

- *TSC_UINT64*
  Defined if the system supports 64 bit integers. This simply offers the

user a way to know how optimized the *tsc_couner_int_t* type is compare to the fastest *tsc_counter_t* type. When this macro is defined the *tsc_counter_int_t* type is actually a synonym for a 64 bit integer, and as such its overhead is no bigger than that of *tsc_counter_t* type. Since the *tsc_counter_int_t* type can be mathematically manipulated, it is more convenient to work with so in this case the programmr can use it with no extra overhead.

2.2.2 Special Value Macros

These macros are special values to be used a function parameters and return value.

- *TSC_NOSPEED*
  A macro which indicates the user asks the library to calculate the CPU speed at runtime, instead of hardcoding it. More in section 2.3.1.

- *TSC_OK*, *TSC_ERROR*
  Macros that indicates the success of an operation. Any function that can fail returns one of these macros, to indicate the status of the operation.

2.3 Functions

The three types are repesented in the function names by C (*tsc_counter_t*), CI (*tsc_counter_int_t*) and D (*double*) (and another type acronym, STR, describes the null terminated string type, *const char**). Each critical operation was implemented using the all the three mathematical types. A function's name indicate its functionality, while the symbol in the function name's suffix indicates its type. This is also the case in the conversion functions: the functions used to convert values between the types. The functions can be grouped according to their functionality:

2.3.1 Initialization

- int tsc_init(int processor_speed_Hz)

  Initializes the library's internal data. This function must be called only once, and before any other library function, in order for the data to be consistent.
  Parameter: the processor's speed in Hz (not MHz). If the user supplies the *TSC_NOSPEED* macro as parameter, the library measures the CPU speed by itself (see section 2.3.2).
  Return value:
  On success *TSC_OK*, on error *TSC_ERROR* (if one ever happens...).
  Note:
  This function might take up to 1 second to run, for an accurate CPU

measurement.

- unsigned long tsc_recalc_CPUSpeed(void)

  Recalculates the cpu speed, so we can get a more accurate speed over time - a user can call this function every some time to make sure the measured speed is accurate over time (after a remark about the stability of Intel's CPU's speed from Ahmet Uyar <auyar@csit.fsu.edu>.
  Return value:
  the CPU speed measured in this iteration (not the average, which can be obtained by tsc_CPUSpeed()).
  Note:
  This function might take up to 1 second to run, for an accurate cpu measurement.

2.3.2 CPU speed

- *double tsc_CPUSpeed()*

  Returns the CPU speed calculated in the init function (or passed to it). The calculation is straight forward: the function takes a cycles counter reading, an operating system time measurement and sleeps for some time. After the sleep, it takes another cycles counter reading and operating system time measurement. Then it is simply a matter of calculating the number of cycles the CPU executed during the measured time. We found that sleeping for 200 miliseconds is accurate enough even for CPUs running at 600MHz, but this might increase for faster CPUs when available. The best option is to check the measurement accuracy on a given host (section 5.2).
  Return value:
  The calculated CPU speed, in Hz (not MHz).

2.3.3 Cycle Counter Read

- *tsc_counter_t tsc_readCycles_C()*
- *tsc_counter_int_t tsc_readCycles_CI()*
- *double tsc_readCycles_D()*

  The main issue. Takes a reading of the cycles counter, and returns the number of cycles the CPU executed since it was last reset. As said before, the return type implicates the measurement's efficiency. Using *tsc_counter_t* values (C suffix) is the fastest measurement, but the values must be converted later to a calculable type. When using the fatest

measurement, it takes ~35 CPU clock cycles (see section 3.2).
Return value:
The number of cycles since the last CPU reset.

2.3.4 Nanoseconds Time Measurement

- *tsc_counter_int_t tsc_readNsec_CI()*
- *double tsc_readNsec_D()*

Get a direct time measurement. Returns the time in nanoseconds since the CPU was last reset, a data which is unimportant in itself, but can be used to time fine grained operations. Since the time measurement involves some mathemtical calculations from the original cycles counter, there is no point returning fastest type (*tsc_counter_t*). Since the CPU speed itself is not accurate, and depends on volatile factors such as temprature, CPU age, etc. we can safely assume that calculating the nanoseconds time will not be impaired if we use *double* variables. This is why all the internal calculations are done using *double*s, so there is actually no difference between the two functions. The first is given simply for sake of versatility, although on some systems the it incurs a much bigger overhead. More in sections 3.3 and 5.1.
Return value:
Number of nanoseconds since the last CPU reset.

2.3.5 Cycles Count to Nanoseconds Conversion

All unit conversion functions are named after a simple naming acheme: *"orig_type 2 new_type"*. The types' acronyms are described in the beginning of this section.

- *tsc_counter_int_t tsc_cycles2Nsec_C2CI(tsc_counter_t counter)*
- *double tsc_cycles2Nsec_C2D(tsc_counter_t counter)*
- *double tsc_cycles2Nsec_CI2D(tsc_counter_int_t counter)*
- *double tsc_cycles2Nsec_D2D(double counter)*

Convert a CPU cycles measurement into a time measurement, either to tsc_counter_int_t type, or to *double*. Again, if the compiler supports 64 integral types (TSC_UINT64 is defined) the first option is faster and more accurate. In other cases, there is no difference. The time is measured in nanoseconds since the computer was last reset. Both the parameter and the return value's type are determined by the functions' suffixes. For example, the CI2D suffix indicates that the function accepts a *tsc_counter_int_t* parameter, and returns *double*.
Parameter:

The CPU cycles measurement. Its type is determined by the name's prefix (C, CI or D).
Return value:
The time since the CPU was last reset, in nanoseconds. Its type is determined by the name's prefix (CI or D)

2.3.6 Type conversion

All type conversion functions are named after a simple naming acheme: *"orig_type 2 new_type"* . The types' acronyms are described in the beginning of this section.

- *tsc_counter_int_t tsc_C2CI(tsc_counter_t counter)*
- *double tsc_C2D(tsc_counter_t counter)*
- *double tsc_CI2D(tsc_counter_int_t counter)*

  Translate the tsc special types to *double*, and among themselves. The special types are optimized for percision and speed, but sometimes it is more convenient to convert them.
  Parameter:
  The original typed value.
  Return value:
  The new typed value.

- *const char* tsc_C2STR(tsc_counter_t counter, int char_num)*
- *const char* tsc_CI2STR(tsc_counter_int_t counter, int char_num)*

  Convert a number value to a string. This function can be used to print a specially typed value.
  Parameters:
  counter - need I say more?
  char_num: the number of characters to print. If the actual value is bigger, only prints the least significant digits. If char_num is 0 the string is not truncated.
  Return value:
  A static buffer containing the string representation of the value.
  Note:
  Both functions return a pointer to a single static buffer, so the next call to either one of these functions will corrupt the value.

3 Overhead

Like every measurement function, ours also incur some overhead. Since the library is designed to let the programmer separate the measurement itself from

the time/cycles calculations, we only measured the overhead of the different measurements, and not the type conversions. We also report the overhead of some frequently used system calls, including the most common for time measurements — gettimeofday [1]and time [2].
All the overhead measurements were taken on a Pentium III running at 500MHz, with 128MB RAM, using special benchmarks, which are described in section 5.1.

3.1 System Calls Overhead (Placebo)

Since we deal with time measurements, we first measured the most common method to obtain such data — the operating system services, or system calls. These measurements we've taken on Linux and BSDI only, but there is not reason to assume that QNX or Windows NT are any faster.

System Call	Overhead in Cycles	Overhead in nanosecs (PentiumIII 500MHz)
time	2504.21	5035.13
gettimeofday	1192.79	2394.85
dup2	9479.65	19041.35
close	2381.47	4781.66

As we can see, it is impossible to measure at a resolution of less than a few microseconds.

3.2 Cycles Count

The test and library were compiled with the optimization flags (see section 1.3) on Linux, QNX, and Windows NT 4. The fastest function (*tsc_readCycles_C*) was measured on all platforms, whereas the other two only on Linux and BSDI. Since the measurements that were taken on all platforms were identical (up to 1%), we will only report the results of the Linux measurements (the number are averages of 1000 iterations):

Library Function	Overhead in Cycles	Overhead in nanosecs (PentiumIII 500MHz)
tsc_readCycles_C()	35.05	70.42
tsc_readCycles_CI()	36.03	87.25
tsc_readCycles_D()	87.25	175.24

Our library offers the programmer a way to take time measurements two order of magnitude faster than system calls. This is a very powerful tool that allows very fine grained time measurements.

3.3 Time Measurements

Again, all measurements were taken on Linux. There is no reason to suspect that there will be any difference between any development environments that support a 64 integral type (such as Windows NT + Visual C++), but on environments that do not support such type the *tsc_counter_int_t* type is implemented using *double* so there will be no difference between the two functions.

Library Function	Overhead in Cycles	Overhead in nanosecs (PentiumIII 500MHz)
tsc_readNsec_CI	312.20	627.57
tsc_readNsec_D	92.51	185.96

This vast difference is simply explained: Since the conversion between cycle count and time is done in *double* values, and since all functions are inlined, the first function is implemented using a call to the second, and then a type conversion from *double* to *tsc_counter_int_t* (which is implemented on Linux using a 64-bit integral type *unsigned long long*). Since 64 bit integrals are not supported by the hardware, such conversion takes place in software and thus is very slow.

4 Limitations

Since we are using some non-standard types to maintain 64-bit integral numbers, certain counters might overflow in specific situations. Also we are using a very hardware oriented code that relies on the CPUs' own counters. When using such code, we must take into account the limitations posed on us:

1. The library is not SMP safe! To make it so we must lock the process to run on a specific CPU, so we won't get different values when migrating from one CPU to another. Most SMP operating systems support such requirements using a special interface, but currently the library itself does not wrap these OS specific calls into one interface.

2. The most strict but least significant limit: The CPU's Time stamp Counter will, of course, overflow after 2^{64} cycles (64-bit counter). For a 1GHz processor this will occure after: 2^{64} cycles $\approx 2^{34}$ seconds ($2^{30} \approx 1G$) ≈ 213503 days ≈ 584 Years, so we realy don't think it matters. If you do encounter such problems, please send a bug report to our descendants...

3. No limitation section is complete without a Microsoft specific limitation, so here is ours: Although Microsoft were gracious enough to supply the programmer with an unsigned 64 bit integral type, they did not bother to implement a direct cast between this type (*unsigned __int64*) and *double*. To do this, one has to cast the unsigned type to the signed type (*cast unsigned __int64* to *signed __int64*), and only then cast it to *double*. This means of course, that all *double* measurements (especially time measurements which are calculated using *double)* overflw at half the values, or after ~277 years. Again, since no Microsoft computer will ever get that uptime, this is really not a problem.

5 Benchmarks

In order to asses our library we wrote several benchmarks, which are part of the library distribution.

5.1 Overhead benchmarks

- *test_overhead*

  Measures the overhead of the library's cycle and nanoseconds measurement functions. Simply iterates on a single measurement 1000 times, and returns the average overhead for each possible return type (*tsc_counter_t*, *tsc_counter_int_t* and *double).* The overhead is displayed both as cycle count and as nanosecond timer.

- *test_syscall <syscall name> <sleep time> <iterations>*

  Measures the overhead incured by the two most common time measurement system calls (*time* and *gettimeofday*), and by two other common system calls (*dup2* and *close*). Executes the system call then sleeps for the requested number of iterations. Displays the overhead for each iteration, as well as the average overhead over all iterations. The sleep between two measurements is required because we noticed that calling a system call consequtively shortens its runtime, whereas in the real world a system call are rarely called consequtively.
  Parameters:
  syscall name - the name of the syscall to measure (one of the four mentioned above).
  sleep time - the time to sleep each iteration, in milliseconds.
  iterations - the number of iterations to repeat the measurement.

5.2 CPU Speed Measurements benchmarks

- *test_cpuspeed <iterations>*

  Tests the self measured CPU speed. This test gives the user and indication that the CPU speed time measurement done internally by the library is correct, and very close to the operating system measurement (that can be found on Linux in */proc/cpuinfo*, or in the Windows NT registery).
  Parameter:
  iterations - the number of times to iterate the test. The more time, the more accuate the mesurement is.

6 Porting issues

Because this library is very hardware oriented and uses inline assembly instruction, and since it requires some non-standard types, we found it necessary to rewrite the low level bindings for each development environment. In order to refrain from clobbering the code with numerous *ifdef* clauses, we decided to extract all the non-standard definitions into one abstract interface, which will hide the operating platform from the implementation (much like the HAL for Windows NT). We called this interface Abstract System Interface — ASI. It is composed of macros, typedefs and inline functions. The exact interface is described in the following section.

6.1 Abstract System Interface

6.1.1 typedefs and types

- typedef struct {
  unsigned long low;
  unsigned long high;
  } tsc_int64;
  This is the only type that is defined by the library and not by the ASI. It is a library internal struct that consists of two *unsigned long* values, and is used to hold the counter values read from the CPU's time stamp counter. It is used to pass parameters to the low level *rdtsc* inline function (section 6.1.3).
- *TSC_UINT64*
  The type for the 64-bit integral values received from the CPU's time stamp counter. Such integral types are not standard, thus only exist on few development environments, and when they do exist there is no standard naming convention. If non suc integral type exists, this should simply be an alias to *double*. However, if this is indeed a large enough integral type, the macro *TSC_UINT64* should be defined (see section 6.1.2).

6.1.2 Macros

- TSC_UINT64
  This macro indicates that the typedef binding of *TSC_UINT64* is to a 64-bit integral type, and not to *double*. If the type *TSC_UINT64* is an alias to *double*, this macro should not be defined.
  Note:
  The idectical names of this macro and the typedef are for clarity. There is not colision problem if the macro is defined as: *"#define TSC_UINT64 TSC_UINT64"*
- *UINT64_TO_DOUBLE(ull), DOUBLE_TO_UINT64(d)*
  These one parameter macros are used convert to between the bound type

and *double.* Their names suggest their functionalities.

- *INLINE*
  This macro is used to declare a function as inlined for best performance. Although most platforms accept the word *inline*, some compilers either recognize some underscored version of the word (*__inline*, *__inline__* etc.), or give a more optimized inlining version (GNUC has *extern inline).* This macro should be bound to the best inlining command available by the compiler.

6.1.3 Inline functions

- *void rdtsc(tsc_int64 * const counter)*
  This is the heart of the library. This inline function contains the assembly code to operate the *rdtsc* opcode that reads the time stamp counters off the CPU [3]. It reads the value into the struct whose pointer is passed as a parameter.
  Parameters:
  counter - A pointer to the struct to which the counter data will be read.
  NOTE:
  Since this is the basis of all measurements, this function is best when implemented using a macro, but in any case it should be super optimized. We suggest compiling the code into assembly first to see that the function is indeed inlined and it is optimized. Sometimes the best idea if to write is all in assembly (as the case with Windows NT and QNX).

The following functions are library internal functions, and it is best if they are not exported. Since they are inlined, a simple *static* declaration is not enough. For this reason the library defined the macro *TSC_PRIVATE_FUNCTIONS* before including the platform specific file, and it is best to protect those functions' declarations with a precompiler ifdef clause: *#ifdef TSC_PRIVATE_FUNCTIONS.*

- *char *tsc_platformUINT64ToStr(TSC_ UINT64 counter, char* buffer)*
  Convert the library's special type into a character array. Since the type itself is platform dependant, and each platform that support 64-bit integrals has its own method of printing them, this function is required. The buffer is guarenteed to be big enough to hold any 64-integer.
  Parameters:
  counter - The integral value
  buffer - The buffer to which the string representation of the counter is to be written.
  Return value:

A pointer to *buffer*

- *double tsc_msTime(void)*
  Simply return a millisecond time counter. Since this function is only used when we measure the CPU speed, its speed is not important and actually it should usually be a wrapper of some system call. We do not care what is the epoch time for the function (the time where its counter started), since we only measure time differences.
  Return value:
  The current time in milliseconds

- *void tsc_msSleep(unsigned long msec)*
  Sleeps for msec milliseconds. A wrapper for each operating system's own sleep functions.
  Parameters:
  msec - The time to sleep, in milliseconds.

Acknowledgements

A special thanks goes to Avi Kavas, who suffered as our beta tester while using the initial versions of this library in his M.Sc. thesis' research.

References

[1] gettimeofday system call manual page.

[2] *IEEE Standard Portable Operating System Interface for Computer Environments.*

[3] Intel Corp. *Intel Architecture Software Developer's Manual*, 1997.