
© Copyright 2008 IEEE
Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
High-Precision Relative Clock Synchronization Using Time Stamp Counters

Guo-Song Tian, Yu-Chu Tian and Colin Fidge
Faculty of Information Technology,
Queensland University of Technology, Brisbane, QLD 4001, Australia

Abstract

In this paper we show how to use a computer processor’s Time Stamp Counter register to provide a precise and stable time reference, via a high-precision relative clock synchronization protocol. Existing clock synchronization techniques, such as the Network Time Protocol, were designed for wide-area networks with large propagation delays, but the millisecond-scale precision they offer is too coarse for local-area applications such as instrument monitoring systems, high-quality digital audio systems and sensor networks. Our new clock synchronization technique does not require specialized hardware but instead uses the Time Stamp Counter already available in the widely-used Intel Pentium processor. Experimental results show that we can achieve a synchronization precision in the order of 10 microseconds in a small-scale local area network using TSC registers, which is much higher than can be achieved by using a computer processor’s Time-Of-Day clock.

1 Introduction

With the emergence of new time-critical distributed computing paradigms, such as distributed control converters, human-robot physical interaction and ad hoc networks composed of large numbers of disposable sensors, clock synchronization is again becoming a critical issue in distributed computing environments [1, 3, 6, 9, 14, 15]. In a typical business application network, the only synchronization requirement is the need to keep clocks in a Wide-Area Network (WAN) loosely synchronized with an absolute Time-Of-Day reference. By contrast, distributed control systems require strong, relative clock consistency in a Local-Area Network (LAN) [4].

Relative clock synchronization refers to the mechanisms and protocols used to maintain mutually-consistent clocks in a coordinated network of computers. That is, relative clock synchronization focuses on the time difference between clocks, not the offset of each clock against absolute time (sometimes called ‘wall clock time’). The accuracy requirements for relative clock synchronization in control systems are of the order of millisecond or even sub-millisecond precision. For instance, real-time controllers typically assume that important events occur at predetermined times, and the system’s calculations might be wrong if sensors are not sampled at the assumed rate or if signals are not sent to actuators at the expected times.

High-precision clock synchronization can be achieved either by hardware or software clocks. The clocks may be Time-Of-Day (TOD) clocks using an oscillator that is consistent with Coordinated Universal Time or a software clock using other absolute time sources such as Internet time servers or a satellite radio clock. In this paper, however, our clock synchronization method is relative only, so does not rely on an external high-precision hardware timer.

Generally, the precision of network clock synchronization depends on two aspects: determining the one-way transmission delay of timestamp messages and the precision of a local clock. The well-known four-timestamp offset-calculation mechanism of the Network Time Protocol (NTP) is widely used by synchronization protocols to improve the accuracy of estimated one-way transmission delays. Networks using NTP-like synchronization mechanisms typically provide precision of the order of one millisecond [12]. In our work, the adopted timestamp mechanism is derived from NTP.

One of keys to achieving high-precision relative clock synchronization is the use of high accuracy local clocks because the goal of the synchronization is to correct local clocks accurately before they drift out of acceptable range. Unfortunately, Time-Of-Day clocks rely on potentially unstable crystal oscillators [9]. Compared with an oscillator, however, we note that a computer processor’s Time Stamp Counter (TSC) register has several advantages for achieving high precision relative clock synchronization. The TSC register counts processor cycles and is normally used for measuring short time intervals. Its time resolution is much higher than an oscillator’s. For example, for a processor running at a frequency of 800MHz, its TSC register can provide a time resolution of 1.25 nanoseconds. Also, the time required to read from a TSC register is far less than that of
reading from the computer’s TOD clock. Finally, a computer’s Central Processing Unit (CPU) frequency is usually more stable than that of the oscillator used to drive the TOD clock.

In this paper we present a high-precision relative clock synchronization protocol with a master-slave synchronization structure. The protocol is intended to support high-precision real-time calculations in Local-Area Networks, rather than the usual low-precision timestamping mechanism supported in Internet applications. Since we are interested in intra-network clock consistency, rather synchronizing clocks against an absolute time standard, the protocol does not require a high-precision time server whose precision has to be guaranteed by additional hardware clocks [3, 12]. The protocol is suitable for use with networks of processors that provide access to a TSC register. Thanks to the stability of the TSC register, the precision achieved by our protocol can be much greater than that of synchronization methods using Time-Of-Day functions, as we demonstrate by experiments performed under a simple synchronization structure.

2 TOD clock and TSC clock characterization

This section introduces the characteristics of both Time-Of-Day and Time Stamp Counter clocks, and compares the performance of timing measurement using a TOD clock and a TSC-based clock. Also the effects of clock skew are measured using TSC-based clocks.

Maintaining a TOD clock requires the computer to generate a clock-tick signal at a predetermined rate. The source of a computer’s system timer is a basic uncompensated quartz crystal oscillator whose frequency is 14.31818 MHz. Dividing the basic frequency by 12 gives 1.19318 MHz (thus the period is 0.8381 microseconds), which is the clock frequency used by the user’s system timers [2].

The precision of a TOD clock is affected by not only the resolution of the computer’s system timer but also the periodic timer interrupt. The Time-Of-Day function in general-purpose computers is commonly implemented using clock readings from an uncompensated quartz crystal oscillator and counter, which delivers a pulse train with a period ranging from 10 to 1 milliseconds. Each pulse causes a timer interrupt, which increments a software logical clock variable by a fixed value scaled in microseconds or nanoseconds. Therefore, the precision of a TOD clock may be greater than 0.8381 microseconds, even as much as 10 milliseconds for conventional Linux systems, which represent wall time as two 32-bit words in seconds and microseconds/nanoseconds from UTC midnight, 1st of January 1970, with no provision for leap seconds [13].

In this paper, a TSC-based clock is defined as a software clock based on a computer processor’s TSC register. Many processors contain a timer that operates at the clock cycle level. The timer is a special register that gets incremented every single clock cycle. Although there is no uniform, platform-independent interface by which programmers can make use of these counters, it is easy to create a program interface for any specific machine with just a small amount of assembly code. The Time Stamp Counter is a cycle counter used in the P6 microarchitecture (the PentiumPro and its successors). It is a 64-bit unsigned word, which counts the number of cycles since the CPU was powered up or last reset. Although the CPU chip’s frequency is affected by external factors, such as age and operating temperature, higher CPU speeds allow finer time resolution of this counter.

The current “time” of a TSC clock, relative to the CPU’s last start-up or reset, can be calculated easily from the value of the TSC register $T$ and the measured CPU frequency $f_m$ as follows.

$$tt = \frac{T}{f_m}$$ (1)

The value of the TSC register can be read by an ‘rdtsc’ instruction. Unfortunately in non-real-time operating system, it is hard to measure the CPU’s frequency exactly without an external high-precision timer, because of effects such as context switching, caching and branch predication. For example, the precision achieved by a coarse CPU frequency measurement may be as much as 1.0% [2]. Therefore, a (relative) TSC-based clock $tt$ and an (absolute) TOD clock $t$ may run at a different speeds.

The precision of a software clock is determined by at least two factors: the time cost of reading the hardware clock and the hardware clock’s intrinsic frequency error. The time required to read a hardware clock can be affected by factors such as the scheduling characteristics of the operating system and the CPU’s computational capabilities. The intrinsic frequency error of the hardware clock further deteriorates the precision of the derived software clock, and accumulated errors cause clock drift.

To inform our synchronization protocol, we need to understand the magnitude of these errors in the system configuration of interest. In the remainder of this section we present the results of experiments we conducted to measure how long it takes to read from hardware clocks and the effects of clock drift. Table 1 summarises the configurations

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Intel Pentium III 800MHz 256M/512M SDRAM Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating systems</td>
<td>Fedora 3/Linux kernel 2.6.12.1</td>
</tr>
<tr>
<td>Network adapter</td>
<td>Network adapter 3COM 3c59x Fast Ethernet Card</td>
</tr>
<tr>
<td>Ethernet switch</td>
<td>Baystack 303 Ethernet Switch</td>
</tr>
</tbody>
</table>

Table 1. Test configuration
The relative clock drift between two clocks on the communicating machines was measured as the time difference between receiving timestamp $t_R$ and sending stamp $t_S$ for each periodic message. Figure 2 shows that the time difference between two TOD clocks grows linearly, and its threshold behavior is obvious. Figure 3, which uses a finer scale, reveals that the time difference may jump by more than 1 ms during a short time (less than 0.2 s). The same experiment was then implemented using timestamps derived from TSC register readings. As shown in Figure 4, the TSC-based clock drifts smoothly with minimal threshold effects.

In summary, relative clock drift when using a TSC-based clock would be caused by errors in the measured CPU frequency. Because a modern microprocessor’s CPU frequency is almost constant, we expect that TSC-based clock drift caused by intrinsic frequency error will be strictly linear.
3 High precision relative clock synchronization protocol

This section discusses the design and implementation of our relative clock synchronization protocol, based on the assumption that every computer processor in the synchronization structure is equipped with a Time Stamp Counter register. In this section, we assume that \( t_x \) is the absolute time, and \( T_x \) and \( t*t_x \) are the ticks of the TSC register and the current time of a TSC-based clock, respectively, both relative to the CPU’s last startup or reset.

3.1 Synchronization structure

We adopt a master-slave structure for the clock synchronization protocol. The distributed system consists of a set of nodes interconnected via an Ethernet switch, as shown in Figure 5. There are two kinds of nodes, a single master node (the time server) and several slave nodes (the time clients). All slave nodes are to be synchronized to the master node.

Normally in clock synchronization protocols it is required that root time servers are connected to a reliable external time source, such as a satellite radio clock or telephone network time server [12]. Unlike other clock synchronization protocols, however, our relative clock synchronization protocol does not require a high-precision hardware absolute time source accessible by the master node.

3.2 Synchronization model

The synchronization model is based on the usual four-timestamp mechanism of the Network Time Protocol as shown in Figure 6. This commonly-used mechanism measures the transmission delay between communicating nodes and uses this to estimate the offset between their respective clocks, in order to determine the error in the client node’s
clock with respect to the time server’s clock. At predetermined intervals, a client sends a request to the time server and waits for a response. This exchange of messages results in four clock timestamps, \( t_{CS} \) at the first message’s sending time, \( t_{SR} \) at its receiving time, \( t_{SS} \) at the second message’s sending time, and \( t_{CR} \) at its receiving time. Timestamps \( t_{CS} \) and \( t_{CR} \) are read from the client node’s clock and timestamps \( t_{SR} \) and \( t_{SS} \) are read from the server node’s clock. The server’s timestamps are sent to the client in the second message. The client then uses the four timestamps to calculate the clock offset and roundtrip message delay relative to the server. The client can then reset its own clock to compensate for any difference with the server’s clock.

The timestamping mechanism is designed based on an assumption that the forward and backward communication delays are symmetric. The time difference \( t_d \) between the time \( t(C) \) on the client and the time \( t(S) \) on the server can be computed as follows.

\[
t_d = t(C) - t(S)
\]
\[
= \left( \frac{(t_{CS} + t_{CR}) - (t_{SS} + t_{SR})}{2} \right)
\]
\[
= \frac{1}{2} \left[ \frac{(T_{CS} + T_{CR})}{f_C} - \frac{(T_{SS} + T_{SR})}{f_S} \right]
\]

Here \( f_C \) and \( f_S \) are the CPU frequencies of the client and the server.

Similarly, using TSC-based software clocks the time difference \( tt_d \) between the time \( tt(C) \) on the client and the time \( tt(S) \) on the server can be computed as follows. This equation is the basis of our synchronization protocol.

\[
\begin{align*}
\quad tt_d &= tt(C) - tt(S) \\
&= \frac{1}{2} \left[ \frac{(T_{CS} + T_{CR})}{f_{mC}} - \frac{(T_{SS} + T_{SR})}{f_{mS}} \right]
\end{align*}
\]

Here \( T_{CS}, T_{SR}, T_{SS} \) and \( T_{CR} \) are timestamps (the ticks of TSC registers) on the server and the client corresponding to \( t_{CS}, t_{SR}, t_{SS} \) and \( t_{CR} \), and \( f_{mC} \) and \( f_{mS} \) are the measured values of the CPU frequencies of the client and the server, respectively.

Figure 7 shows the time difference \( t_d \) between \( t(C) \) and \( t(S) \) that are absolute time at the client and the server, and the time difference \( tt_d \) between \( tt(C) \) and \( tt(S) \) that are the time of TSC-based clocks at the client and the server, respectively. We assume that there is no intrinsic frequency error in both clocks in the client and the server. So the absolute time difference \( t_d \) is flat using absolute time clocks. Unlike absolute time clocks, TSC-based clocks in the server and the client may run at different speeds, and the relative time difference \( tt_d \) between them increases with time. When measured clock frequency is equal to the real clock frequency, \( tt_d \) is equal to \( t_d \). The error in measured CPU frequencies and its impact on the precision of relative clock synchronization will be discussed in Section 4.

3.3 Implementation

In this section we describe our prototype clock synchronization implementation, based on Time Stamp Counter time differences calculated as per Equation 3.

3.3.1 Operating system

The prototype protocol is currently implemented in a conventional Linux kernel, rather than a specialised real-time operating system. The kernel, therefore, guarantees no bounds on interrupt servicing latencies. The frequency of timer interrupts in conventional Linux is 100 Hz, i.e., the tick interval is 10 ms. Both transmission and reception of timestamp messages are interrupt-driven events. Variations in interrupt latencies produce errors in the delay estimates that the synchronization protocol uses to coordinate clocks. When the CPU is lightly loaded, the error in time measurements caused by variability in the tick interval is very small, even negligible. Therefore, with some care, the precision achieved for relative clock synchronization using conventional Linux for our experiments was considered satisfactory.

3.3.2 The TSC-based clock

To compute the time for a TSC-based clock, an rdtsc instruction is used to read the local TSC register and a kernel
module was developed for measuring the CPU frequency.

When booting up a Linux system an attempt is made to calibrate the CPU frequency by comparing it with the Programmable Interval Timer (PIT). The calibration result is in the format of TSC ticks per microsecond. This 50 ms calibration is coarse, with a precision of 100 parts per million because it is flawed by the I/O delays in accessing the PIT. The measured CPU frequency can be read from the file '/proc/cpuinfo'.

Using the same method, we programmed a CPU frequency test as a module that runs in the kernel. The module performs a 50 ms test repeatedly, 100 times. The fluctuation range of the measured CPU frequency during 100 tests is about 50 PPM. Although the program runs in kernel space and disables some interrupts, such as the speaker’s, the expected 50 ms interval between two consecutive readings of the TSC register may be prolonged by enabled interrupts. Therefore, the maximum measured CPU frequency among 100 tests is close to the true CPU frequency, and is used as the measured CPU frequency \(f_m\) to compute the time of TSC clock \(tt\).

<table>
<thead>
<tr>
<th>Receiving Timestamp (64bit)</th>
<th>Sending Timestamp (64bit)</th>
<th>Time Difference (64bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>Reserved</td>
<td>32bit</td>
</tr>
</tbody>
</table>

**Figure 8. Synchronization message’s format**

### 3.3.3 Synchronization message’s format

We send all synchronization messages via the UDP protocol under Linux, because this protocol is a better choice for real-time applications than the TCP protocol [1,5]. Figure 8 shows the synchronization message including two timestamps, a message identifier and information fields. Two timestamps are 64-bit words whose values are set using TSC register values. The message’s identifier is used to match timestamps in the time server and time client, and detect packet loss. The time difference is computed in the client. The time server learns of the computed time difference via the time difference field received from the time client. Finally, the four ‘reserved’ bytes will be used to change the frequency of message transmission in terms of achieved precision of synchronization in future work.

### 3.3.4 Operating system modification

As a multi-tasking computing environment, conventional Linux provides two modes of background process, user space and kernel space. For accuracy, the synchronization protocol relies on simple kernel-space routines for the message timestamps.

Our program interfaces with the kernel through standard Linux system calls. The sending and receiving timestamps are recorded in the NIC driver, via an rdtsc instruction, and the timestamps are passed to the user space by an ioctl() call. The entire modification is small, requiring only a few tens of lines of code. The timestamping mechanism can be implemented in Linux kernel 2.4 and 2.6.

### 4 Error analysis

As well as implementing the synchronization process, it is important that we understand the worst-case accuracy of the result. In this section we therefore analyze the potential errors in the time difference measurements that are used in the synchronization protocol.

As shown in Figure 6, \(D_S\) and \(D_R\) are the one-way transmission delay of the two synchronization messages. Let \(D_{S-R}\) denote \(D_S - D_R\). Then the absolute time difference between the client and server is related to the four timestamps of the message and one-way transmission delay of the messages as follows.

\[
t_d = \frac{1}{2} \left[ \left( T_{CS} + T_{CR} \right) \left( T_{SS} + T_{SR} \right) \right] + \frac{D_{S-R}}{2} \tag{4}
\]

We assume that the errors in these measurements are \(\Delta f_{mC}\) and \(\Delta f_{mS}\). Let \(\Delta f_X\) be CPU frequency \(f_X\) minus measured value of CPU frequency \(f_{mX}\). Then the measured TSC-based time difference between the client and time server can be computed by the client using the following formula.

\[
tt_{md} = \frac{1}{2} \left( \frac{T_{CS} + T_{CR}}{f_{mC}} - \frac{T_{SS} + T_{SR}}{f_{mS}} \right) = \frac{\frac{\Delta f_{c}}{f_{mC}} - \frac{\Delta f_{s}}{f_{mS}}}{1 + \frac{\Delta f_{c}}{f_{mC}}} \left( \frac{f_{mC}}{D_{S-R}} + t_{mRRT} \right) \tag{5}
\]

where the round trip time \(t_{mRRT}\) measured by the client is the difference between two timestamps using TSC register at the client \((T_{CR} - T_{CS})\), divided by measured CPU frequency of the client \(f_{mC}\).

We observe that one term of the measured time difference,

\[
\frac{\Delta f_{c}}{f_{mC}} - \frac{\Delta f_{s}}{f_{mS}} \left( \frac{T_{CS}}{f_{mC}} \right) + \frac{\Delta f_{s}}{f_{mS}} \left( \frac{T_{SS}}{f_{mS}} \right)
\]

\[
= \frac{\Delta f_{c}}{f_{mC}} - \frac{\Delta f_{s}}{f_{mS}} \left( \frac{T_{CS}}{f_{mC}} \right)
\]

is time-varying if the CPU frequencies of the server and the client, and their measured values are constant. Therefore, the measured time difference is expected to be proportional to the time of TSC-based clock $T_{CS}/f_{mC}$ at the client. The slope of the drift is

$$K = \frac{\Delta f_c/f_{mC} - \Delta f_S/f_{mS}}{1 + \Delta f_c/f_{mC}}.$$  

It is clear that the drift results from the different speeds at which the two TSC-based clocks in the client and the server run. The difference between speeds is called clock skew [7]. When the measurement period is long enough, this non-zero slope can be estimated accurately by some clock frequency synchronization methods such as a skew estimation algorithm [7, 11, 16, 17]. For simplicity, linear regression is used for clock skew correction in this paper.

When the measured CPU frequency of the time client is corrected based on the estimated slope, the speeds of the two TSC-based clocks in the client and server are synchronized. We assume that the time of the two TSC-based clocks in the client and the server are $tt(C)$ and $tt(S)$. Figure 9 shows the process of clock skew correction. Two TSC-based clocks in the client and the server run at different speeds, so measured time difference $tt_d$ drifts. Also the measured CPU frequency of the client is regulated using the estimated slope, so the time of the TSC-based clock in the client is changed from $tt(C)$ to $tt'(C)$ correspondingly, which is computed by the measured CPU frequency and the value of TSC register. After clock skew correction, the drift caused by the clock skew is eliminated and the measured time difference becomes constant from $tt_d$ to $tt'_d$, as shown in Figure 9.

![Figure 9. TSC clock of time client correction](image)

A simple experiment was designed to verify the expected drift, and validate the clock skew correction. Similar to the experiment in Section 2.2, two machines were interconnected via Ethernet and a switch. At a predetermined interval (usually 100 ms) one machine transmits UDP packets to the other machine continuously, 5000 times. In terms of the timestamps mechanism, four timestamps were acquired by the TSC-based clock at the Network Interface Card driver for accuracy.

Measured time difference $tt_{md}$ is computed by four timestamps. The TSC-based clock at time client is set to zero when the experiment starts. Figure 10 shows the drift caused by the clock skew between the client and the server is linear as expected and Figure 11 indicates the drift caused by the clock skew is almost eliminated. After the clock skew is corrected, the measured time difference $tt_{md}$ between the time client and time server can be estimated by the following formula.

$$tt_{md} = \left[\frac{\Delta f_c}{f_{mC}} - \frac{\Delta f_S}{f_{mS}} \right] t_{mRRT} + \frac{f_S}{f_{mS}} t_d - \frac{f_S}{f_{mS}} \left(\frac{D_S - R}{2}\right)$$  

The measured time difference $tt_{md}$ can be used to estimate the absolute time difference $t_d$. Its accuracy, however, cannot be guaranteed, particularly when the absolute time difference $t_d$ is very large. For example, assume that the error in the measured CPU frequency is 50 PPM, and the absolute time difference is 10000 s, then the error in the
measured time difference may be as much as 500 ms. Fortunately, the deviation of absolute time difference $\Delta t_d$ can be estimated accurately by the deviation of measured time difference $\Delta t_{md}$ when the synchronization update interval is small enough. The error $E_{TSC}$ between $\Delta t_{md}$ and $\Delta t_d$ is measured as the precision of synchronization, and can be computed by the following formula.

$$E_{TSC} = \frac{\Delta f_{mc}}{f_{mc}} \Delta t_{md} - \frac{\Delta f_{md}}{f_{md}} \Delta t_d$$

$$(7)$$

Because the measured CPU frequencies of the time client and server have been coordinated, the main source of error is asymmetric one-way transmission delays. This error $E_{TSC}$ can be estimated as follows.

$$E_{TSC} \approx -\frac{f_S}{f_mS} \frac{\Delta D_{S-R}}{2}$$

$$(8)$$

By contrast, when the software clocks in the client and the server are TOD clocks, the precision of synchronization using a timestamp exchange mechanism is $\frac{\Delta D_{S-R}}{2}$. The practical performance comparison of relative clock synchronization using TOD clocks versus TSC-based clocks is discussed in the next section.

5 Performance tests

In this section we summarise the results of several experiments conducted to assess the performance of our clock synchronization protocol.

5.1 Test set-up

The hardware configuration of the time client and server is shown in Table 1, and the structure of the computers used in the experiments was the same as the synchronization structure, as explained in Section 3.1. Beside the time client and server, one computer was used as a traffic generator, running Windows XP on on Intel Pentium 4 processor with a 2.0 GHz CPU and 1 GB DDRAM memory. To reduce interference from other software processes, most of the background processes and tasks were shut down during the experiments. The sending and receiving timestamps were recorded at the NIC driver via the modified Linux kernel. The software clocks on the time server and time clients were TSC register-based as defined in Section 2.

5.2 The precision of relative clock synchronization

The two main sources of synchronization error are asymmetric one-way transmission times and the intrinsic frequency errors of the CPUs, as per the error analysis in Section 4. In this section we analyze the precision achieved by relative clock synchronization under different traffic patterns.

Two experiments were designed to analyze the impact of two kinds of transmission delay on the precision of relative clock synchronization: delay in the network device (switch), and delay in the network interface cards (NIC). Delays in the NIC result from traffic jams when the NIC is sending or receiving too many packets. It is related to the processing capacity of the NIC and the operating system. Because the time server can easily modulate packet transmission by methods such as traffic smoothing [8], we only analyzed the delay in the NIC when the time server is receiving packets from the traffic generator.

The deviation $\tilde{E}$ between two consecutive measured clock differences $t_{md}$ is defined as the precision of relative clock synchronization. In experiments, a time client and its time server were interconnected via an Ethernet switch. The traffic generator was used to simulate multiple time clients’ environments, and different traffic patterns. The time client initiates synchronization requests and periodically sends a request message to its time server. Once the time server receives the request message, it responds with a message that contains two TSC timestamps. Because the timestamp information is caught at the NIC, not at the application layer, it has to wait for transmission until the next cycle. That is, the last two TSC timestamps in reply messages are recorded. Finally, the time difference is computed in the time client, and then transmitted to the time server if necessary.

In all experiments, the message interval was 100 ms, and the number of messages was 10000. Because the round trip time of a message in a LAN is normally less than 100 ms and the traffic stream caused by the designed protocol is almost negligible (less than 1 Kbps), this message interval is practical in a LAN. Because some time clients may need to synchronize with the time server at the same time, the traffic generator sent multiple requests to the time server to simulate multiple time clients’ environments during the experiments. The traffic load was about 33 Kbps.

In the first experiment, the deviation $\tilde{E}$ between two consecutive measured clock differences was measured when the switch is under traffic load (9.8 Mbps) and no traffic load. Their confidence limits under different confidence intervals are shown in Table 2. The precision of relative clock synchronization is about 10 $\mu$s with an above 95% confidence interval whether the switch is under traffic load or not.

The second experiment measured the synchronization
### Table 3. Precision achieved when the time server is under different traffic loads

<table>
<thead>
<tr>
<th>Traffic load</th>
<th>Packet loss</th>
<th>CL ±10 us</th>
<th>±20 us</th>
<th>±50 us</th>
</tr>
</thead>
<tbody>
<tr>
<td>117 kbps</td>
<td>0</td>
<td>CI 94.76%</td>
<td>97.00%</td>
<td>97.30%</td>
</tr>
<tr>
<td>286 kbps</td>
<td>0</td>
<td>CI 91.15%</td>
<td>94.78%</td>
<td>95.10%</td>
</tr>
<tr>
<td>523 kbps</td>
<td>0</td>
<td>CI 81.53%</td>
<td>83.61%</td>
<td>84.47%</td>
</tr>
<tr>
<td>1.1 Mbps</td>
<td>0</td>
<td>CI 69.52%</td>
<td>71.46%</td>
<td>72.86%</td>
</tr>
<tr>
<td>2.3 Mbps</td>
<td>0</td>
<td>CI 44.97%</td>
<td>46.39%</td>
<td>47.65%</td>
</tr>
<tr>
<td>9.8 Mbps</td>
<td>1.16%</td>
<td>CI 0.66%</td>
<td>1.62%</td>
<td>4.01%</td>
</tr>
</tbody>
</table>

CL: Confidence limit  CI: Confidence interval

### Table 2. Precision achieved when the switch is under different traffic loads

<table>
<thead>
<tr>
<th>Traffic load</th>
<th>CL ±10 us</th>
<th>±20 us</th>
<th>±50 us</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>CI 97.54%</td>
<td>99.72%</td>
<td>99.86%</td>
</tr>
<tr>
<td>9.8 Mbps</td>
<td>CI 97.40%</td>
<td>99.76%</td>
<td>99.88%</td>
</tr>
</tbody>
</table>

CL: Confidence limit  CI: Confidence interval

### Table 4. Precision achieved using a TOD clock without traffic load

<table>
<thead>
<tr>
<th>Confidence limit</th>
<th>±10 us</th>
<th>±20 us</th>
<th>±50 us</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidence interval</td>
<td>68.01%</td>
<td>84.70%</td>
<td>88.22%</td>
</tr>
</tbody>
</table>

### Table 5. Precision achieved by the NTP protocol

<table>
<thead>
<tr>
<th>Computer</th>
<th>CL ±10 us</th>
<th>±20 us</th>
<th>±50 us</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>CI 69.72%</td>
<td>91.96%</td>
<td>98.66%</td>
</tr>
<tr>
<td>B</td>
<td>CI 64.28%</td>
<td>88.94%</td>
<td>96.26%</td>
</tr>
</tbody>
</table>

CL: Confidence limit  CI: Confidence interval

precision when the time server was under different traffic loads. Table 3 indicates that the traffic jam at the NIC influences the precision of synchronization badly, and even causes synchronization message losses. The experimental results reveal that the synchronization precision can be guaranteed statistically when the speed of receiving packets in the time server is restricted to low enough. It is not hard to implement the necessary traffic restriction on the switch’s port.

### 5.3 Synchronization using TOD clocks

The characteristics of a software clock implemented using a Time-Of-Day function were analyzed in Section 2. Similar to the first experiment in Section 5.2, an experiment was implemented to measure the precision of relative clock synchronization using a TOD clock. No traffic load was generated other than synchronization messages. The synchronization request interval was 100 ms, and the number of messages was 10000. The difference $E_{TOD}$ between two consecutive measured time differences $t_{md}$ was measured as the synchronization precision. Table 4 shows that the confidence of the synchronization precision is much worse than the precision achieved by using a TSC-based clock as per Table 2.

As the most widely used clock synchronization protocol, NTP software uses the TOD function to access the time, and piggybacked timestamps to achieve absolute synchronization with respect to a time server. To deal with the clock drift caused by intrinsic frequency errors, the hybrid kernel Phase-Locked Loop/Frequency-Locked Loop (PLL/FLL) was designed by David Mills for the NTP project [13]. At the specified time interval (between 64 s and 1024 s), the NTP program updates the time client relative to the time server.

A simple experiment was designed to test the precision of synchronization achieved by the NTP. The timer server and experiment machines are in the same LAN. The NTP program ran continuously on two computers (A and B) whose hardware configuration is shown in Table 1. The clock update by the NTP program was performed 5000 times. The time adjustment by the NTP program at each computer was recorded during the experiment. As shown in Table 5, it is clear that the confidence of the synchronization precision achieved by the NTP program was much lower than that achieved by our method using TSC registers.

### 6 Conclusion and future work

To support time-critical computations in Local-Area Networks, we have presented a high-precision relative clock synchronization protocol built using the Intel Pentium processor’s Time Stamp Counter register, rather than the Time-Of-Day clock usually used for this purpose. It was shown that a high precision of (relative) clock synchronization could be achieved using TSC registers, significantly outperforming the (absolute) synchronization possible under the
widely-used Network Time Protocol. This outcome offers a low cost time synchronization solution rather than using expensive customised hardware, such as a satellite radio receiver, while still providing high accuracy.

The precision of a local clock is influenced by the intrinsic frequency error of the hardware oscillator used, which may cause non-deterministic clock drift [9]. Although previous work has compensated for this problem as far as possible [13], a TOD clock is hard to use for high precision (sub-millisecond) timing measurements. Compared with a TOD clock, a clock derived from a Time Stamp Counter register is more stable and easier to read. Our experiments showed that for relative clock synchronization, the degree of confidence of the precision achieved using a TSC-based clock is much higher than that using traditional TOD clocks.

In future work, our method will be evaluated and improved in complex network architectures, because the symmetric delay assumed in this paper may not hold under large-scale network architectures with time-varying traffic load. The synchronization implementation will be packaged as a library routine, and may be integrated into one of the Linux-based operating systems.

Acknowledgement This work was supported in part by the Australian Research Council under Linkage Projects grant number LP0776344 to C. Fidge, by the Australian Government’s Department of Education, Science and Training under International Science Linkages Scheme grant number CH070083 to Y.-C. Tian, and by the Natural Science Foundation of China under grant number 60774060 to Y.-C. Tian.

Thanks to the anonymous ICECCS 2008 reviewers for their helpful suggestions about future work.

References