A High-precision Time Handling Library

Irina Fedotova
Faculty of Information science and Computer Engineering
Siberian State University of Telecommunication and Information Sciences, Novosibirsk, Russia
i.fedotova@emw.hs-anhalt.de

Eduard Siemens, Hao Hu
Faculty of Electrical, Mechanical and Industrial Engineering
Anhalt University of Applied Sciences
Koethen, Germany
{e.siemens, h.hu}@emw.hs-anhalt.de

***Abstract*—An appropriate assessment of end-to-end network performance presumes highly efficient time tracking and measurement with precise time control of the stopping and resuming of program operation. In this paper, a novel approach to solving the problems of highly efficient and precise time measurements on PC-platforms and on ARM-architectures is proposed. A new unified *High Performance Timer* and a corresponding software library offer a unified interface to the known time counters and automatically identify the fastest and most reliable time source, available in the user space of a computing system. The research is focused on developing an approach of unified time acquisition from the PC hardware and accordingly substituting the common way of getting the time value through Linux system calls. The presented approach provides a much faster means of obtaining the time values with a nanosecond precision than by using conventional means. Moreover, it is capable of handling the sequential time value, precise sleep functions and process resuming. This ability means the reduction of wasting computer resources during the execution of a sleeping process from 100% (busy-wait) to 1-1.5%, whereas the benefits of very accurate process resuming times on long waits are maintained.**

Keywords-high-performance computing; network measurement; timestamp precision; time-keeping; wall clock.

I. INTRODUCTION

Estimation of the achieved quality of the network performance requires high-resolution, low CPU-cost time interval measurements along with an efficient handling of process delays and sleeps [1][2]. The importance on controlling these parameters can be shown on the example of a transport layer protocol. Its implementation may need up to 10 time fetches and time operations per transmitted and received data packet. However, performing accurate time interval measurements, even on high-end computing systems, faces significant challenges.

Even though Linux (and in general UNIX timing subsystems) uses auto-identification of the available hardware time source and provides nanosecond resolution, these interfaces are always accessed from user space applications through system calls. Thus it costs extra time in the range of up to a few microseconds – even on contemporary high-end PCs [3]. Therefore, direct interaction with the timing hardware from the user space can help to reduce time fetching overhead from the user space and to increase timing precision. The Linux kernel can use different hardware sources, whereby time acquisition capabilities depend on the actual hardware environment and kernel boot parameterization. While the time acquisition of some time sources costs up to 2 microseconds, others need about 20 nanoseconds. In the course of this work, a new *High Performance Timer* and a corresponding library *HighPerTimer* have been developed. They provide a unified user-space interface to time counters available in the system and automatically identify the fastest and the most reliable time source (e.g. Time Stamp Counter (TSC) [4][5] or High-Performance Event Counter (HPET) [6][7]). In the context of this paper, the expression *time source* means one of the available time hardware or alternatively the native timer of the operating system, usually provided by the standard C library.

Linux (as well as other UNIX operating systems) faces a significant problem of inaccurate sleep time, which is known for many years, especially in older kernel versions, when Linux has provided a process sleep time resolution of 10 msec. This leads to a minimum sleep time of about 20 msec [8]. Even nowadays, when Linux kernels usually reduce this resolution down to 1 msec, waking up from sleeps can take up to 1-2 msec. With kernel 2.6 the timer handling under Linux has been changed significantly. This change has reduced the wakeup misses of sleep calls to 51 µsec on average and to 200-300 µs in peaks. However, for many soft-real-time and high-performance applications, this reduction is not sufficient. Presented *High Performance Timer* not only significantly improves the time fetching accuracy, but also addresses the problem of those imprecise wakeups from sleep calls under Linux.

These precision issues lead to the fact that, for high-precision timing within state machines and communication protocols, busy-waiting loops are currently commonly used for waits, preventing other threads from using the given CPU. The approach of the *High Performance Timer* library aims at reducing the CPU load down to an average of 1-1.5% within the sleep calls of the library and at raising the wakeup precision to 70-160 nsec. Reaching these values enables users of this library to implement many protocols and state machines with soft real-time requirements in user space.

The remainder of the paper is organized as follows. In Section II, related work is described. Section III shows the specific details of each time source within the suggested single unified High-Performance Timer class interface. In Section IV, we briefly describe the implemented library interface. Some experimental results of identifying appropriate timer source along with their performance characteristics are shown in Section V. In Section VI,

 ISBN: 978-1-61208-256-1

precise process sleeping aspects are shown. Finally, Section VII describes next steps and future work in our effort to develop a tool for highly efficient high-performance network measurements.

II. Related Work

Since the problem of inefficient time keeping in Linux operating system implementation has become apparent, several research projects have suggested to access the timing hardware directly from user space [1][9][10]. However, most of this research considers handling of a single time hardware source only, predominantly the Time Stamp Counter [1][9][11]. Other solutions provide just wrappers around timer-related system calls and so inherit their disadvantages such as the high time overhead [12][13]. In other proposals, the entire time capturing process is integrated into dedicated hardware devices [14][15]. Most of this research focuses only on a subset of the problems, addressed in this work. Our work with the *HighPerTimer* library improves timing support by eliminating the system call overhead and also by application of more precise process sleep techniques.

III. Unified Time Source of the *HighPerTimer* Library

While most of the current software solutions on Linux and Unix use the timing interface by issuing *clock_gettime()* or *gettimeofday()* system calls, *HighPerTimer* tries to invoke the most reliable time source directly from the user space. Towards the user, the library provides a unified timing interface for time period computation methods along with sleep and wakeup interfaces, independently from the used underlying time hardware. So, the user sees a "unified time source" that accesses the best possible on the underlying hardware, and that generally avoids system call overheads. The *HighPerTimer* interface supports access the mostly used time counters: TSC, HPET and, as the last alternative, the native timer of the operating system, through one of the said Unix system calls. The latter time source we call the *OS Timer.*

Using the *Time Stamp Counter* is the fastest way of getting CPU time. It has the lowest CPU overhead and provides the highest possible time resolution, available for the particular processor. Therefore, in the context of our library, the TSC is the most preferable time source. In newer PC systems, the TSC may support an enhancement, referred to as an *Invariant TSC* feature. *Invariant TSC* is not tightly bound to a particular processor core and has, in contrary to many older processor families, a guaranteed constant frequency [16]. The presence of the *Invariant TSC* feature in the system can be tested by the *Invariant TSC* flag, indicated by the *cpuid* processor instruction. For most cases, the presence of this *Invariant TSC* flag is essential in order to accept it as a *HighPerTimer* source.

Formerly referred by Intel as a Multimedia Timer [7], the *High Precision Event Timer* is another hardware timer used in personal computers. The HPET circuit is integrated into the south bridge chip and consists of a 64-bit or 32-bit main counter register counting at a given constant frequency between 10 MHz and 25 MHz. Difficulties are faced when the HPET main counter register is running in 32-bit mode because overflows of the main counter arise at least every 7.16 minutes. With a frequency of 25 MHz, register overflows would occur even within less than 3 minutes. So, time periods longer than 3 minutes can't reliably measured in 32 bit mode. So, in the *HighPerTimer* library, we decided to generally avoid using the HPET time source in case of a 32-bit main counter.

For systems, on which neither TSC nor HPET are accessible or TSC is unreliable, an alternative option of using the OS Timer is envisaged. This alternative is a wrapper issuing the system call *clock_gettime()*. This source is safely accessible on any platform. However, it has the lowest priority because it issues system calls, with their time costs of up to 2 microseconds in worst case [17][18].Depending on the particular computer architecture and used OS, these costs can be less due to the support of the so-called virtual system calls. These calls provide faster access to time hardware and avoid expensive context switches between user and kernel modes [19]. Nevertheless, invocation of *clock_gettime()* through a virtual system call is still slower than the acquisition time value from current time hardware directly. The difference between getting the time value using virtual system calls and getting the time values directly from the hardware is about 3 to 17 nsec, as measurement results, discussed in Section V, show.

IV. The HighPerTimer Interface

The common guidelines on designing any interfaces cover efficiency, encapsulation, maintainability and extensibility. Accordingly, the implementation of the *HighPerTimer* library pays particular attention to these aspects. Using the new C++11 programming language standard [20], the library achieves high efficiency and easy code maintainability. Furthermore, regarding the platform-specific aspects, *HighPerTimer* runs on different 64-bit and 32-bit processors of Intel, AMD, VIA and ARM, and considers their general features along with specialties of time keeping.

However, some attention must be paid to obtaining a clean encapsulation of hardware access when using C++. For this encapsulation, the *HighPerTimer* library comprises two header files and two implementation files called *HighPerTimer* and *TimeHardware*. Each of them contains three classes. *HighPerTimer* files contain *HighPerTimer*, *HPTimerInitAndClear* and *AccessTimeHardware* classes, as described below. In *TimeHardware* files, the classes *TSCTimer*, *HPETTimer* and *OSTimer* corresponding to the respective time sources TSC, HPET and the OS source have been implemented. Through an assembly code within the C++ methods, they provide direct access to the timer hardware, initialize the respective timer source, retrieve their time value and are at only *HighPerTimer* class's disposal. Dependencies between the classes are presented in Fig. 1.

 ISBN: 978-1-61208-256-1

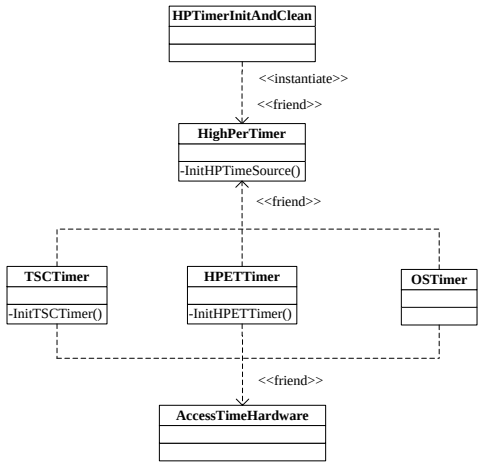

Figure 1. Simplified class diagram of *HighPerTimer* library

TSCTimer, *HPETTimer* and *OSTimer* classes have a "friend" relationship with the *HighPerTimer* class, which means that *HighPerTimer* places their private and protected methods and members at friend classes' disposal. For safety and security reasons, we protect the hardware access from use by application users directly and permit access only from special classes. An *AccessTimeHardware* class provides a limited read-only access to some information on CPU and specific time hardware features, obtained in a protected interface. For example, some advanced users can find out failure reasons of the initialization routine of the HPET device and get a corresponding error message:

```
std::cout << AccessTimeHardware::HpetFailReason();
```

However, all the routines of time handling along with access to the actual timer attributes such as clock frequency are accessed by the library users via the *HighPerTimer* class. For interfacing with other time formats, *HighPerTimer* class provides a set of constructors that sets its object to the given time provided in seconds, nanoseconds or in the native clock ticks of the used time source. Via specific constructor, a time value in a Unix-specific time format [21] can also be assigned to a *HighPerTimer* object. The current time value is retrieved using the following piece of code:

```
// declare HighPerTimer objects
HighPerTimer timer1, timer2;
HighPerTimer::Now (timer1);
// measured operation
HighPerTimer::Now (timer2);
```

Comparison operators allow effective comparison to be performed using the main counter values. Some of these methods are declared as follows:

```
bool operator>= (const HighPerTimer& timer) const;
bool operator<= (const HighPerTimer& timer) const;
bool operator!= (const HighPerTimer& timer) const;
```

The user can also set the value of a timer object explicitly to zero and add or subtract the time values in terms of timer objects, tics, nanoseconds or seconds. Since the main "time" capability of a timer object is kept in the main counter only, the comparison operations between timer objects, as well as arithmetical operations on them, are nearly as fast as comparisons and elementary arithmetical operations on two int64 variables. Recalculations between tics, seconds, microseconds and nanoseconds are only done in the "late initialization" fashion when string representations of the timer object or seconds, microseconds or nanoseconds of the object are explicitly requested via the class interface:

```
// subtract from timer object
HighPerTimer & SecSub (const uint64_t Seconds);
HighPerTimer & USecSub (const uint64_t USeconds);
HighPerTimer & NSecSub (const uint64_t NSeconds);
HighPerTimer & TicSub (const uint64_t Tics);

// add to timer object
HighPerTimer & SecAdd (const uint64_t Seconds);
HighPerTimer & USecAdd (const uint64_t USeconds);
HighPerTimer & NSecAdd (const uint64_t NSeconds);
HighPerTimer & TicAdd (const uint64_t Tics);
```

Assignment operators allow a *HighPerTimer* object to be set from the Unix-format of time values - `timeval` or `timespec` structs [21]. Both of these structures represent time, elapsed since 00:00:00 UTC on 01.01.1970. They consist of two elements: the number of seconds and the rest of the elapsed time represented either in microseconds (in case of `timeval`) or in nanoseconds (in case of `timespec`):

```
struct timeval {
   long tv_sec;  /* seconds */
   long tv_usec; /* microseconds */
}

struct timespec {
   long tv_sec;  /* seconds */
   long tv_nsec; /* nanoseconds */
};
```

Assignment to these structures is also possible with *HighPerTimer* objects through copying or moving:

```
const HighPerTimer & operator= (const struct
timeval & TV);
const HighPerTimer & operator= (const struct
timespec & TS);
const HighPerTimer & operator= (const HighPerTimer
& Timer);
HighPerTimer & operator= (HighPerTimer && Timer);
```

This way, the *HighPerTimer* library provides a fast and efficient way to handle time values by operating main counter value and seconds and nanoseconds values only on demand. It also relieves users from the manual handling of specific two-value structures such as `timeval` or `timespec`.

However, for the whole routine of handling time values, some central parameterization of the library must be performed at the initialization time of the library. Primarily, this is the initialization of the *HighPerTimer* source, which is accomplished on the basis of the appropriate method calls from the *TimeHardware* file. Especially,

 ISBN: 978-1-61208-256-1

InitHPTimeSource() calls *InitTSCTimer()* and *InitHPETTimer()* methods, which attempt to initialize respective time hardware and return true on success or false on failure (see Fig. 1).

Before using any timer object, the following global parameters must be measured and set: the frequency of the main counter as a double precision floating point value and as a number of ticks of the main counter within one microsecond, the value of the shift of the main timer counter against Unix Epoch, the maximum and minimum values of *HighPerTimer* for the given hardware-specific main counter frequency, and the specified HZ frequency of the kernel. The value of HZ is defined as the system timer interrupt rate and varies across kernel versions and hardware platforms. In the context of the library, the value of HZ is used for the implementation of an accurate sleep mechanism, see Section VI. The strict sequence of the initialization process is determined within an additional *HPTimerInitAndClean* service class (see Fig. 1) by invoking corresponding *HighPerTimer* initialization methods through their "friend" relationship. A strict order of initialization of the given global variables must be assured, which is somewhat tricky since all the variables must be declared static and must be initialized before entering the main routine of the application.

Despite the advantage of automatic detection of the appropriate time source, situations sometimes arise when an application programmer prefers to use a different time source than the one automatically chosen at library initialization time. To account for this, a special ability to change the default timer is provided. This change causes a recalculation process for most of the timer attributes:

```
// create variable for a new value of time source
TimeSource MySource;
MySource = TimeSource::HPET;
HighPerTimer::SetTimerSource ( MySource );
```

However, since this change leads to invalidation of all the already existing timer objects within the executed program, this feature should be used with caution and only at the system initialization time, and definitely before instantiation of the first *HighPerTimer* object.

V. TIME FETCHING PERFORMANCE RESULTS

Table I shows the performance results when getting the time values using the *HighPerTimer* library as measured on different processor families. The mean and standard deviation values of the costs of setting a *HighPerTimer* object are shown. For this investigation, time was fetched in a loop of 100 million consecutive runs and set to a *HighPerTimer* object. Since we are interested here in measuring the time interval between two consecutive time fetches only, without any interruption in between, we filter out all outlying peaks. These peaks are most probably caused by process interruption by the scheduler or by an interrupt service routine. Thus, filtering out such outliers allows us to get rid of the bias caused by physical phenomena, which are outside the scope of this investigation.

TABLE I. COSTS OF SETTING TIMER ON DIFFERENT PROCESSORS

Processor (CPU)	Time source	Mean, nsec	St. deviation, nsec
Intel ® Core ™ i7-2600, 1600 MHz	TSC	16.941	0.1231
VIA Nano X2 U4025, 1067 MHz	TSC	38.203	0.3134
Athlon ™ X2 Dual Core BE-2350, 1000 MHz	HPET	1063.3	207.92

The following two examples demonstrate the behavior of *HighPerTimer* sources in more detail and allow a comparison of their reliability and costs depending on the particular processor conditions. Although Table I shows the results for all three processors, later investigations are shown only for less powerful systems. It makes sense to examine in more depth those systems, where for example, TSC is unstable or does not possess *Invariant TSC* flag (see Section III).

In the first case, processor VIA Nano X2 has TSC as a current time source. Costs of time fetching here are about 38 nsec. Since TSC source has the highest priority and has been initialized successfully, the HPET device check is not necessary and so omitted here. Moreover, on this processor, the Linux kernel is also using TSC as its time source and so, within the *clock_gettime()* call, the kernel is also fetching the TSC register of the CPU. Fig. 2 shows the relation between the TSC, OS and HPET timers on this processor. Similarity between TSC and OS costs are seen very clearly. As seen in Table 2, the difference between the mean value of time fetching between OS Timer and TSC Timer is 64 nsec. Each system call with a context switch would last at least ten times longer, thus we can conclude that, on this system, a virtual system call is issued by *clock_gettime()* instead of a real system call with a context switch. HPET source for the library can be set by the static method *HighPerTimer::SetTimerSrouce*. However, we would expect here much slower time operations, as seen in Table 2.

TABLE II. MEAN AND STANDARD DEVIATION VALUES OF HPET, TSC AND OS TIMER COSTS ON THE VIA NANO X2 PROCESSOR

Timer source	Mean, nsec	Standard deviation, nsec
TSC Timer	38.23	0.3134
HPET Timer	598.72	76.015
OS Timer	102.20	0.5253

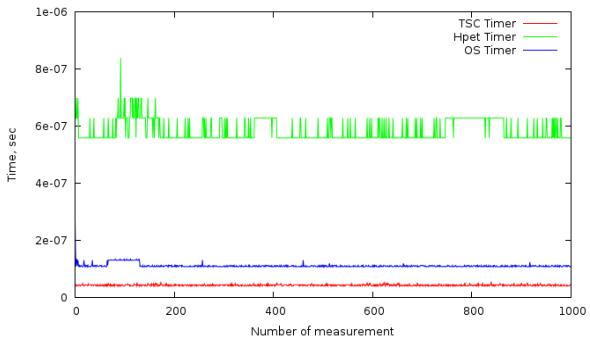

Figure 2. Measurements of TSC, HPET and OS Timer costs on the VIA Nano X2 processor

 ISBN: 978-1-61208-256-1

The next example illustrates another case of a dependence on the OS Timer from the current time source. For the processor AMD Athlon X2 Dual Core, the TSC initialization routine fails because TSC is unstable here. However, since the HPET device is accessible, there are two more options for the time source for *HighPerTimer* – HPET or OS Timers - and it is necessary to check the mean costs of getting the ticks of both timers.

Although the mean value of time fetching for TSC can be significantly lower than for HPET, the *HighPerTimer* library considers the TSC to be a non-stable, unreliable time source since the *Invariant TSC* flag (see Section III above) is not available and the TSC constancy is not identified by additional library checks. So, it must be assumed that TSC frequency changes from time to time due to power saving or other techniques of the CPU manufacturers. In the next step, HPET and OS Timer characteristics must be considered. The difference between the mean values of HPET and OS Timer is about 54.1 nsec, which is not enough for a system call with a context switch. Thus we conclude that *clock_gettime()* also uses the HPET timer and passes it to the user via a virtual system call. However, to provide an appropriate level of reliability, we also evaluate numbers through their deviation values. For this evaluation, a threshold for the difference of mean values was chosen. When the difference of the mean values of HPET and OS Timer is no more than 25%, we also take into account standard deviation values of time fetching and so check the temporal stability of the considered time source. Consequently, when the mean time fetching value of the two time sources is similar, the *HighPerTimer* library would give precedence to the time source with a less standard deviation of the time fetching costs.

TABLE III. MEAN AND STANDARD DEVIATION VALUES OF HPET, TSC AND OS TIMER COSTS ON THE AMD ATHLON PROCESSOR

Timer source	Mean, µsec	Standard deviation, µsec
TSC Timer	0.0251	0.0015
HPET Timer	1.0633	0.2079
OS Timer	1.1174	0.3743

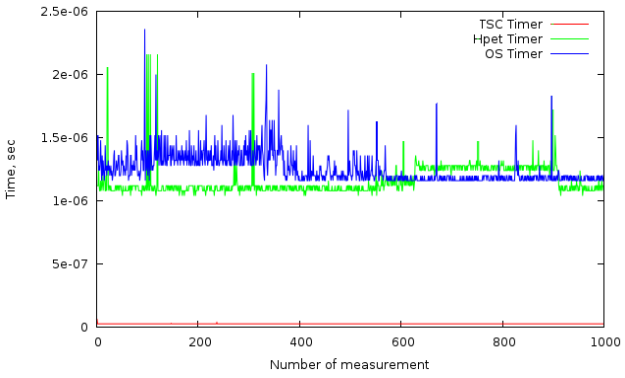

Figure 3. Measurements of TSC, HPET and OS Timer costs on the AMD Athlon processor

VI. PRECISE PROCESS SLEEPING ASPECTS

For process sleeping or suspension, Linux provides the sleep function (implemented in the standard C library). Dependent on the sleep duration, the function either suspends from the CPU or waits in the busy-waiting mode (sometimes also called spinning wait). However, measurements performed in this work revealed that the sleep function of the standard C library misses the target wake-up time by more than 50 microseconds on average. Such an imprecision however is unacceptable for high-accuracy program sleeps. By comparison, pure busy wait implementations within an application miss the target return time by about 100 nanoseconds, but keep the CPU busy throughout the wait time.

Unlike the C library's sleep call, the sleep of the *HighPerTimer* library combines these two ways of sleeping. It has very short miss times on waking up with a minimum CPU utilization at the same time. This improvement provides a big competitive advantage over the predecessor solutions.

HighPerTimer provides a wide range of functions for making a process sleep. For example, the user can define the specific sleep time, given purely in seconds, in microseconds or nanoseconds. A process suspension with a nanosecond resolution can be done as follows:

```
HighPerTimer timer1;
uint32_t SleepTimeNs(14500);
// sleep in nanoseconds
timer1.NSecSleep(SleepTimeNs);
```

Alternatively, the time value of a *HighPerTimer* object can be set to a specific time value at which the process shall wake up. On the call of *SleepToThis()*, the process will then be suspended till the system time has reached the value of that object :

```
//declare timer object equaled to 10 sec, 500 nsec
HighPerTimer timer2 (10, 500);
timer2.SleepToThis();
```

Table IV shows the precision of sleeps and busy-waits using different methods. Miss values are here the respective differences between the targeted wakeup time and real times of wakeups measured in our tests. However, the miss values of sleep times heavily depend on the fact, whether target sleep interval was shorter or longer, than time between two timer interrupts. So, Table IV consists of two parts – one where sleep time is longer then 1/HZ, and one where it is less than 1/HZ. Thus, the left column shows results for waits lasting longer than a period of two kernel timer interrupts. The right column shows the results for the scenario, in which the sleep call lasts less than the interval between two kernel timer interrupts. These measurements have been performed on the Intel Core–i7 processor. Other than in measurements from Section V, in this case it makes sense to show results on a more stable and powerful system. Moreover, it was expected that the accuracy of sleeps would be higher on the newer Linux kernel versions where time handling has been changed significantly. However, as the

 ISBN: 978-1-61208-256-1

measurements below show, these kernel changes are still not sufficient.

In this test scenario, we have issued the respective sleep method within a loop of 100000 sleeps with different sleep times between 0.25 sec and 1 µsec, and then the mean value of the sleep duration miss has been calculated.

TABLE IV. THE COMPARISON OF MISS VALUES OF DIFFERENT METHODS OF SLEEPING, PERFORMED WITH TSC ON THE INTEL CORE –I7 PROCESSOR

	Sleep time >= 1/HZ	Sleep time < 1/HZ
	Mean miss, µsec	**Mean miss, µsec**
System sleep	61.985	50.879
Busy-waiting loop	0.160	0.070
HighPerTimer sleep	0.258	0.095

The above experiment took about 830 minutes, so the upper limit of the range for sleep time value was reduced to 0.25 sec. The chart in Fig. 4 demonstrates more detailed results of this experiment and shows the dependency of miss against the target sleep time in dependence from sleep duration. To track this dependency more deeply, here the range of sleep time value was increased and is taken between 10 sec and 1 µsec.

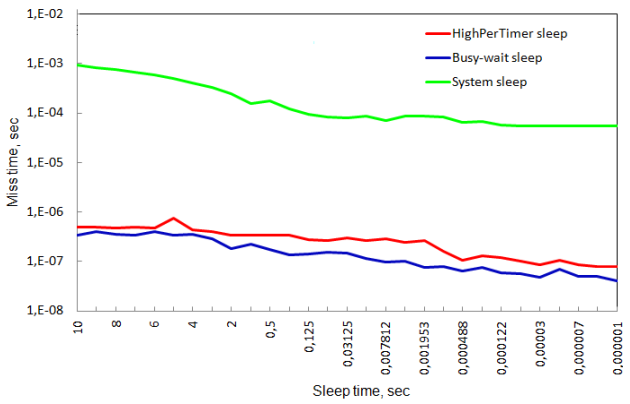

Figure 4. Dependency of miss on the target time from sleep time, performed with TSC on the Intel Core –i7 processor, HZ = 1000

In the next step, we measured the CPU consumption of the respective sleep routine. In the busy-waiting loop, the total CPU consumption during the test achieves, as expected, almost 100%. For the sleep function of the standard C library, it tends to zero. In the case of sleeping using the *HighPerTimer* library, the overall CPU consumption during the test was 1.89%, which can be considered as a good tradeoff between precision of waking up time and CPU consumption during the sleep.

VII. CONCLUSION AND FUTURE WORK

In accordance with the requirements of advanced high-speed data networks, we showed an approach for the unified high performance timer library that successfully solves two significant problems. Firstly, *HighPerTimer* allows identification of the most efficient and reliable way for time acquisition on a system and for avoiding system calls invocation on time acquisition. Secondly, it solves the problem of precise sleeping aspects and provides new advanced sleeping and resuming methods.

The *HighPerTimer* library has the potential to become widely used in estimation network packet dynamics, particularly when conducting high-accuracy and high-precision measurements of network performance. At this stage, the integration of the suggested solution into the appropriate tool for distributed network performance measurement [22] is in progress. Moreover, to the next steps, the better support of the ARM processor will be addressed. Since the ARM processor possesses neither HPET nor TSC, the only way to support ARM at this stage is to select OS Timer. Presumably, an invocation of the initial ARM system timer can afford to save several additional microseconds and improve the timer accuracy.

REFERENCES

[1] R. Takano, T. Kudoh, Y. Kodama, and F. Okazaki, "High-resolution Timer-based Packet Pacing Mechanism on the Linux Operating System," IEICE Transactions on Communication, Tokyo, vol. E94-B, no. 8, pp. 2199-2207, Nov., 2011.

[2] J. Micheel, S. Donnelly, and I. Graham, "Precision time stamping of network packets," Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement, San Francisco, CA, USA, pp. 273-277, Nov., 2001.

[3] H. Hu, "Untersuchung und prototypische Implementierung von Methoden zur hochperformanten Zeitmessung unter Linux," (German), Bachelor Thesis, Anhalt University of Applied Sciences, Koethen, Germany, Nov., 2011.

[4] Performance monitoring with the RDTSC instruction. URL: http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf, retrieved: Jan., 2013.

[5] E. Corell, P. Saxholm, and D. Veitch, "A user friendly TSC clock," Proc. PAM, Adelaide, Australia, pp. 141-150, Mar., 2006.

[6] S. Siddha, V. Pallipadi, and D. Ven, "Getting maximum mileage out of tickles," in Proc. of the 2007 Linux Symposium, pp. 201-208, 2007.

[7] Intel IA-PC HPET (High Precision Event Timers) Specification. URL: http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf, retrieved: Jan., 2013.

[8] D. Kang, W. Lee, and C. Park, "Kernel Thread Scheduling in Real-Time Linux for Wearable Computers," ETRI Journal, Daejeon, Korea, vol. 29, no. 3, June 2007, pp. 270-280, doi: 10.4218/etrij.07.0506.0019.

[9] P. Orosz and T.Skopko, "Performance Evaluation of a High Precision Software-based Timestamping Solution," International Journal on Advances in Software, ISSN 1942-2628, 2011, vol. 4, no. 1, pp.181-188.

[10] T. Gleixner and D. Niehaus, "Hrtimers and Beyond: Transforming the Linux Time Subsystems," The Linux Symposium, Ottawa, Canada, 2006, vol. 1, pp. 333-346.

 ISBN: 978-1-61208-256-1

[11] D. Kachan, E.Siemens, and H.Hu, "Tools for the high-accuracy time measurement in computer systems," (Russian), 6th Industrial Scientific Conference "Information Society Technologies", Moscow, Russia, 2012, pp.22-25.

[12] Intel Trace Collector Reference Guide, p. 5.2.5. URL: http://software.intel.com/sites/products/documentation/hpc/ics/itac/81/ITC_Reference_Guide/ITC_Reference_Guide.pdf, retrieved: Jan., 2013.

[13] D. Grove and P. Coddington, "Precise MPI Performance Measurement Using MPIBench," Proc. of HPC Asia, pp. 1-14, Gold Coast, Australia, 2001.

[14] The Dag project. URL: http://www.endace.com, retrieved: Jan., 2013.

[15] A. Pásztor and D. Veitch, "PC based precision timing without GPS," The 2002 ACM SIGMETRICS international conference on Measurement and modeling of systems, Marina Del Rey California, USA, vol. 30, no. 1, pp. 1-10, June, 2002, doi: 10.1145/511334.511336.

[16] Intel 64 and IA-32 Architectures, Software Developer's Manual, vol. 3B 17-36. URL: http://download.intel.com/products/processor/manual/253669.pdf, retrieved: Jan., 2013.

[17] J. Dike, "A user-mode port of the Linux kernel," USENIX Association Berkeley, pp. 63-72, California, USA, 2000.

[18] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," Proc. of the ISOC Symposium on Network and Distributed System Security, pp.19-34, Feb., 2000.

[19] J. Corbet, "On vsyscalls and the vDSO. Kernel development news," Linux news site LWN. URL: http://lwn.net/Articles/446125/, retrieved: Jan., 2013.

[20] Online C++11 standard library reference, URL: cppreference.com, retrieved: Jan., 2013.

[21] GNU Operating System Manual, "Elapsed Time". URL: http://www.gnu.org/software/libc/manual/html_node/Elapsed-Time.html, retrieved: Jan., 2013.

[22] E.Siemens, S.Piger, C. Grimm, and M. Fromme, "LTest – A Tool for Distributed Network Performance Measurement," Consumer Communications and Networking Conference, Las Vegas, NV, USA, 2004, pp. 234-244, doi: 10.1109/CCNC.2004.1286865.

 ISBN: 978-1-61208-256-1