12-14-2001

Global Synchronization of Asynchronous Computing Systems

Richard Neil Barnes

GLOBAL SYNCHRONIZATION OF ASYNCHRONOUS COMPUTING SYSTEMS

By

Richard Neil Barnes II

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

December 2001

GLOBAL SYNCHRONIZATION OF ASYNCHRONOUS

COMPUTING SYSTEMS

By

Richard Neil Barnes II

Approved:

James C. Harden Graduate Coordinator of the Department of Computer Engineering (Director of Thesis)	Nicholas H. Younan Professor of Electrical Engineering (Committee Member)
Donna S. Reese Professor of Computer Science (Committee Member)	A. Wayne Bennett Dean of Engineering

Name: Richard Neil Barnes II

Date of Degree: December 14, 2001

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Dr. James C. Harden

Title of Study: GLOBAL SYNCHRONIZATION OF ASYNCHRONOUS COMPUTING SYSTEMS

Pages in Study: 93

Candidate for Degree of Master of Science

The MSU ERC UltraScope system consists of a distributed computing system, custom PCI cards, GPS receivers, and a re-radiation system. The UltraScope system allows precision timestamping of events in a distributed application on a system where the CPU and PCI clocks are phase-locked. The goal of this research is to expand the UltraScope system, using software routines and minimal hardware modifications, to allow precision timestamping of events on an asynchronous distributed system.

The timestamp process is similar to the Network Time Protocol (NTP) in that it uses a series of timestamps to improve precision. As expected, the precision is less accurate on an asynchronous system than on a synchronous system. Results show that the precision is improved using this sequence of timestamps, and the major error component is due to operating system delays. The errors associated with this timestamping process are characterized using a synchronous system as a baseline.

ACKNOWLEDGMENTS

The author extends his sincere appreciation to the members of his graduate committee, Dr. Jim Harden, Dr. Donna Reese, and Dr. Nicholas Younan, for all the advice and help that they have provided me throughout my graduate studies.

The author would also like to acknowledge Dr. Donna Reese, Greg Henley, and See-kit Lam for their contributions towards the software, operating system considerations, and general debugging.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Monitoring physically distributed computing systems require the ability to create precision timestamps based on events that occur during an application. Precise timestamps are used in a variety of applications including database systems, communications, distributed system performance evaluation, and parallel application debugging. There are a wide range of implementation schemes for precision timestamping that involve both hardware and software solutions. However, a hardware-only solution is usually expensive and may perturb the system when collecting timestamp data, while software-only solutions tend to have poor timestamp precision. A hybrid solution of low-cost hardware and software routines can provide high precision timestamps. The MSU ERC UltraScope is an example of such a hybrid system that allows precision timestamping of events in distributed and parallel applications [1].

1.1 The UltraScope system

The UltraScope system consists of a distributed computing system equipped with custom PCI boards and Global Positioning Satellite (GPS) receivers. Embedded in the parallel application are software probes that trigger timestamps for application events. The custom, PCI-bus-based Probe Acquisition Board (PAB) allows a software event to

be correlated to Universal Coordinated Time (UTC) time. Using the GPS receiver's one-pulse-per-second (1PPS) signal allows the PAB board to resolve a timestamp into UTC time. GPS system timing is precise to within approximately ± 15 ns [2]. A high-precision link between the CPU and PAB timestamps is facilitated by PC-based hardware where the CPU and PCI clocks are synchronized with respect to each other. With the popularity of the Internet, techniques have been developed for time synchronization of physically separated computers. The Network Time Protocol (NTP) is a formal standard by which computers are time synchronized. By using similar techniques, the UltraScope system can be expanded to systems where the CPU and PCI clocks are unsynchronized.

1.2 Timestamping on an asynchronous system

A limitation of the current implementation of the UltraScope system is the CPU and PCI clocks must be synchronous with respect to each other. Only a subset of computing systems, such as Intel PC's, currently has this characteristic. Furthermore, there is no guarantee designs will continue to be synchronous in the future. Software routines based on NTP can be used to estimate the system characteristics, such as the ratio of the CPU clock frequency to the PCI bus clock frequency and the time offset between the host processor and the PAB board. From these characteristics, the delays associated with accessing the PAB board, such as the PCI bus latency, can be corrected. When these errors are accounted for, the estimated timestamp will have a high correlation

to UTC time. Naturally, there will be some error associated with the timestamps that is not present in the synchronous case.

1.3 Error sources in the asynchronous timestamp process

Additional errors in the timestamping process are due to delays in creating the initial timestamp, the PCI bus latency, the operating system's activities, and translating the timestamp into UTC time. The system characteristics to be estimated include the ratio of the CPU clock frequency to the PCI bus clock frequency and the time offset between the times represented by the processor and the PAB board. Errors in these calculations translate into a loss of precision in the final timestamp value. By using the asynchronous timestamping methods described in Section 1.2 on a synchronous system, each of these errors can be characterized, which will allow the determination of the specific error contributions incurred by the asynchronous timestamping methods. Additionally, this procedure validates the correctness of the asynchronous methods.

1.4 Research scope

The goal of this research is to extend the UltraScope system using software routines and minimal hardware modifications to precisely generate and resolve timestamps to UTC time on asynchronous, distributed computing systems and to characterize any remaining sources of error.

The approach taken is to modify the timestamping process so that it is similar to NTP. To timestamp an event, timestamps must be created both at the host processor and at the PAB board. The PAB hardware must be modified so that an accurate local timestamp can be created by the software. Software routines are needed to determine the system characteristics, perform the timestamping process, and to correct for the PCI bus delays that are a result of accessing the PAB board.

This thesis presents a review of previous research in the area of precision timestamping, the theory and design of the hardware modifications and software routines, the results of this research, and the conclusions made from these results.

CHAPTER II

LITERATURE REVIEW

The previous research in precision timestamping presented here covers several areas: applications of precision timestamping, timestamp ordering issues in distributed systems, methods of clock synchronization, accuracy of the GPS signals, and the error sources involved in timestamping.

2.1 Applications of precision timestamping

There are many different applications of precision timestamping from e-commerce to communications systems. For e-commerce systems, many issues, such as ownership of an item, monetary transactions, and various other legal issues would benefit from precise timestamps that correspond to a universal time [3]. If a client and server have two different times, it is possible that fraud or even theft of a product or service could occur [3].

Distributed databases also benefit from precision timestamps that correspond to a universal time. Transactions to the database can occur from several different clients at once. Without precise timestamps synchronized to a global timebase, the database would not know the correct order in which to commit the transactions [4]. Some databases also

use the timestamps as a method of obtaining locks on the data so that it can be modified exclusively by a client.

Digital communications systems are another example of system that can use precision timestamping to obtain an estimate of the quality of service over the network. Communication networks can be operating over a large distance with a few or many clients. In some cases, the clients may be devices on the network that act as slaves such as on an interface bus or they may be computers attached to a wide area network. Precision timestamping allows for distributed communication between the clients which will allow it to operate as a large metasystem [1].

Distributed, parallel applications are difficult to debug. In addition, estimating the performance of the algorithm is problematic. Each processing node is running code with no synchronization to other processors except through explicit synchronization calls. Using small triggers embedded in the software that generate timestamps can create a trace of the events on a processing node. If these events are precisely synchronized to a global time, then the events on all the nodes can be merged into a single trace of events [5]. This merged event trace can then be used to debug errors in the code or to determine exactly how long a section of code required to execute.

2.2 Timestamping

Timestamping is used to resolve the order of system events, even when the system is distributed or characterized by non-deterministic delays. A timestamp is generated by

an event trigger and can provide an absolute or relative indication of the time at which the event occurred.

The timestamp for a sequence of events on a system can be chosen to indicate one of the following: time before any of the sequence of events, the time immediately before a specific event, the time immediately after a specific event, or the time after all events have occurred. The time that a timestamp represents is chosen by the application [4]. For example, timestamps can be used as concurrency controls to create locks in the application [4]. The synchronization of the time references is important. Having an unsynchronized time reference can cause the timestamps to appear to have been received in the reverse order from which they were generated [6].

Timestamps can also act as events that trigger other events in a system. For this type of event triggering, especially in a distributed system, it is important that the order of the timestamps is preserved across all communication. Using timestamps with a global time reference is an obvious way to generate events and messages that are suitable for event-based computing [7]. Using a global time reference may not always be feasible, and not every application requires global timestamping or high precision. Some applications may only need the timestamp to be resolved to a relative time.

2.3 Clock synchronization

Clock synchronization is required where there are two or more clocks in the system that are generated from different sources. These clocks can be on the same local

system or physically distributed. Clock synchronization attempts to provide a way these two separate clocks can be deterministically related to each other. For clock oscillators, this can be a matter of a phase lock between the two oscillators. For a clock-generated timestamp based on a counter, synchronization adjusts the rate at which the counter increments to achieve synchronization.

2.3.1 Types of algorithms for clock synchronization

Much research has been done in the area of clock synchronization. The two major types of synchronization systems are hardware algorithms and network algorithms [8]. Hardware algorithms require a dedicated set of communication links through which the clocks can be broadcast. This method is expensive, but the communication delays are known and deterministic in the dedicated network [8]. Network algorithms share the communication links with the rest of the system. This method is cheaper than the hardware methods, but it introduces uncertainties in the communication delays. This problem is usually solved by sending timestamp messages back and forth between the two clock sources. The downside of this technique is that these messages increase the load on the network [8].

2.3.2 Probabilistic clock synchronization

Many of the network algorithms are probabilistic in nature. Unlike deterministic hardware solutions, these algorithms use timestamp messages to estimate the statistical properties of the communication delays and try to remove these delays from the system.

The primary statistics that are estimated are the roundtrip delay between the timestamp messages, the time offset between the clocks, and the amount of drift between the two clocks. Research has been conducted to find better estimates of these parameters and better ways to implement synchronization messages.

There are many different schemes for network clock synchronization. One probabilistic scheme uses wavelets. The wavelet-based approach is able to completely filter out any pattern-dependent jitter [9]. Another scheme is to continuously run the synchronization algorithm. This way the network load due to the synchronization is constant rather than bursty [8]. Other methods include the Time Transmission Protocol [10], the time protocol, and the Internet Control Message Protocol (ICMP) timestamp messages [11]. The most popular of all the clock synchronization protocols is NTP.

2.3.3 NTP

The NTP protocol is based on the time protocol and ICMP timestamp messages [11]. There are over 100,000 NTP servers and clients on the Internet [12]. NTP uses a series of timestamps as shown in Fig. 2.1 [13]. In the NTP protocol, a series of these timestamp exchanges are made between the client and server. The roundtrip delay, time offset between the clocks, relative clock drift, and the most accurate time server are determined. The results are passed through a clock filter and then used to update the local clock oscillator. The amount of error in the synchronization determines the length of the interval the algorithm waits before exchanging another series of

timestamps [13]. NTP is able to synchronize clients to a time server over the Internet to within 10 ms [12].

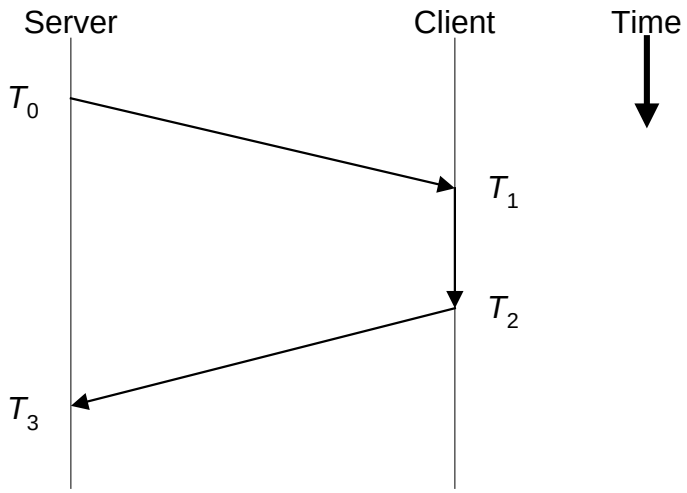

Fig. 2.1 NTP timestamp sequence

2.4 GPS

The GPS system consists of a constellation of 24 satellites orbiting the earth. Each satellite contains an onboard atomic clock that is used to create a signal that is broadcast to earth. The atomic clock is synchronized to UTC time [14]. The GPS receivers receive the signals broadcast by the satellites. The receiver uses these signals to compute its current position. The current UTC time can then be solved for using the positional equations contained in the receivers [14]. When the position of the receiver is accurately known, a more accurate UTC time can be computed. Until May 2, 2000, the

government used a process called selective availability to intentionally degrade the performance of the civilian GPS receivers [2], making this time less accurate.

The GPS receiver used in the UltraScope system is the Motorola UT Oncore receiver. When selective availability was active, the precision of the 1PPS signal from this receiver was ± 50 ns (1-sigma) when the current position of the receiver was known [15]. Now that selective availability is disabled, the precision of the 1PPS signal has been measured to be ± 15 ns when the current position of the receiver is known [2]. For timing purposes, the GPS 1PPS signal has excellent long-term stability, but poor short-term stability.

2.5 Error sources

There are many different error sources involved in clock synchronization and precision timestamping. In clock synchronization, there are errors inherent in the clock characterization, the synchronization of the clocks, the communications network, and the synchronization algorithms. In precision timestamping, the major sources of error are the generation of the timestamp and the communication network.

2.5.1 GPS error sources

The major errors in the GPS system are the receiver error and the precision of the 1PPS signal. The receiver error consists of positional error, atmospheric error, multipath error, and local oscillator error. The positional error is due to not knowing exactly where

the receiver is located. As mentioned earlier, the UTC time is determined by solving the positional equations in the receiver. If the position is not exactly known, there will be error introduced into the solution of the equations. This error is reduced by telling the receiver its precise location [14]. Atmospheric error is due to the change in the speed of the satellite signals when the signals penetrate the earth's atmosphere. The multipath error is due to the signal reflecting off surfaces and delaying its time of arrival at the receiver. Multipath error can also occur when both the GPS signal and a delayed version of that signal reach the receiver. If the reflected, delayed signal is strong enough, it could be interpreted by the receiver as the true signal rather than noise. Local oscillator error is due to the asynchronous relationship between the GPS signals and the local clock oscillator. The result of this error is a sawtooth pattern, shown in Fig. 2.2 [2]. The sawtooth pattern results from underestimating or overestimating the actual beginning of the 1PPS signal.

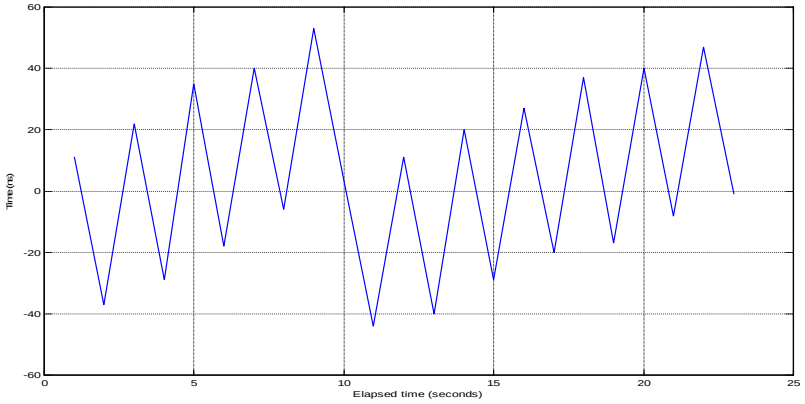

Fig. 2.2 The sawtooth pattern present in the GPS 1PPS signal

The precision of the 1PPS signal is within 300 ns when selective availability is active and the receiver is not told its current position. When the receiver has a known position and selective availability is active, the precision increases to ± 50 ns [14]. Measurements have been made using the Motorola UT Oncore receiver now that selective availability has been disabled. These measurements indicate that the precision of the 1PPS is now ± 15 ns when the receiver has a known position [2].

2.5.2 Quartz Clock oscillator error sources

Quartz oscillators are commonly employed and in general have good short-term stability, but poor long-term stability. An oscillator's error comes from two major factors: aging of the crystal and temperature variations. A quartz crystal oscillator will typically drift up to 5 parts in 10^8 per week due to aging [2]. Crystals also experience temperature instability. The amount of instability caused by temperature varies by the type of crystal and the cut of the crystal. The temperature stability of a crystal is usually a cubic function of temperature that varies from ± 10 parts per million (ppm) to ± 100 ppm [2].

2.5.3 Clock synchronization error sources

Clock synchronization errors are due to mismatch in the two clocks and the communications network. Clock synchronization has several parameters to describe the synchronized clocks: time offset, frequency offset, and relative clock drift. The time offset is due to the clocks starting at different instants in time. The frequency offset is

due to the difference in frequencies between the two clocks and determines the relative rate at which one clock increments with respect to the other clock. The relative clock drift is an indication of variation in clock synchronization over time and can be caused by not being able to correct the frequency offset between the two clocks due to temperature or crystal aging. The frequency offset and relative clock drift can be corrected by phase locking one clock oscillator to the other. The time offset can be corrected by adjusting the frequency of the clock oscillator until the time offset is reduced.

2.5.4 Communication network error sources

The communication network provides additional errors when the synchronized clocks are physically distributed. The roundtrip delay, the possibility of receiving timestamp messages out-of-order, and network traffic can affect the synchronization process. In many synchronization techniques, the delay is assumed to be symmetric between the source and destination when computing the roundtrip delay. Checkpoints have to be inserted in to the synchronization algorithms to insure that the timestamps are operated on in order. Using out-of-order timestamps can cause erroneous results when computing the roundtrip delay, time offset, etc.

2.5.5 Timestamp error sources

Timestamp errors are caused by the generation of the timestamp itself and the communication network. The error in generation of timestamps can be caused by delays in reading the timebase of the system or even in the software itself. Reading of the local

timebase can be delayed by the operating system and the low-level processor instructions needed to access it. An operating system manages many different processes including itself. Significant delays can occur if the allocated time for the process requesting the timestamp expires as it initiates the request of the timestamp. Smaller delays are also caused by the general overhead present in the operating system. Because of the speed of most processors compared to the memory system, a large delay can occur if the instructions to read the local timebase have to be loaded into cache. This delay can also be quite small if the instructions are already located in the instruction cache of the processor. The processor can also delay the reading of the timebase due to the out-of-order execution found in most of the modern processors. For the timestamp to be completely accurate when indicating the order of events, the processor's instruction pipeline should be flushed. The flushing of the pipeline causes a non-deterministic delay that depends on how many instructions were loaded into the pipeline ahead of the timestamp instruction. This delay can be reduced by not flushing the pipeline. However, there will be no way to guarantee the order of events in this case.

The magnitude of these errors is not the only important characteristic. The variability of the timestamp errors can cause problems as well. If an error source has a low variability, this error can be corrected by shifting the timestamp in time. The error is more difficult to correct when a high variability error source is present. The accuracy of the timestamp is reduced since the accuracy will vary by the same amount as the error sources.

In conclusion, high-precision timestamps have many different uses from database systems to communications systems. There are many different error sources in a timestamp including the creation of the timestamp and, in the case of NTP, transmitting a timestamp across a widely distributed network. By using technologies such as GPS and NTP, these errors can be reduced resulting in a more precise timestamp. A combination of hardware and software can yield even better precision timestamps at a relatively low cost compared to strictly hardware-based systems.

CHAPTER III

THEORY OF OPERATION

This chapter discusses the differences between synchronous and asynchronous computing systems. There are different requirements for timestamping on each system. These requirements are reflected in the hardware and software components of the UltraScope. This section covers the fundamental differences between these two types of computing systems, the hardware and software components of the UltraScope for each system, and an overview of the source of errors incurred on an asynchronous system.

3.1 Synchronous and asynchronous computing systems

For the purpose of this work, a synchronous computing system is a system that has a fixed relationship between the clock signal for the processor and the clock signal for the PCI bus [1]. This fixed relationship is usually in the form of phase-locked signals. The PCI bus and the processor have different clock frequencies, but the ratio between the processor clock frequency and the PCI bus frequency is fixed and deterministic. Examples of synchronous systems include many of the current Intel-based PC's.

On an asynchronous system, the clock frequency ratio is not a deterministic value. This can be due to having two non-phase-locked clock signals or by having a method that maintains a phase-lock only periodically. Examples of asynchronous systems include

many Sun Microsystem workstations and servers, such as the UltraSparc 10 workstation and the Enterprise 450 server.

3.2 Synchronous operation

To create a timestamp, the synchronous UltraScope must establish a relationship between the event to be timestamped and the time at which that event occurred. By using GPS, the time of a timestamp can be expressed as UTC time, a precise global time reference suitable for use in widely distributed systems. The function of the PAB board is to create a relationship between the event and UTC time [1].

There are three components necessary to establish this correlation: the re-radiation system, the PAB hardware, and the software routines to timestamp an event. The re-radiation system allows the GPS receivers to receive the GPS transmissions without requiring direct line-of-sight with the GPS satellites. The PAB hardware correlates the software event to UTC time. The software routines are called by the distributed application to initiate the timestamping of an event.

3.2.1 Hardware

The hardware components of the synchronous UltraScope system are the re-radiation system and the PCI-based PAB board. The PAB board hardware consists of a GPS receiver, a temperature sensor, an FPGA, and 512KB of onboard memory. The FPGA on the PAB board is divided into five functional blocks: PCI, GPS, Timecore,

Time Lookup Table (TLT), and the Temperature block. The PCI and GPS blocks allow the PAB board to interact with the PCI bus and GPS receiver, respectively. The Timecore and the TLT blocks contain the functionality for the actual timestamping process. The Temperature block is used to improve the accuracy of the timestamps when the UltraScope is used on a synchronous system.

The majority of the Timecore is a free-running 60-bit counter called the PAB counter. The PAB counter is initialized to zero when the PCI bus reset is released. The PAB counter is clocked using the 33MHz PCI bus clock. The Timecore also contains a 60-bit register (PPS count) that is loaded using the 1PPS signal from the GPS receiver. At the rising edge of the 1PPS, the PAB counter is sampled and stored in this register. Since the PAB counter is clocked using the PCI bus clock, there is a direct relationship between events on the processor and the PAB counter itself.

The TLT provides a way to store the PPS counts over time. The TLT allows a PAB count to be correlated to a UTC time. When the TLT is enabled by the user application, the current PPS count and UTC time are stored in the PAB's onboard memory. On every rising edge of the 1PPS signal, a compressed version of the PPS count for that second is stored in the TLT. The TLT establishes the link between a PAB count and UTC time.

The TLT uses the second derivative of the PPS count to reduce the 60-bit PPS counts to 8-bit values. The first derivative of a PPS count is denoted with a single prime (*PPS'*); the second derivative uses a double prime (*PPS"*). When the TLT is started the current PPS count PPS_0, its first derivative PPS'_0, and the current UTC time, UTC_0, is

stored in a header in the PAB board's memory. At each second, the TLT computes and stores the second derivative of the PPS count using:

$$PPS'_n = PPS_n - PPS_{n-1} \quad (3\text{-}1)$$

$$PPS''_n = PPS'_n - PPS'_{n-1} \quad (3\text{-}2)$$

The value of PPS''_n is stored in the TLT. This process is shown as a block diagram in Fig. 3.1. The TLT also stores the current temperature at the time a new PPS count is generated. A similar compression method is used to store the temperature.

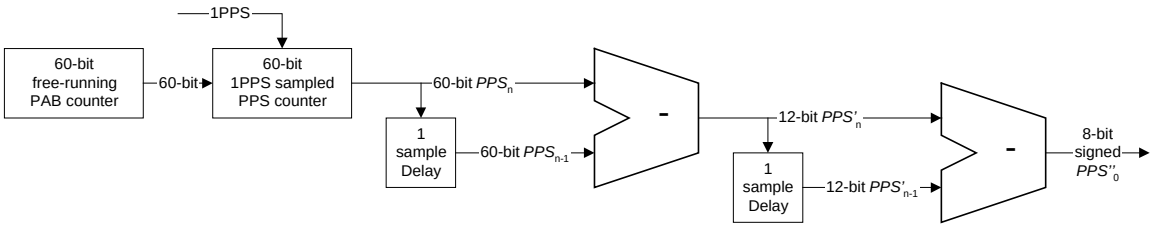

Fig. 3.1 The hardware block diagram for computing the *PPS*"

3.2.2 Software

The timestamps are collected during the execution of the user application. On a post-mortem basis, the timestamps are converted to UTC times. The timestamping software consists of several steps. The first step collects the actual timestamp. This is done by reading the 64-bit performance monitoring counter on the host processor. The performance monitoring CPU counter increments using the processor clock, and it is initialized to zero when the CPU reset is released on bootup [16]. This 64-bit CPU counter value represents the time, with respect to the processor, at which an event

occurred. By only reading the CPU counter, the timestamp generation is a low overhead routine that causes little perturbation of the executing software [1].

The next step converts the CPU count, CNT_{CPU} to a corresponding PAB count, CNT_{PAB}. There are two parameters needed to convert the CPU count to the corresponding PAB count. The first parameter is clock frequency ratio (R) between the PCI clock and the processor clock. The other parameter is the time offset (CNT_{Offset}) between the counters. Once these parameters are known, it is possible to convert a CPU count to a PAB count using:

$$CNT_{PAB} = \frac{(CNT_{CPU} + CNT_{Offset})}{R} \tag{3-3}$$

The final step is to convert the PAB count to the UTC time at which the event occurred. The first step in this process is to iterate through the TLT and reconstruct the PPS counts using the difference counts stored in the TLT. As the PPS counts are determined, the UTC time for each PPS count is also calculated and stored. The PAB count corresponding to the software event can now be correlated to UTC time. Through interpolation of the PPS counts, the PAB count of the event is converted to a UTC time and stored. The timestamps are now in their final form of UTC times and can be used for a variety of purposes.

3.3 Operating synchronous UltraScope on an asynchronous system

When using the synchronous version of the UltraScope on an asynchronous system, there are several problems. One problem is computing a highly accurate estimate

of the clock frequency ratio. Initially, this does not seem to be a problem. Time averaging estimates of the clock frequency ratio over a long period of time usually provides a highly accurate estimate. However, the counter values for both the CPU and PAB counters become very large as the system is running. Even a very small error in the clock ratio estimate can amount to seconds or hours of error depending on the length of time the computing system has be running. The error due to the clock frequency ratio is a linear function of time. This problem can be solved using a sufficient number of calibration cycles over a period of time. The calibration cycles can correct for this linear error.

However, an even more difficult problem deals with temperature variations. In a system where the PCI and CPU clocks are phase-locked, temperature variations will affect both clocks, but not the clock frequency ratio. In an asynchronous system, the temperature variations primarily affect the stability both of clocks and can affect the clock frequency ratio. Unless the temperature for both clocks can be controlled or monitored for correction the entire time the system is powered up, there is little that can be done to improve synchronization of the counters with each other. This prevents a precise relationship between the CPU count and the PAB count.

3.4 Modifications required for asynchronous operation

On an asynchronous system, the deterministic relationship between the PCI clock and the CPU clock does not exist. To complicate matters, the clock frequencies for the

PCI and CPU clock oscillators are temperature dependent. The temperature dependency between the PCI and CPU clocks causes the clock frequency ratio to vary over time. Without a precise estimate of the clock frequency ratio, the relationship between the CPU and PAB counters cannot be established. The traditional software method of synchronization, such as NTP, is to use a series of timestamps. These timestamps are collected at both the server (the host processor) and the client (the PAB board). The disadvantage to this method is the large amount of overhead involved in collecting a timestamp sequence.

The timestamp process is modified so that an event triggered series of timestamp measurements are made on both the processor and the PAB board. The timestamp sequence is then used to correct some of the errors incurred by accessing both the CPU counter and the PAB board. These modifications include changing the hardware to support accurate indicators that can be used as a timestamp and modifying the software to collect a timestamp sequence rather than a single timestamp measurement.

3.4.1 Hardware

The PAB hardware must be modified so that precision timestamps may be obtained on an asynchronous system. Rather than creating more hardware for timestamp measurements, the existing 60-bit PAB counter is used. The modified timestamp process requires that the PAB count must be readable by the application software. For the synchronous UltraScope, the sampled PPS count is readable by software at offset 0x05

from the base address of the PCI board. The logic was changed so that the free-running PAB count is now available at offset 0x05 instead of the PPS count.

Since the free-running PAB count is now readable by application software, the 60-bit counter itself was modified to be more robust. Originally, the PAB counter was massively pipelined into 15 stages. This is changed so that the counter is pipelined into 4 stages. The reduction in pipelined stages reduces the latency required to read a PAB value. It also reduces the possibility of an error due to lower stages of the pipelined counter rolling over. If the PAB counter could be read in a single, atomic operation, this error would not be possible. A 32-bit PCI bus implementation is used for the PAB board. The PAB counter is 60-bits so it requires two accesses to the PCI bus to completely read the PAB counter. There is no hardware mechanism that prevents the first 32-bits from changing as the remainder of the counter is read. Bounds checking can be performed that will prevent an invalid count from being used in the timestamp calculations.

3.4.2 Software

The software required for asynchronous operation has two major parts. The first part collects the timestamp. This step now involves a sequence of readings similar to NTP. The timestamp sequence used in the asynchronous case is shown in Fig. 3.2. A timestamp sequence is created by reading the CPU counter, C_0, then the free-running PAB counter, P_0, and then another CPU count, C_1. The two CPU counter readings are taken to allow for correction for PCI bus transaction delays. Unlike the synchronous

case, there is no conversion needed to convert the timestamp into a PAB count. The PAB counter now represents the timestamp of an event rather than a CPU count.

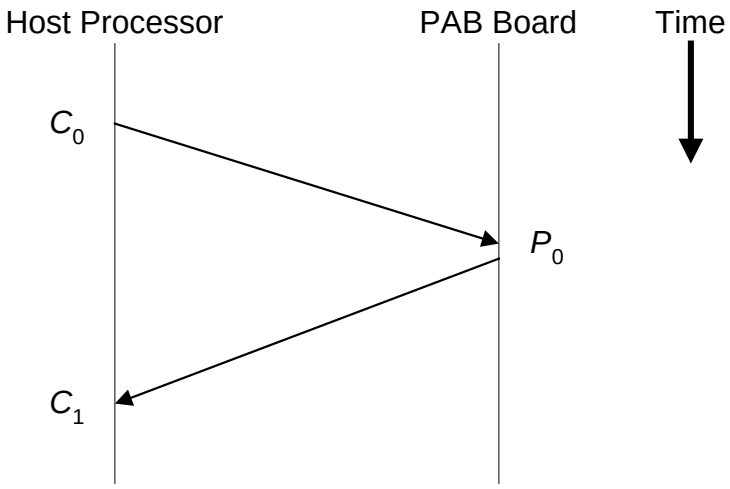

Fig. 3.2 The sequence of timestamps required on an asynchronous system

The only conversion step necessary is to use the TLT to interpolate the UTC time given the PAB count of the timestamp. Because this timestamp uses the PAB count directly, the delay of the PCI bus transaction is contained inside the timestamp. By recording the CPU counts before and after reading the PAB counter, a correction can be applied to minimize the effect of the PCI bus transaction.

3.4.3 PCI transaction correction

The method used to adjust the timestamp value P_0 to account for the PCI bus delays depends on several assumptions. A timestamp sequence can be divided into three distinct regions. These regions are shown in Fig. 3.3. The actual event is designed by *E*.

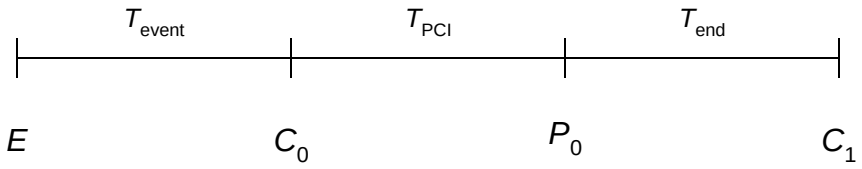

Fig. 3.3 Regions of delays in a timestamp sequence

The first region, T_{event}, is the time it takes to read the first CPU count C_0 after the actual event, E, occurs. This time is determined by the overheads of entering the timestamp generation routine, loading the instructions for the CPU counter into cache, and reading the first CPU count. The next region, T_{PCI}, is the time elapsed from C_0 to P_0. This time is dominated by the PCI bus transaction delays. The final region, T_{end}, is the time elapsed from P_0 to C_1. This time is composed of the time to read the final CPU count and the operating system delays.

Reading the CPU counter accesses a local resource, but reading the PAB counter requires a PCI bus transaction. Therefore, it is possible that T_{PCI} will be large compared with T_{event} and T_{end}. Since T_{event} and T_{end} contain similar delays, they will appear to be nearly equal as T_{PCI} grows relative to T_{event} and T_{end}. Therefore, the first assumption is to equate T_{event} with T_{end}. The time T_{event} contains the time required to enter the timestamp routine, load the required processor instructions into cache, and read the CPU counter, C_0. When reading the last CPU count, C_1, the time T_{end} includes the operating system overhead, reading the CPU counter, and returning from the timestamp routine. The

delays in T_{event} and T_{end} are similar in both nature and magnitude. The next assumption is that the clock frequency ratio can be approximated by:

$$R \approx R_{est} = \frac{C_1}{P_0} \tag{3-4}$$

Using (3-4), a PAB count can be converted to a CPU count if the ratio and the time offset, CNT_{Offset}, are known. When a system has been running for a sufficiently long time, the counter values for the PAB counts and CPU counts will be greater than the time offset by several orders of magnitude. When approximating the ratio in (3-4), the absence of the time offset results in a minimal difference between R and R_{est}. This is illustrated in more detail in Section 3.5.

The correction for the PCI bus is given as follows:

$$T_{event} = T_{end}$$

$$\frac{C_0}{R} - E = \frac{C_1}{R} - P_0$$

$$E = P_0 - \frac{C_1 - C_0}{R} \tag{3-5}$$

$$R \approx R_{est} = \frac{C_1}{P_0}$$

$$E' = P_0 - \left(P_0 - P_0 \frac{C_0}{C_1} \right)$$

$$E' = P_0' = P_0 \frac{C_0}{C_1} \tag{3-6}$$

By using this estimation of the event timestamp, the corrected PAB count is being shifted in time so that it occurs before the PCI bus transaction, as shown in Fig. 3.4. Since both T_{event} and T_{end} incur the delay of reading the CPU counter, this delay is also corrected. The disadvantage to this correction is any delays that are a part of T_{end} and not a part of T_{event} will cause the corrected PAB count to be moved an additional amount of time. If T_{end} is greater than T_{event}, then the PAB count will be corrected so that it is now represents a time before the software event.

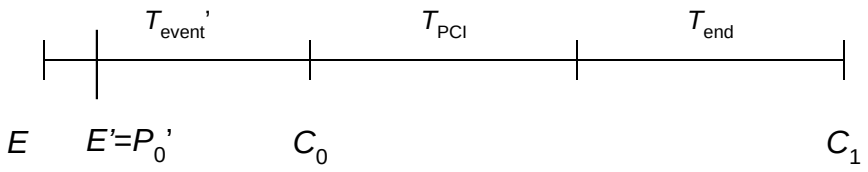

Fig. 3.4 The timestamp sequence after correction

To analyze the error resulting from shifting the PAB count too far or too little, let K be the PAB count at the time of the actual event, E, and use the approximation in (3-6) to attempt to estimate the event. The estimate of the event, E, is given by E'. Also, let the times T_{event}, T_{PCI}, and T_{end} be in terms of CPU counts. This results in:

$$C_0 = RK + T_{event}$$

$$P_0 = \frac{RK + T_{event} + T_{PCI}}{R}$$

$$C_1 = RK + T_{event} + T_{PCI} + T_{end}$$

$$E' = P_0 \frac{C_0}{C_1} = \frac{RK + T_{\text{event}} + T_{\text{PCI}}}{R}\left(\frac{RK + T_{\text{event}}}{RK + T_{\text{event}} + T_{\text{PCI}} + T_{\text{end}}}\right)$$

$$E' = K\left(1 + \frac{T_{\text{event}} + T_{\text{PCI}}}{RK}\right)\left(\frac{1 + \frac{T_{\text{event}}}{RK}}{1 + \frac{T_{\text{event}} + T_{\text{PCI}} + T_{\text{end}}}{RK}}\right)$$

Let the error be ε where $\varepsilon = E' - E$ (3-7)

$$\varepsilon = K\left(1 + \frac{T_{\text{event}}}{RK} + \frac{T_{\text{PCI}}}{RK}\right)\frac{\left(1 + \frac{T_{\text{event}}}{RK}\right)}{\left(1 + \frac{T_{\text{event}}}{RK} + \frac{T_{\text{PCI}}}{RK} + \frac{T_{\text{end}}}{RK}\right)} - K$$

Using Binomial expansion: $(1+X)^{-1} = 1 - X + X^2 - X^3 + \ldots$, $X^2 < 1$

or $(1+X)^{-1} \approx 1 - X$, $X^2 << 1$. This can be used since $K >> T_{\text{event}}$, T_{PCI}, or T_{end}.

$$\varepsilon = K\left(1 + \frac{T_{\text{event}}}{RK} + \frac{T_{\text{PCI}}}{RK} + \frac{T_{\text{end}}}{RK}\right)\left(1 - \frac{T_{\text{event}}}{RK} - \frac{T_{\text{PCI}}}{RK} - \frac{T_{\text{end}}}{RK}\right) - K$$

$$\varepsilon = K\left(1 + \frac{T_{\text{event}}}{RK} - \frac{T_{\text{end}}}{RK}\right) - K$$

If $T_{\text{event}} = T_{\text{end}}$ then

$$\varepsilon = K - K = 0 \quad (3\text{-}8)$$

If $T_{\text{end}} = T_{\text{event}} - \Delta$ then

$$\varepsilon = K\left(1 + \frac{\Delta}{RK}\right) - K$$

$$\varepsilon = \frac{\Delta}{R} \quad (3\text{-}9)$$

Note that the error, ε, does not depend on T_{PCI}. The variations in the delays associated with T_{PCI} will have no effect on the error; only variation in T_{event} or T_{end} will cause a change in ε. As long as $K >> T_{event}$, T_{PCI}, and T_{end}, the resultant error ε does not depend on K. Therefore, if $T_{event} \neq T_{end}$, then the difference between these two times is the amount of error that is added to the corrected PAB count, P_0'. This result is confirmed experimentally in Chapter 4.

3.5 Overview of error sources

With the asynchronous UltraScope system, there are several error sources in the timestamping process. When collecting the timestamp, there are delays due to reading the CPU counter, the PCI bus transaction, and operating system functions. Other error sources include the inaccuracies associated with the 1PPS signal and computing the approximation to the clock frequency ratio. Table 3.1 lists each error source, the general amount of error supplied to the timestamps, and indicates if that error can be reduced in software.

Table 3.1 Error sources and their contributions to the overall error

Error Source	Amount of error supplied	Can be reduced?	Clock frequency dependent?
Reading the CPU counter	1 - 2 μs	Yes	Yes
PCI bus transaction	2 - 3 μs	Yes	Yes
PCI bus arbitration	1 - 2 μs depending on the number of devices †	Yes	Yes
Operating system delays	0.5 - 3 μs depending on the system load	No*	Yes
GPS 1PPS error	± 15 ns	Error is negligible	No
Clock frequency ratio approximation	< ± 1 ns	Error is negligible	No

* Using an open-source operating system such as Linux, may allow better insight into these errors, and therefore allow the errors to be reduced.
† The PCI bus specification provides typical delays due to bus arbitration [17].

To read the CPU counter, the processor's pipeline must be flushed. If the pipeline is not flushed, the count returned may represent a time before the software event occurs or significantly after the event [16]. This uncertainty is due to the out-of-order execution capability that most modern processors utilize. Flushing the processor's pipeline removes this uncertainty; however, it adds a small amount of delay before the CPU counter can be read. The delay of flushing the pipeline and reading the CPU counter is incurred twice for every timestamp created.

The PCI bus transaction is a major source of error when creating a timestamp on an asynchronous system. A problem with a PCI bus transaction is the variability of the

time required to complete a transaction. As the number of different transactions on the PCI bus increases, the arbitration of the PCI bus becomes a large factor in the PCI delays.

The delays incurred from the operating system are the most troublesome. The operating system can switch processes just before a timestamp is created or during the timestamping process. Additionally, when accessing the PCI bus, the operating system places the distributed application into a blocked state until the PCI bus transaction has completed [18]. When the PCI bus transaction has completed, the operating system must switch back to the distributed application. This context switching adds a delay before the second CPU counter value is read. A diagram of the relevant process states under Unix is shown in Fig. 3.5 [18]. Because the operating system underlies all of the software that is running on a system, these delays cannot be accurately measured nor characterized.

The errors due to the GPS signals and the approximation of the clock frequency ratio are miniscule in comparison to the other sources of error. The GPS 1PPS signal, without selective availability, has a variation of approximately $\pm$ 15 ns [2].

Each of these error sources will be discussed and characterized more specifically for several cases of system usage. The test cases used are for when 1) the PCI bus is nearly idle, 2) the PCI bus is busy but under normal usage, and 3) the PCI bus has several devices attempting to use the bus simultaneously.

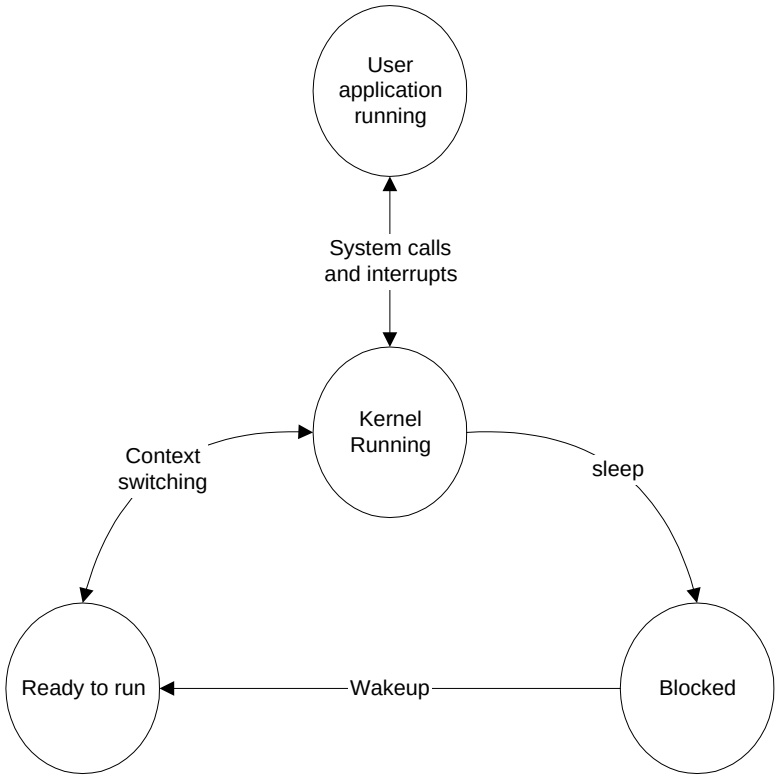

Fig. 3.5 The process states and transitions possible when accessing the PCI bus

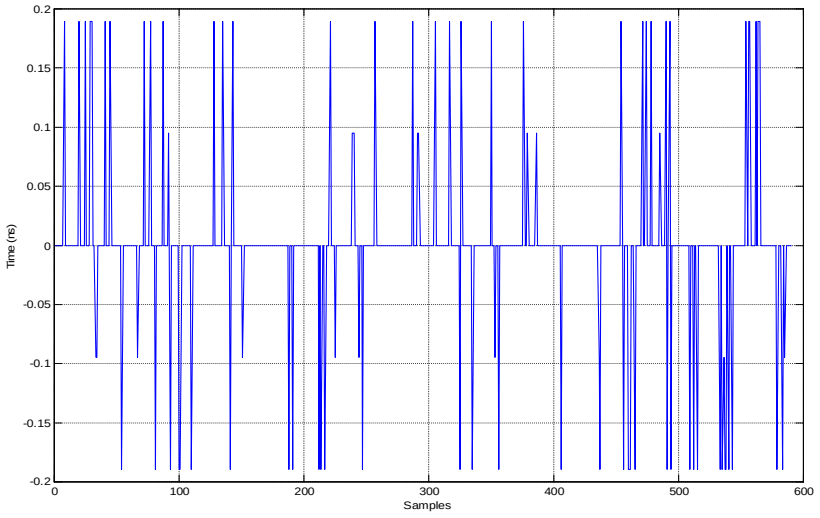

Fig. 3.6 Measured error from using an approximation of the clock frequency ratio

CHAPTER IV

RESULTS

This chapter discusses the methodology used for verification of the hardware and software modifications as well as collection and analysis of the timestamp data. Three major cases are investigated. The first case is when the PCI bus is idle during most of the application's runtime. The second case is when the PCI is busy but under a normal load, such as on a multi-user system. The last case is when the PCI bus is busy and device arbitration occurs. The error for each of these cases is characterized and analyzed.

4.1 Test and verification procedure

A series of tests and programs are run to determine the characteristics of a computing system. These tests are outlined in Appendix A. The tests can be divided into three categories: testing the hardware and software modifications to the PAB board, determining the system characteristics, and timestamp collection and analysis. All of these tests were performed on an Intel 233 MHz Pentium II PC. The test computer is a synchronous system.

Testing the PAB counter modifications requires two PAB boards. One board is installed running the synchronous design, and the other board contains the modified, asynchronous design. Over a period of several days, the counters on both PAB boards

are read. The difference between the two PAB counters should be within one second of each other. Other functions, such as communications with the GPS receiver and the temperature sensor, are checked in case the hardware modifications caused other areas of the board to malfunction.

The characteristics of the computing system can be determined with the software listed in Appendix A. An important characteristic is whether a system is synchronous or not. When probing the processor clock signal and the PCI bus clock signal with an oscilloscope, a synchronous system will exhibit a phase lock between these two signals. In an asynchronous system, the phase between these two clocks will vary over time, possibly only on an intermittent basis. Such was the case for the Sun systems tested where the phase lock of the CPU and PCI clocks were allowed to slip periodically.

Most of the analysis of an asynchronous system is performed to validate the timestamp algorithm. From these tests, an estimate of the overall timestamp precision can be formed. A significant amount of error is due to the PCI bus transactions. The load of the PCI bus is varied so that the error, and its impact on timestamp precision, can be determined. Three cases of PCI bus loading are investigated. When the PCI bus is idle, there is a minimum of extraneous system activity, which allows the error to be characterized accurately. The load of the PCI bus is increased which better simulates the actual conditions on a multi-user system. The timestamp precision achieved in this case will be more representative of the actual achievable precision. As the load on the PCI bus increases, the probability that bus arbitration occurs increases as well. PCI bus arbitration adds a significant amount of time to a bus transaction. For this reason, these

arbitration delays will be simulated to investigate their affect on the precision of a timestamp.

4.2 PCI bus idle

A common trend in distributed computing is the move to a cluster of computers instead of using a large multiprocessor machine. In a cluster environment, a user is assigned a number of nodes from the cluster. Except for the operating system and the queuing software, the user application has exclusive access to these nodes. Exclusive access to a group of nodes reduces the amount of traffic on the PCI bus. Most of the PCI bus activity would be caused by the PAB board, the network interface card, and a disk controller card. A single PAB board is used in the test PC with no applications running other than the test software.

Timestamp data is collected over both short and long periods of time. Recall from Fig. 3.3, a timestamp can be divided into three times: T_{event}, T_{PCI}, and T_{end}. By using a synchronous system, all PAB counts can be converted to CPU counts, and therefore, T_{PCI} and T_{end} can be measured. The time of the actual event is unknown, so T_{event} cannot be directly measured. However, (3-8) and (3-9) use T_{end}, which can be measured, to estimate T_{event}. T_{PCI} and T_{end} are formed by:

$$T_{\text{PCI}} = \frac{CNT_{\text{PAB}_0} + CNT_{\text{Offset}}}{R} - CNT_{\text{CPU}_0} \tag{4-1}$$

$$T_{\text{end}} = CNT_{\text{CPU}_1} - \frac{CNT_{\text{PAB}_0} + CNT_{\text{Offset}}}{R} \tag{4-2}$$

Once the timestamp data is collected, the PAB count is shifted in time using (3-3). The corrected PAB count is converted into CPU counts so the data can be analyzed.

The two measurable times, T_{PCI} and T_{end}, are shown in Fig. 4.1. It is important to observe that the delays due to the PCI bus are similar in magnitude to the remaining delays. A strong motivation for correcting the PAB count was the thought that the PCI bus delays would be much worse than the remaining errors. The variation of the time for a bus transaction is not much more than the variations due to the operating system and reading the CPU counter. However, it can be seen in Fig. 4.1 that the PCI bus delays have no effect on T_{end}, which implies no effect on the timestamp precision. These results validate the approach taken with the instrumentation hardware and the software routines.

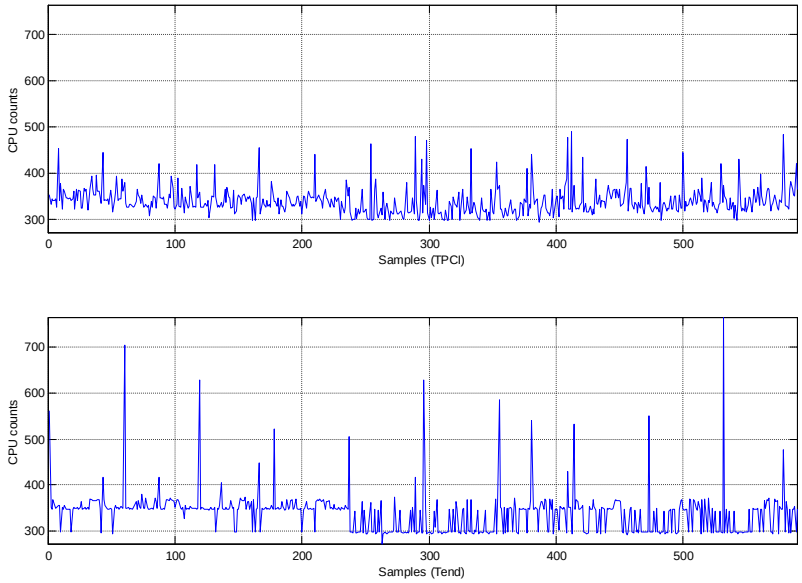

Fig. 4.1 Timestamp data when the PCI bus is idle

To create an estimate of the timestamp precision, the magnitude and variation of T_{end} must be examined. When the PCI bus is idle, the average delay for T_{end} is 300 to 350 CPU counts. The variation of this delay is ± 25 CPU counts. The amount of error contained in T_{end} is assumed to be the amount of error in the estimate of the time the event actually occurred. Therefore, the precision is limited by that error. For a distributed system consisting of a cluster of computers, the overall precision of a timestamp is limited to 1.28 - 1.50 μs ± 107 ns, when the processor clock frequency is 233 MHz. The precision is dependent on the clock frequency of the processor. The computing system is at the lowest possible activity, so this is the best case scenario for an asynchronous machine.

4.3 PCI bus busy

A situation where the PCI bus experiences moderate to frequent activity is when the distributed system is a multi-user system. In this case, the user does not have exclusive access to the processing nodes. Software other than the operating system may be running at all times. The amount of processing time and I/O time required by the running software will vary. This makes the timestamping process less accurate due to the increased level of activity on the computing system.

A second PAB board was installed into the test system. Both boards are PCI bus target devices with no bus mastering capability. A program that continuously accessed the second PAB board was started in the background. The background program was

designed to continuously read a 48-byte block of the secondary PAB board's registers. In the foreground, the timestamp collection program was running and collecting timestamps on a regular basis using the primary PAB board.

The procedure used to analyze the timestamps is the same procedure used when the PCI bus was idle. The results of the timestamp data are shown in Fig. 4.2. For this data, there is significantly more variation in the PCI bus delays than in the previous case. Similar to when the bus is idle, the PCI bus delays do not affect the estimate of when the event occurs.

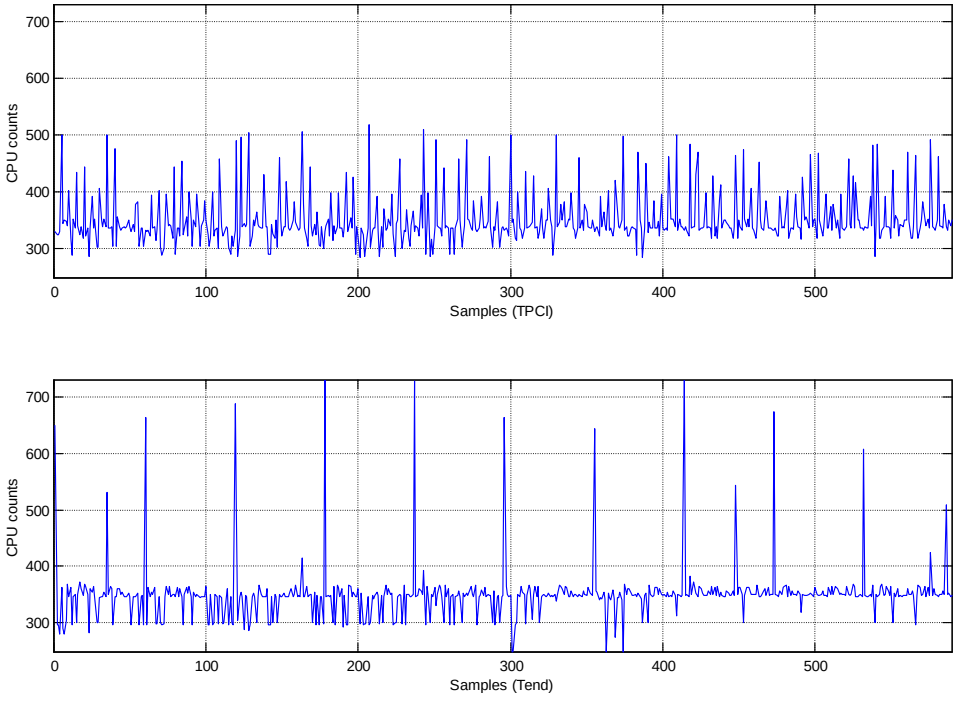

Fig. 4.2 Timestamp data when the PCI bus is busy

The operating system has a more active role in determining the precision of the system. On a 233 MHz computer, it requires 220 CPU counts to read the CPU counter, assuming the instructions have already been loading into cache. There is some additional delay, approximately 75 CPU counts, that can only be caused by the operating system. As the operating system is managing multiple user applications, these delays are slightly worse than in the previous case. The large spikes that appear in T_{end} are likely to be caused by the operating system responding to the background application rather than the foreground test program.

The additional error from the operating system slightly degrades the timestamp precision when the PCI bus is heavily loaded. The average amount of error present in T_{end} yields a precision of 350 - 350 CPU counts, or 1.50 - 1.63 μs ± 107 ns on a 233 MHz Intel PC. Surprisingly, the variation of the precision is not any worse than when the PCI bus is idle.

4.4 PCI bus with arbitration

If multiple bus mastering devices use the PCI bus on a system, bus arbitration may occur. Bus arbitration does not require a heavily loaded bus. If at any time two or more devices request access to the bus, arbitration will occur. There are an assortment of arbitration schemes that can be used. Fixed priority, priority-based round-robin, and equal priority round-robin are just a few of the arbitration schemes that may be used. If the requesting PCI devices are bus masters, there can be several physical devices

participating in the bus arbitration. If none of the devices are bus masters, the motherboard must handle the arbitration between all of the devices requesting the PCI bus. The PCI bus specification leaves the actual implementation details of the arbitration scheme up to the manufacturers. Currently, Intel uses a round-robin scheme in their PCI bridge chipsets [19]. Sun also implements a round-robin arbitration scheme for their systems [20].

The PAB board is a target-only device, so it is not capable of participating in bus arbitration. The Intel PC used in these tests did not contain any bus master PCI devices. To test the performance of the UltraScope with bus arbitration, a Matlab simulation was created. The PCI arbitration simulation adds bus arbitration delays to existing timestamp data.

The arbitration simulation requires several parameters that are used to define the delays associated with bus arbitration. These parameters determine the number of devices that are requesting the PCI bus when arbitration occurs and the amount of delay each requesting device adds to the PAB board's PCI bus transaction. If a device gains control of the PCI bus before the PAB board, the overhead of the signaling protocol and the bus transaction itself must be accounted for. Once a device is granted access to the PCI bus, the signaling protocol typically takes two PCI bus clock cycles [17]. The latency that is added by a single device's bus transaction during arbitration is dependent on a latency counter. The latency counter defines the maximum amount of time a master may hold the PCI bus during an arbitrated transaction. The value of this counter is configurable and determined by the manufacturer of the PCI device. A typical value for

the latency counter for a bus master is 22 PCI bus clock cycles [17]. The last parameter specifies the frequency at which bus arbitration occurs. For this simulation, a cluster of computers is assumed. Therefore, the PCI bus traffic will be light so bus arbitration will occur infrequently.

The timestamp data used in this simulation represents an idle PCI bus where infrequent arbitration occurs. To better visualize the effect arbitration has on a timestamp, Fig. 4.3 displays the timestamps without arbitration. Fig. 4.4 represents the timestamp data after the arbitration delays have been added. For the T_{PCI} measurements in Fig. 4.4, all of the measurements that have a magnitude greater than 500 CPU counts are due to the arbitration delays.

By examining Fig. 4.3 and Fig. 4.4, it can be seen that the only differences between timestamps with and without arbitration delays occur in T_{PCI}. None of the arbitration delay is present in the estimate of when the event occurs. Consequently, the performance of this timestamp algorithm is not affected by PCI bus arbitration. The precision of the timestamp data resulting from this simulation is the same as when the PCI bus is idle without arbitration, 1.28 - 1.50 μs.

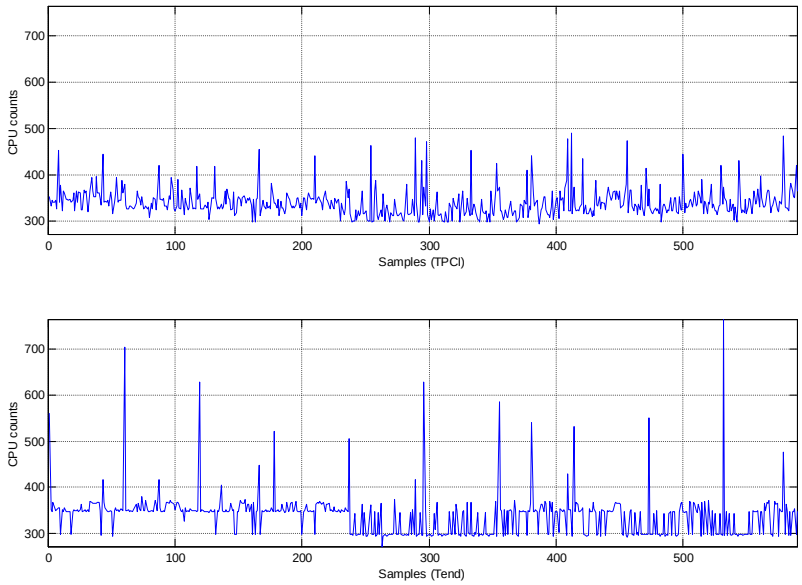

Fig. 4.3 Timestamp data without arbitration

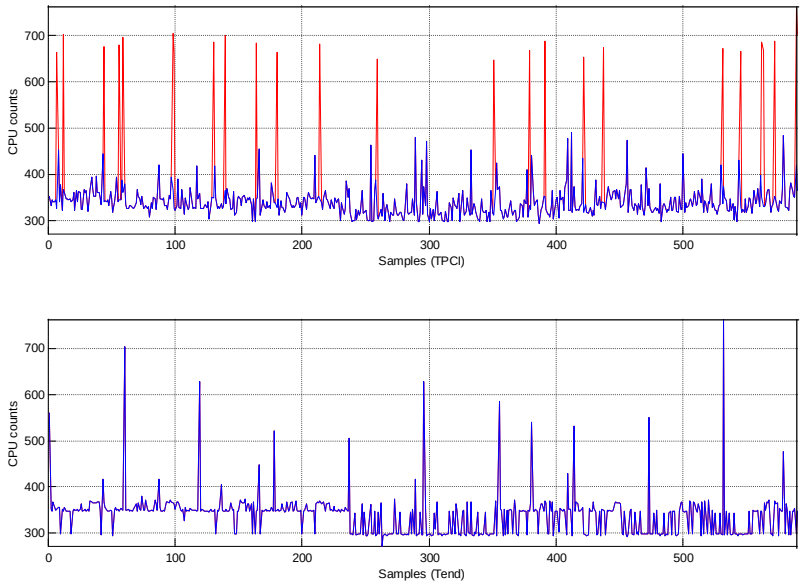

Fig. 4.4 Timestamp data with bus arbitration

4.5 Error characterization

Just as important as the overall precision, are the sources of error that affect the precision. In this section, the error sources will be characterized with respect to each of the three test cases. The error sources due to the timestamp process are reading the CPU counter and accessing the PCI bus to read the PAB counter. Other sources of error are from other executing applications. These are characterized by the operating system error and the PCI bus arbitration error.

Reading the CPU counter first requires the processor's pipeline to be flushed, and then the CPU counter can be read. The time required to flush the pipeline is dependent on the type and number of instructions that are currently executing on the processor. Measurements show flushing the pipeline takes on average 100 clock cycles. Then, it requires 220 clock cycles for the processor to read the CPU counter. This operation has been measured to increase up to 500 clock cycles if the instructions are not held in cache. This error is incurred at the beginning of a timestamp sequence and at the end of the sequence. When the PAB counter is adjusted, the estimate of the timestamped event is improved by 320 clock cycles, which is 1.37 μs.

To read the PAB counter, a PCI bus transaction is initiated. If bus arbitration does not occur, the average error is 2 - 3 μs. Bus arbitration adds an additional 1 - 2 μs depending on the number of devices requesting the PCI bus. The distribution of the error due to the PCI bus is a Raleigh distribution, which is commonly used to model communications systems. Once the PAB counter is adjusted, the effect of the PCI bus error on a timestamp's precision is negligible.

The most troublesome error source is the operating system. Depending on the system load, the operating system contributes 1 - 3 μs. As the system load increases, the error supplied by the operating system is likely to increase as well. This error is worse than the PCI bus error because it cannot be better estimated or corrected without access to the inner workings of the operating system. The distribution of this error source can be characterized similar to the PCI bus error. Fig. 4.5 shows the empirical distribution of error that is contributed from the operating system. The empirical distribution is a histogram that reports the frequency of occurrence for a range of values. The distribution shown in Fig. 4.5, is similar to the distribution of the PCI bus errors, which is a Raleigh distribution.

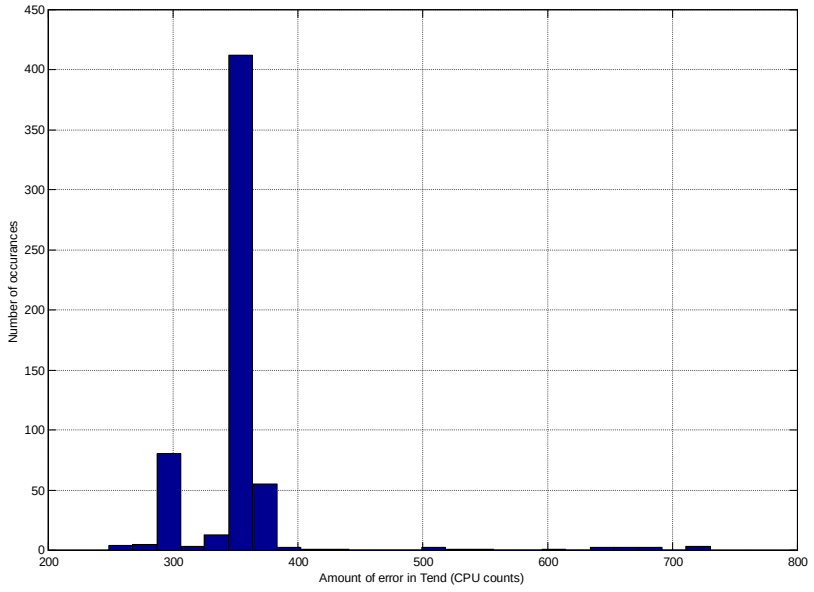

Fig. 4.5 Distribution of error contributed by the operating system

The significant sources of error are the PCI bus transaction and the operating system delays. By correcting the PAB counter in a timestamp sequence, the PCI bus errors are removed as well as most of the errors due to reading the CPU counter. It is likely that some of the error due to the operating system is corrected by the overhead when entering the timestamp collection routine; however, this cannot be verified.

CHAPTER V

CONCLUSIONS

The goal of this work is to allow precision timestamping of events in distributed and parallel applications. The UltraScope instrumentation hardware and software allowed precision timestamps to be created on systems where the PCI bus clock and the processor clock were phase locked together. This research expands on this previous work to allow precision timestamping on computing systems that do not exhibit this characteristic.

Modifications to both the hardware and software components were required to achieve this goal. The timestamping process was modified so that an event driven series of timestamps is collected on both the host processor and the PAB board. A correction to the final timestamp value is created to account for the delays incurred by the PCI bus, such as bus arbitration and initiating and completing a bus transaction.

5.1 Timestamp precision

The overall precision and contributions of the error sources were investigated for three cases: the PCI bus is idle, the PCI bus is busy, and the PAB must arbitrate for access to the PCI bus. A distributed system that consists of a cluster of nodes allows a user to gain exclusive control over a group of nodes. The result is a nearly idle PCI bus

and low system load. The PCI bus is being heavily used represents a parallel system that is shared among many simultaneous users. Finally, a simulation of PCI bus arbitration is used to verify the corrections applied to a timestamp can adjust for bus arbitration. Table 5.1 lists each case and the average precision of the timestamp data.

Table 5.1 Each test case and the average precision of the timestamps

Test case	**Precision (CPU counts)**	**Precision (Time)**
PCI bus idle	300 - 350 ± 25 counts	1.28 - 1.50 μs ± 107 ns
PCI bus busy	350 - 380 ± 25 counts	1.50 - 1.63 μs ± 107 ns
PCI bus with arbitration	300 - 350 ± 25 counts	1.28 - 1.50 μs ± 107 ns

When the PCI bus is idle, the overall timestamp precision was 300 to 350 CPU counts. This translates into 1.28 - 1.50 μs on a 233 MHz Pentium II Intel computer. The precision suffered a variability of approximately ± 107 ns.

The major differences between a busy PCI bus and the idle case are the operating system delays and the variability of the PCI bus delays. The variability of the PCI bus delays had a negligible impact on the timestamp precision. The accuracy of these timestamps is 1.50 - 1.63 μs with a variation of ± 107 ns. The increased operating system delays of a busy, multi-user system reduces the overall precision slightly.

The PCI bus arbitration simulation was designed to test the performance of the PCI bus correction given by:

$$P_0' = P_0 \frac{C_0}{C_1} \tag{5-1}$$

The PCI bus arbitration is treated as a longer than normal PCI bus transaction. The correction in (5-1) positioned the final timestamp so that it occurs before the PCI bus arbitration. Consequently, the precision obtained in this case is similar to the previous cases. The overall precision is 1.28 - 1.50 μs with a variability of ± 107 ns.

These three cases validated the use of (5-1) to accurately correct for the PCI bus delays. The experimental results shown in this work validate the mathematical justifications given in Chapter 3. These results confirm that using a series of timestamps rather than a single timestamp value can be used to increase the precision on an asynchronous distributed system, by accounting for the PCI bus transaction delays.

5.2 Error sources and their contributions

Examination of the test results using a synchronous system allows the classification of the various errors that are present in the timestamps. The error sources and their contributions are listed in Table 5.2.

Table 5.2 List of error sources and their contribution to the overall error

Error source	Final error supplied to a timestamp	Affects precision?	Clock frequency dependent?
Reading the CPU counter	< 1.20 μs	Yes	Yes
PCI bus transaction	± 30 ns	No	Yes
PCI bus arbitration	± 30 ns	No	Yes
Operating system delays	0.5 - 3 μs	Yes	Yes
GPS 1PPS error	± 15 ns †	No	No
Clock frequency ratio approximation	± 30 ns	No	No
Timestamp correction	± 30 ns	No	No

† This is the measured GPS 1PPS error after Selective Availability was disabled [2].

The major errors are the PCI bus delays, reading the CPU counter, and the operating system delays. By using the correction factor given in (5-1), the PCI bus delays can be reduced substantially. The delays associated with reading CPU counter include loading the instructions into the processor's cache, flushing the processor's pipeline, and reading the counter itself. The time required to read the counter can be measured and has a low variability. Flushing the pipeline requires additional time and is a high variability delay. Loading the instructions into cache is also a non-deterministic delay. The operating system delays are non-deterministic. The operating system sits under all of the tests that can be performed using the UltraScope. This prevents a more thorough investigation of these delays.

As the processor clock frequency increases, the precision of the timestamps is affected. The clock frequency dependent error sources shown in Table 5.2 will improve

as the clock frequency increases. The errors due to the PCI bus transaction and bus arbitration will improve as the clock frequency of the PCI bus increase. The two prominent errors, reading the CPU counter and the operating system delays, will improve as the processor clock frequency increases. The operating system delays are unlikely to improve at the same rate as the other clock frequency dependent error sources. Each version of an operating system is usually more complex than before; therefore, the actual delays will be reduced, but there will be more work for the operating system for a given task.

5.3 Future Work

Further investigation needs to be performed in the characterization of the operating system delays. This can involve using of an open-source operating system such as Linux or using low-level debugging tools. By learning more about the operating system on a low level, with respect to hardware usage, the delays may be reduced in a similar manner as the PCI bus delays.

There are distributed systems that would benefit from a real-time resolution of timestamps into UTC time. Real-time timestamping can be used as part of a feedback system that allows the detection of an abnormal condition. The system can then correct or bypass this condition and resume normal operation. Examples of this type of real-time system would include wide-area network routers and systems that have a network of redundant systems. To facilitate real-time resolution of timestamps, the PAB hardware

would need to efficiently return a precise UTC time rather than a PAB count. The PAB logic can be modified so that the PAB count and UTC time are stored every second. Additional logic will need to be created so that a PAB count occurring between the 1PPS pulses will be interpolated into a UTC time. The GPS receiver causes problems with precision in a real-time system. The GPS receiver returns a UTC time at the beginning of every second. The hardware must interpolate using the UTC times to convert a PAB count to a UTC time. Some GPS receivers can be configured to output a 100 pulse-per-second signal rather than a one-pulse-per-second. Under this configuration, the UTC time will be updated 100 times per second and will allow better accuracy when translating a PAB count to a UTC time. When the user application initiates the generation of a timestamp, the interpolated UTC time is returned to the application. Much of the logic pertaining to the generation of the second derivative of the PAB counts and the TLT can be removed.

Other issues include the synchronization of the PAB counter to current UTC time and the format of the UTC time returned from the system. The PAB counter is a free running counter, but the UTC time is only output from the GPS receiver at set intervals. There must be some logic in the system that will continuously synchronize the PAB counter with a UTC time. Finally, the format of the output of the UltraScope must be considered. Currently a UTC time is represented by a 64-bit integer that is the number of nanoseconds since January 1, 1970. The UTC time can be compressed, but the method of compression will be determined by the usage of the real-time UltraScope. If the real-time system is a custom built system, the UTC time may be compressed into a non-standard

format, much like the PPS counts described in Chapter 3. A more widespread usage will require a common method of compression. Compression of the UTC time is desired due to the PCI bus restrictions. Reading a 64-bit UTC time from the PAB board will required two separate PCI bus transactions. By compressing the UTC time into 32-bits, only one PCI bus transaction will be needed to produce a timestamp.

Furthermore, the UltraScope system can be expanded for use in network applications such as Quality of Service or for use in timestamping of analog data. This system can be expanded to operate on a variety of applications. Future versions of the UltraScope can be modified for use on a faster bus such as the next generation 3GIO/PCI-X buses.

REFERENCES

[1] J. C. Harden and W. C. Francis, "GPS-Based Instrumentation for Computing Systems," in *Proc. Southeast Symposium on System Theory*, March 1999.

[2] T. P. McMahon, "Improving the Precision of GPS-Based Instrumentation Systems," *Mississippi State University Project Report*, May 2000.

[3] Hugh Melvin and Andy Shearer, "GPS and SCADA: Taking Care of Business in Internet Time," in *GPS World*, Nov 2000.

[4] Betty Salzberg, "Timestamping After Commit," in *Proc. International Conference on Parallel and Distributed Information Systems*, 1994, pp. 160-167.

[5] Jim C. Harden, Donna S. Reese, Marlene B. Evans, Sudarshan Kadambi, Gregory J. Henley, and Chuck E. Hudnall, "In Search of a Standards-Based Approach to Hybrid Performance Monitoring," *IEEE Parallel and Distributed Technology: Systems Applications*, Vol 3. No 4, pp. 61-71. Winter 1995.

[6] C. J. Bouras and P. G. Spirakis, "The Perfect and Imperfect Clocks Approach to Performance Analysis of Basic Timestamp Ordering in Distributed Databases," in *Proc. International Conference on Computing and Information*, 1993, pp. 403-407.

[7] C. Liebig and M. Cilia, "Event Composition in Time-dependent Distributed Systems," in *Proc. IFCIS International Conference on Cooperative Information Systems*, 1999, pp. 70-78.

[8] Alan Olson, Kang G. Shin, and Bruno J. Jambor, "Fault-Tolerant Clock Synchronization for Distributed Systems Using Continuous Synchronization Messages," in *Proc. International Symposium on Fault-Tolerant Computing*, 1995, 154-163.

[9] Fred Daneshgaran and Marina Modin, "Clock Synchronization Using Wavelets," in *Proc. IEEE Global Telecommunications Conference*, 1995, pp. 1287-1291 vol. 2.

[10] K. Arvind, "Probabilistic Clock Synchronization in Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol 5. No 5, pp. 474-487, May 1994.

[11] David L. Mills, "Internet Time Synchronization: The Network Time Protocol," *IEEE Transactions on Communications*, Vol 39. No 10, pp. 1482-1493, October 1991.

[12] David L. Mills, "Improved Algorithms for Synchronizing Computer Network Clocks," *IEEE/ACM Transactions on Networking*, Vol 5. No 3, pp. 245-254, June 1995.

[13] D. L. Mills, "Network Time Protocol (Version 3)," *RFC-1305*. March 1992.

[14] J. Blake Bullock, T. Michael King, Howard L. Kennedy, Edward D. Berry, and Gregg Zanfino, "Test Results and Analysis of a Low Cost Core GPS Receiver for Time Transfer Applications," in *Proc. IEEE International Frequency and Control Symposium*, 1997, pp. 314-322.

[15] Motorola Corporation, *Motorola GT/UT Oncore User's Guide*. Motorola, 1997.

[16] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*. Intel, 2001.

[17] *PCI Local Bus Specification, Revision 2.1, June 1, 1995*. PCI Special Interest Group, June 1, 1995.

[18] Uresh Vahalia, *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentis Hall, 1996.

[19] Intel Corporation, *Intel 440 LX AGPSET: 82443LX PCI AGP Controller (PAC) Datasheet*. Intel, 1998.

[20] Sun Microsystems, *U2P: UPA to PCI Interface User's Manual*. Sun Microsystems, 1997.

APPENDIX A

TEST PLAN

A new system consists of the re-radiation system, a GPS receiver, the PAB board, and the computing system. Each of these components must be tested to insure all of the components have been installed correctly. In addition to testing, several parameters must be determined before the UltraScope can be used as part of a user application. The system parameters are the clock frequencies of the CPU and PCI clocks, if the system is synchronous, and the time offset between the CPU counters and the PAB board.

A.1 Re-radiation system

The re-radiation system can be tested using a handheld GPS receiver. By moving the receiver around the general area of the computing system, the GPS receiver should have good signal strength.

A.2 Physical indictors

When the PAB board is installed and the system is powered up, the following steps are performed:

1. The LED next to the DB9 connector on the PAB board must be lit. This indicates the PAB board has power. If this LED is not turned on, check to make sure the PAB board is seated completely in the PCI slot.
2. The second LED from the DB9 connector must also be lit. This indicates the PAB board has been programmed. If this LED is not on, the EPROM near the top of the PAB board must be checked. Make sure this chip is completely seated into the socket and reboot the computer.

3. The third LED indicates the GPS 1PPS signal is enabled and being received. When this LED is not on, it could indicate the GPS receiver is either not seated properly or that the 1PPS output of the receiver is disabled.
4. The fourth LED should only be active when using a serial connection to another PAB board in place of a GPS receiver.

A.3 GPS functionality

The proper GPS receiver operation can be checked by examining the output of various GPS messages. The current date and time, the 1PPS functionality, and the current position mode of the receiver need to be checked.

1. Execute *display_psd* for at least 30 seconds.
    - Ignore the first five iterations. They are old, invalid messages that are displayed.
    - Check the Date and Time fields for the correct values.
    - Verify the position is the correct and does not change with each iteration.
    - Verify the velocity does not change with each iteration.
    - Verify that at least four satellites are tracked by the receiver.
    - If *display_psd* hangs, that indicates a possible error with the 1PPS output.
2. If the position and velocity vary, execute *phm_on* and enter the correct position.

A.4 PAB functionality

The PAB consists of several functional blocks: the PCI block, the GPS block, the Timecore block, the Time Lookup Table (TLT) block, the Memory block, and the

Temperature block. The PCI block and the GPS block are tested as a result of testing the remainder of the PAB board.

1. Execute *cpu_tlt*.
2. Examine the TLT header that is displayed.
    - Check the Date and Time fields for the correct date and time.
    - Verify the PAB count and PAB Int Count are non-zero.
    - The value of the PAB Int Count should be very close to the PCI bus frequency.
    - Verify the Init Temperature (in degrees Celsius) is correct.
3. Examine the Dump of the TLT Data, and check to insure the results are non-zero. The two right-most digits represent the PAB difference count.

A.5 System parameters

The system parameters determine if the system is synchronous or asynchronous, the clock frequencies for the processor and PCI bus, and the time offset between the processor and PAB values.

1. Execute *cpu_tlt*.
    - The data at the top of the output gives an estimate of the processor clock frequency.
    - The PAB Int Count in the TLT Header gives an estimate of the PCI bus clock frequency.
    - The 10 successive CPU clock readings differences should be approximately 1,000,000,000 counts apart.

2. Examine the seconds since boot according to the CPU count and the PAB count.
    - If the difference is greater than 2 - 5 seconds, the computing system is likely to be asynchronous.
    - If the system is asynchronous, the difference between these two values should grow in a linear fashion over time.
3. Execute *timestamp 300 –1pps a_filename*
    - The output header returns a rounded-off estimate of the processor clock frequency, the PCI bus clock frequency, and the period in nanoseconds for each clock.
    - The header returns the integer clock frequency ratio used in the synchronous version of the timestamp software.
4. Examine the next output header.
    - This header provides estimates of the time offset between the CPU counter and the PAB board. This is used in the synchronous version of the timestamp software.
    - The roundtrip delay is returned which can be used for applications such as network measurements and Quality of Service applications.

5. Examine the histograms for the Time Offset and Clock Drift.
    - If the histogram for the Time Offset is evenly distributed, this indicates an asynchronous system.
    - If the histogram for the Clock Drift represents a wide range of values, a system is asynchronous.
    - The minimum and maximum values for the Clock Drift should be extremely small if the computing system is synchronous.

A.6 Data collection and analysis

There are two steps to collecting and analyzing the error sources in the timestamp data. First the timestamp data must be collected, this is done using *time_seq*. Once the data is collected, it is then analyzed using two Matlab scripts.

1. Execute *time_seq*. This program collects timestamp data over a long period of time.
2. The data in the files are arranged in groups of 60 timestamps. This 60 timestamps were collected over a short period of time. This allows the short-term properties of the asynchronous system to be examined.
3. The groups of 60 timestamps are collected over a long period of time so that the long-term properties can be investigated.
4. Execute the Matlab script *time_analysis.m*.
5. The graphs produced are estimates of the delays resulting from collecting C_0, P_0, and C_1.

6. Execute the Matlab script *pci_sim.m*. This script randomly adds arbitration delays to the timestamp data.
7. The resulting plots show how PCI bus arbitration delays affect the performance of the asynchronous timestamp algorithm.
8. If the correction algorithm is functioning properly, the delays to read C_0 and C_1 should be equal to the corresponding delays when no arbitration is present.

APPENDIX B

PAB LOGIC MODIFICATIONS

This appendix describes the hardware modifications made to the logic of the FPGA on the PAB board. Two modifications were made. The first modification reduced the number of pipelined stages used to create the 60-bit PAB counter. A diagram of the modified counter logic presented in Fig. B.1. The number of pipelined stages was reduced from 15 stages to 4 stages.

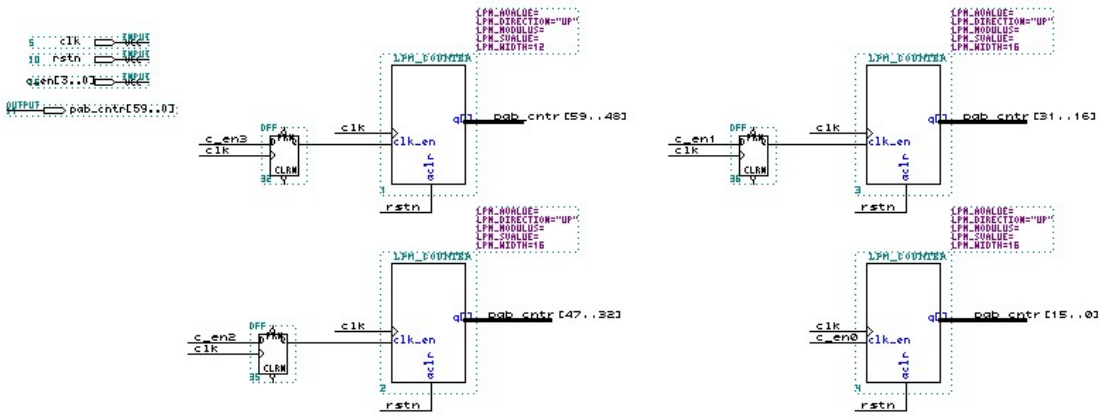

Fig. B.1 The modified PAB counter schematics

The second modification allowed application software to directly read the value of the PAB counter. In the original design, only a sampled version of the PAB count is available. The specific VHDL code modifications, shown in Section B.1, to use the PAB counter rather than the sampled PAB counter is straightforward. For completeness, the full source code is presented in Section B.2. The VHDL and schematic capture files that define the PAB counter are located at:

/ccs/issl/general/ca/UltraScope/max2work/async/timecore_top.

B.1 Modifications to allow software access to the PAB counter

```
DOUT <= ZEROS24&ctrl_reg when ADR=CTRL_REG_BASE else

        "0000000000000000000"&auto_survey&traim_msg_valid&traim_msg_new&no_traim_msg&
                pos_msg_valid&pos_msg_new&no_pos_msg&gps_1pps&pps_error&traim_soln&
                insuff_sats&utc_alarm when ADR=STATUS_REG_BASE else
                "00"&ZEROS24&interrupt_in when ADR=INTERRUPT_REG_BASE else
                alarm_reg when ADR=ALARM_REG_BASE else
                "00000000000"&HOUR&"00"&MIN&"00"&SEC when ADR=UTC_REG_BASE else
-- ************************************************************
-- Modified to allow access to the free-running PAB counter
-- ************************************************************
                c_count(31 downto 0) when ADR=PPS_CNT_LO_BASE else
                "0000"&c_count(59 downto 32) when ADR=PPS_CNT_HI_BASE else
-- ************************************************************
-- End modification
-- ************************************************************
                avg_cnt when ADR=AVG_CNT_REG_BASE else
                ZEROS24&temperature when ADR=TEMPERATURE_BASE else
                ZEROS24&gps_ramq when ADR(7 downto 6) = GPS_MSG_BASE else
                DIN;
```

B.2 Full VHDL source code for the PAB registers

```
-------------------------------------------------------
-- File:        timecore_pci_reg.vhd
-- Author:      Wyatt Francis
-- Date:      March 17, 1999
-- Description: This VHDL file contains the PCI interface logic
--            for the Time Core.  It also contains the control/
--            status registers.
-- History:
--   (03/17/99) REV. 0.1 : Using the PCI_FIFO interface from the GPS section.
--            Adding an address decode for the registers used.  The
--            registers will be implemented as variables within this design
--            file at first, however, they may be switch to LPM functions.
--    (05/09/99) REV. 0.2 : Added a TLT_BUILD control line.
--   (05/13/99) REV. 0.3 : Changing FSM of PCI interface.  Adding ability to read
--            RAM containing GPS parsed messasge.  Thought about moving the registers
--            to a RAM block, however, most information that will be read will be
--             available elsewhere on the chip, so it only needs to be read when it is
needed.
--   (06/25/99) REV. 0.4 : Adding inputs/outputs for additional registers and interrupt
control.
--   (09/28/99) REV. 0.5 : Added a default value for interrupt_clr so it would
automatically let an
--                              an interrupt write to the interrupt register after it had
been cleared.
--
-- Register Description:
--
-------------------------------------------------------
--library lpm;
--use lpm.lpm_components.all;
--library altera;
--use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity timecore_pci_reg is

   generic (
        N : integer := 32    -- Number of bits
         );


    port (

          -- Signals from the PCI BUS
          rstN : in std_logic;          -- PCI Reset

          -- Signals from PCI Megafunction
          lt_adr  : in std_logic_vector(7 downto 2);  --Address from PCI bus
          lt_dato : in std_logic_vector(N-1 downto 0);   -- Data from PCI bus
          lt_cmd0 : in std_logic;  -- PCI command  (read=0; write=1)
          lt_ackn : in std_logic;       -- Local side acknowledge

          -- Signals to the PCIT1
          lt_rdyn : out std_logic;      -- Ready signal
          lt_dati : out std_logic_vector(N-1 downto 0);  -- Data to PCI bus

        -- Signals from counters
        pps_cnt : in std_logic_vector(59 downto 0);
        avg_cnt : in std_logic_vector(31 downto 0);
          c_count   : in std_logic_vector(59 downto 0);

          -- Signals from address decoder
          hit : in std_logic;           -- address space hit

        -- Signals from GPS parser
        gps_ram_busy    : in std_logic;
        gps_ramq      : in std_logic_vector(7 downto 0);
        gps_lock : out std_logic;
        pos_msg_new : in std_logic;
        pos_msg_valid : in std_logic;
        traim_msg_new : in std_logic;
        traim_msg_valid : in std_logic;
        auto_survey : in std_logic;
        insuff_sats : in std_logic;
        traim_soln : in std_logic_vector(1 downto 0);
        gps_1pps : in std_logic;
        utc : in std_logic_vector(16 downto 0);

        -- Signals from DS75
        temperature   : in std_logic_vector(7 downto 0);

          -- Signals to/from TLT_CONTROL
          tlt_build : out std_logic;
          pps_error : in std_logic;
        no_pos_msg : in std_logic;
        no_traim_msg : in std_logic;

          -- Signals to/from interrupt control block
        utc_alarm : in std_logic;
        utc_alarm_value : out std_logic_vector(31 downto 0);
        interrupt_clr : out std_logic_vector(5 downto 0);
        interrupt_in  : in std_logic_vector(5 downto 0);
        utc_alarm_int_en : out std_logic;
        traim_soln_not_ok_int_en : out std_logic;
        pps_error_int_en : out std_logic;
        pps_edge_int_en : out std_logic;
```

```
        pos_msg_int_en : out std_logic;
         traim_msg_int_en : out std_logic;

          -- Clock inputs
          clk : in std_logic           -- PCI clock

          );

end timecore_pci_reg;

architecture behav of timecore_pci_reg is

--BASE ADDRESS FOR REGISTERS (PCI address bits 7 downto 2)
   constant CTRL_REG_BASE       : std_logic_vector(lt_adr'range) := "000000";  --OOh
   constant STATUS_REG_BASE     : std_logic_vector(lt_adr'range) := "000001";  --04h
    constant INTERRUPT_REG_BASE   : std_logic_vector(lt_adr'range) := "000010";  --08h
   constant ALARM_REG_BASE     : std_logic_vector(lt_adr'range) := "000011";  --0Ch
   constant UTC_REG_BASE       : std_logic_vector(lt_adr'range) := "000100";  --10h
   constant PPS_CNT_LO_BASE     : std_logic_vector(lt_adr'range) := "000101";    --14h
   constant PPS_CNT_HI_BASE     : std_logic_vector(lt_adr'range) := "000110";  --18h
    constant AVG_CNT_REG_BASE   : std_logic_vector(lt_adr'range) := "000111";  --1Ch
   constant TEMPERATURE_BASE    : std_logic_vector(lt_adr'range) := "001000";  --20h
   constant GPS_MSG_BASE        : std_logic_vector(1 downto 0)   := "01";       --40h to
7ffh (bits 6 and 7)
   constant RESERVED_BASE      : std_logic_vector(lt_adr'range) := "100000";    --80h

   constant ZEROS24  : std_logic_vector(23 downto 0)  := "000000000000000000000000";

   type state_type is (IDLE, RAM_WAIT, READ, WRITE_WAIT, WRITE);

    subtype byte is std_logic_vector(7 downto 0);
    subtype word is std_logic_vector(15 downto 0);
    subtype doubleword is std_logic_vector(31 downto 0);

    alias HOUR: std_logic_vector(4 downto 0) is utc(16 downto 12);
     alias MIN: std_logic_vector(5 downto 0) is utc(11 downto 6);
     alias SEC: std_logic_vector(5 downto 0) is utc(5 downto 0);

    alias ADR: std_logic_vector(lt_adr'range) is lt_adr;
    alias DIN: std_logic_vector(N-1 downto 0) is lt_dato;
    alias DOUT: std_logic_vector(N-1 downto 0) is lt_dati;

   signal state : state_type;

--REGISTERS
   signal ctrl_reg    : byte;
   signal alarm_reg   : doubleword;



begin  -- behav

-- ***********************************************************
PCI_FSM : process (clk, rstN)

begin  -- process PCI_FSM
    if rstN = '0' then                  -- Reset
        state <= IDLE;
    elsif clk'event and clk = '1' then  -- Rising edge of clock
        case state is
            when IDLE =>
                interrupt_clr <= "000000";

               if hit = '1' and lt_cmd0 = '1' then
               state <= WRITE_WAIT;      -- Write cycle started.
                elsif hit = '1' and lt_cmd0 = '0' and lt_ackn = '0' then
```

```
               if ADR(7 downto 6) = GPS_MSG_BASE and gps_ram_busy = '0' then
                  state <= RAM_WAIT;   -- Read cycle to RAM started
                    elsif ADR(7)='0' and ADR(6)='1' and gps_ram_busy='1' then
                  state <= IDLE;  -- RAM read cycle started, but RAM is busy....wait.
               else
                  state <= READ;   -- Other read cycle started
                              -- If ADDR not valid, DOUT=DIN.
               end if;
                else
                   state <= IDLE;   -- No Read/Write cycle started or lt_ackn not
asserted.
                end if;

         when RAM_WAIT =>
            if gps_ram_busy = '1' then
               state <= IDLE;
            else
               state <= READ;
            end if;
            when READ =>
               if (hit = '0' or lt_ackn = '0') then
                  state <= IDLE;
               else
                  state <= READ;
               end if;

--Write Cycle
         when WRITE_WAIT =>
            if hit='0' then
               state <= IDLE;
            elsif lt_ackn = '0' then
               state <= WRITE;
            else
               state <= WRITE_WAIT;
            end if;

         when WRITE =>
            case ADR is
               when CTRL_REG_BASE =>
                  ctrl_reg <= DIN(7 downto 0);
               when INTERRUPT_REG_BASE =>
                  interrupt_clr <= DIN(5 downto 0);
               when ALARM_REG_BASE =>
                  alarm_reg <= DIN;
               when others =>
            end case;

            if (hit = '1' and lt_ackn = '0' and lt_cmd0 = '1') then
               state <= WRITE;
            else
               state <= IDLE;
            end if;

            when others =>
               state <= IDLE;
        end case;
    end if;

end process PCI_FSM;
-- ***************************************************************
--Control Register Bits
tlt_build <= ctrl_reg(0);
gps_lock <= ctrl_reg(1);
utc_alarm_int_en <= ctrl_reg(2);
traim_soln_not_ok_int_en <= ctrl_reg(3);
pps_error_int_en <= ctrl_reg(4);
```

```
pps_edge_int_en <= ctrl_reg(5);
pos_msg_int_en <= ctrl_reg(6);
traim_msg_int_en <= ctrl_reg(7);

lt_rdyn <= '0' when state=READ else
         '0' when state=WRITE_WAIT else '1';

utc_alarm_value <= alarm_reg;

DOUT <= ZEROS24&ctrl_reg when ADR=CTRL_REG_BASE else
      "0000000000000000000"&auto_survey&traim_msg_valid&traim_msg_new&no_traim_msg&
      pos_msg_valid&pos_msg_new&no_pos_msg&gps_1pps&pps_error&traim_soln&
      insuff_sats&utc_alarm when ADR=STATUS_REG_BASE else
      "00"&ZEROS24&interrupt_in when ADR=INTERRUPT_REG_BASE else
      alarm_reg when ADR=ALARM_REG_BASE else
      "00000000000"&HOUR&"00"&MIN&"00"&SEC when ADR=UTC_REG_BASE else
-- ***************************************************************
-- Modified to allow access to the free-running PAB counter
-- ***************************************************************
      c_count(31 downto 0) when ADR=PPS_CNT_LO_BASE else
      "0000"&c_count(59 downto 32) when ADR=PPS_CNT_HI_BASE else
-- ***************************************************************
-- End modification
-- ***************************************************************
      avg_cnt when ADR=AVG_CNT_REG_BASE else
      ZEROS24&temperature when ADR=TEMPERATURE_BASE else
      ZEROS24&gps_ramq when ADR(7 downto 6) = GPS_MSG_BASE else
      DIN;
-- ***************************************************************
-- ***************************************************************
end behav;
```

APPENDIX C

DATA COLLECTION SOFTWARE

There are three programs used in this research to collect timestamp data. The first program, *cpu_tlt*, is used to gain insight into the relationship between the CPU and PCI clocks. This program is very useful in determining if the computing system, the PAB board, and the GPS receiver are configured correctly. The *cpu_tlt* program outputs the time difference between the start of the TLT and the corresponding CPU count. Even when a long period of time has elapsed since the computer was rebooted, this difference should be approximately two or three seconds if the machine is synchronous. The full listing of the source code for *cpu_tlt* is provided. All of the data collection programs listed in this Appendix are located at:

/ccs/issl/general/ca/UltraScope/software/rbarnes/async.

C.1 Full source code listing for *cpu_tlt*

```
/*************************************************************************
   tlt.c
*************************************************************************/


/*************************************************************************
   Includes
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>

#ifndef INTEL
#define SPARC
#endif

#include "pabutil.h"


/*************************************************************************
   Defines
*************************************************************************/
#define NUM_SECS 10


/*************************************************************************
   Typedefs
*************************************************************************/
typedef union
{
    unsigned long long f;
```

```
    unsigned long      hl[2];
} CPU_COUNT;


/************************************************************************
   Function Definitions
************************************************************************/


int main (int argc, char *argv[])
{
    ONCORE_RESPONSE      oncore_response;
    ONCORE_PSD_RESPONSE *psd_response;
    int                  status;
    PABUTIL_TLT_HEADER          tlt_header;
    CPU_COUNT                   tscs [NUM_SECS];
    unsigned long long          nsecs;
    unsigned long              *tlt_data;
    unsigned long              *tlt_temp;
    int                         i, seconds, error;

    error = PABUTIL_Map_board ();
    if (error != PABUTIL_SUCCESS)
    {
        printf ("Error mapping PAB board\n");
        exit (1);
    }

    /*    error = PABUTIL_Fill_mem (0, 0x80000, 0xffffffff);
    if (error != PABUTIL_SUCCESS)
    {
        printf ("Error filling memory\n");
        PABUTIL_Unmap_board ();
        exit (1);
       }*/

    error = PABUTIL_Start_tlt ();
    if (error != PABUTIL_SUCCESS)
    {
        printf ("Error turning on TLT\n");
        PABUTIL_Unmap_board ();
        exit (1);
    }

    psd_response = (ONCORE_PSD_RESPONSE *) &oncore_response;

    for (i = 0; i < 4; i++)
    {
        status = PABUTIL_Get_msg (&oncore_response);
        while ((status == PABUTIL_SUCCESS) &&
               (psd_response->type != PSD_RESPONSE_TYPE))
        {
            status = PABUTIL_Get_msg (&oncore_response);
        }
    }

    for (i = 0; i < NUM_SECS; i++)
    {
#ifdef INTEL
        PABUTIL_GET_CPU_COUNT (tscs[i].hl[1], tscs[i].hl[0]);
#else
        PABUTIL_GET_CPU_COUNT (tscs[i].hl[0], tscs[i].hl[1]);
#endif
       if (i == 0) {
         printf ("\t%llu\n", tscs[i].f);
       } else {
```

```
      printf ("\t%llu ==> diff %llu\n",tscs[i].f,tscs[i].f-tscs[i-1].f);
    }
    status = PABUTIL_Get_msg (&oncore_response);
    while ((status == PABUTIL_SUCCESS) &&
           (psd_response->type != PSD_RESPONSE_TYPE))
    {
        status = PABUTIL_Get_msg (&oncore_response);
    }
}

error = PABUTIL_Get_tlt_header(&tlt_header);
if (error != PABUTIL_SUCCESS) {
  printf("Error getting header\n");
  PABUTIL_Unmap_board ();
  exit(1);
}

printf("Seconds since boot according to first CPU count %5.2f\n",
       tscs[0].f/(7*33.333E6));
printf("Second since boot accoudgin to PAB count %5.2f\n",
       tlt_header.pab_cnt/33.333E6);

error = PABUTIL_Stop_tlt ();
if (error != PABUTIL_SUCCESS)
{
    printf ("Error turning off TLT\n");
    PABUTIL_Unmap_board ();
    exit (1);
}

printf ("TLT Header\n");
printf ("    Date:           %02d/%02d/%04d\n",
        (int) tlt_header.month,
        (int) tlt_header.day,
        (int) tlt_header.year);
printf ("    Time:           %02d:%02d:%02d\n",
        (int) tlt_header.hours,
        (int) tlt_header.minutes,
        (int) tlt_header.seconds);
printf ("    PAB Count:      %llu\n",
        tlt_header.pab_cnt);
printf ("    PAB Int Count:  %x(%lu)\n",
        tlt_header.pab_int_cnt,tlt_header.pab_int_cnt);
printf ("    Init Temp:      %4.3f\n",
        tlt_header.starting_temp );

PABUTIL_Get_tlt_data_ptr (&tlt_data);

printf("\n\nDump of tlt data\n");
for (i = 0; i < (NUM_SECS + 1) / 2; i++)
{
    printf ("    Data:           %08x\n", *(tlt_data + i));
}

printf ("%d successive CPU clock readings:\n\n", NUM_SECS);
for (i = 0; i < NUM_SECS; i++)
{
    unsigned long long last_nsecs;

    error = PABUTIL_Cpu_count_to_nsecs (tscs[i].f, &nsecs);
    if (error != PABUTIL_SUCCESS)
    {
        printf ("Error converting count to nanoseconds\n");
        PABUTIL_Unmap_board ();
        exit (1);
    }
```

```
        printf ("\t%llu ==> %llu", tscs[i].f, nsecs);
        if (i == 0)
        {
            printf ("\n");
        }
        else
        {
            printf ("\t(diff = %llu)\n", nsecs - last_nsecs);
        }
        last_nsecs = nsecs;
    }

    error = PABUTIL_Unmap_board ();
    if (error != PABUTIL_SUCCESS)
    {
        printf ("Error unmapping PAB board\n");
        exit (1);
    }

    return 0;
}
```

Another useful program is *timestamp*. The *timestamp* program returns a number of statistics about the computing system. The statistics estimated by this program are an estimate of the clock frequency ratio, the roundtrip delay between the processor and the PAB board, the time offset between the CPU and PAB counters, and the amount of drifting that occurs between the CPU and PAB counters. This program creates a series of files that contains these statistics for each timestamp sequence created. The timestamp sequence used in this program is based directly on NTP. The full source listing is provided in Section C.2.

C.2 Full source code listing for *timestamp*

```
#include <stdio.h>
#include <fcntl.h>
#include <math.h>
#include <string.h>

#ifndef INTEL
#define SPARC
#endif

#include "perfmon.h"
#include "pabutil.h"
#include "timestamp.h"
```

```
#define RTDELAY "_rtdelay.dat"
#define CLKOFF  "_clkoff.dat"
#define DRIFT   "_drift.dat"
#define COUNTS  "_counts.dat"

int main(int argv, char *argc[])
{
   /* Timestamp arrays */
   /* 1st timestamp in transaction at i */
   /* 2nd timestamp in transaction at i+num_itr */
   CPU_COUNT *cpu_count;
   PAB_COUNT *pab_count;

   /* Statistical data about timestamps */
   double *rt_delay = NULL;
   double *c_off = NULL;

   /* Drift data */
   double *o_drift = NULL;
   int    o_num_drift = 0;

   /* Average statistics of the timestamps */
   double avg_rt_delay = 0.0;
   double avg_c_off = 0.0;


   /* CPU and PAB speed info */
   int factor = 0;
   double pab_hz, cpu_hz;
   double pab_tick, cpu_tick;

   /* Handle cmd line args */
   int sleep_mode = 0;
   int sleep_time = 0;
   int num_itr = 0;

   /* Output file variables */
   char  fname[255];
   char  prefix[255];
   FILE *rt_file = NULL;
   FILE *c_off_file = NULL;
   FILE *drift_file = NULL;
   FILE *counts_file = NULL;

   /* General and device variables */
   int fd, rc;
   int error, status;
   int i, j;


   /* Get cmd line args */
   if (argv != 1)
   {
      num_itr = get_num_itrs(argv, argc);
      sleep_mode = get_sleep_mode(argv, argc);
      if (sleep_mode == SLEEP_ACTIVE && argv == 5)
      {
         sleep_time = get_sleep_time(argv, argc);
         strcpy(prefix, argc[4]);
      }
      else if (sleep_mode == PPS_ACTIVE && argv == 4)
      {
         strcpy(prefix, argc[3]);
      }
      else if (sleep_mode == NO_WAIT && argv == 3)
```

```
      {
         strcpy(prefix, argc[2]);
      }
      else
         print_usage();
   }
   else
      print_usage();


   /* Allocate space for variables */
   cpu_count = (CPU_COUNT *)malloc(2*num_itr * sizeof(CPU_COUNT));
   pab_count = (PAB_COUNT *)malloc(2*num_itr * sizeof(PAB_COUNT));
   o_drift   = (double *)malloc(num_itr * sizeof(double));


   /* Open perfmon */
   fd = open("/dev/perfmon", O_RDONLY);
   if (fd == -1)
   {
      perror("open(/dev/perfmon)");
      exit(1);
   }

#ifdef INTEL
   /* Enable reading of the TSC register and PerfCtrs */
   rc = ioctl(fd, TSC_PERFCTRS_EN);
   if (rc < 0)
   {
      perror("ioctl(TSC_PERFCTRS_EN)");
      exit(1);
   }
#endif

   /* Map in PAB Board */
   error = PABUTIL_Map_board ();
   if (error != PABUTIL_SUCCESS)
   {
      printf ("Error mapping PAB board\n");
      exit (1);
   }

   /* Get PAB registers ptr */
   PABUTIL_Get_regs_ptr(&pab_regs);



   /* Get PAB clock speed */
   pab_hz = PCI_CLK;
   pab_tick = 1.0 / pab_hz;

   /* Compute factor */
   factor = compute_factor();

   /* Get CPU clock speed */
   cpu_hz = pab_hz * factor;
   cpu_tick = 1.0 / cpu_hz;

   /* Print details */
   printf("CPU Freq:\t%5.3f MHz\tCPU Tick:\t%5.3f ns\n",
          cpu_hz / 1000000.0, cpu_tick * 1000000000.0);
   printf("PAB Freq:\t%5.3f MHz\tPAB Tick:\t%5.3f ns\n",
          pab_hz / 1000000.0, pab_tick * 1000000000.0);
   printf("CPU is %d times faster than the PAB\n",
          factor);
   printf("\n");
```

```
   /* Empty GPS message fifo */
   for (i = 0; i < 5; i++)
      wait_sec();

   /* Collect timestamps*/
   read_timestamp(cpu_count, pab_count, (double)factor, sleep_mode, sleep_time, num_itr);

   /* Calculate Results */
   rt_delay = roundtrip_delay(cpu_count, pab_count, num_itr, (double)factor);
   c_off = clock_offset(cpu_count, pab_count, num_itr, (double)factor);
   overall_drift(cpu_count, pab_count, num_itr, (double)factor, pab_hz, o_drift,
&o_num_drift);

   avg_rt_delay = mean(rt_delay, num_itr);
   avg_c_off    = mean(c_off   , num_itr);


   /* Print results */
   printf("Number of packet iterations: %d\n", num_itr);

   printf("RT Delay mean:  %9.4f us\t",
          avg_rt_delay*cpu_tick*1000000.0);
   printf("RT Delay variance:  %9.4f us\n",
          variance(rt_delay, num_itr)*cpu_tick*1000000.0);

   printf("Clk off mean :  %9.4f us\t",
          avg_c_off*cpu_tick*1000000.0);
   printf("Clk off variance :  %9.4f us\n",
          variance(c_off, num_itr)*cpu_tick*1000000.0);

   printf("\n");


   printf("\t\tRT DELAY (CPU)\n");
   histogram(rt_delay, num_itr, 15);

   printf("\t\tCLK OFFSET (CPU)\n");
   histogram(c_off, num_itr, 15);

   printf("\t\tOVERALL CLOCK DRIFT (CPU COUNTS/CPU COUNT)\n");
   histogram(o_drift, o_num_drift, 15);


   /* Output counts to a file */
   strcpy(fname, prefix);
   strcat(fname, COUNTS);
   counts_file = fopen(fname, "w");

   for (i = 0; i < (num_itr+num_itr); i++)
   {
      if (i < num_itr)
         fprintf(counts_file, "%llu\t%llu\n",
                 cpu_count[i].f,
                 pab_count[i]*factor);
      else
         fprintf(counts_file, "%llu\t%llu\n",
                 cpu_count[i].f,
                 pab_count[i]*factor);
   }
   fclose(counts_file);

   /* Output rt_delay to a file */
   strcpy(fname, prefix);
```

```
   strcat(fname, RTDELAY);
   rt_file = fopen(fname, "w");

   for (i = 0; i < num_itr; i++)
   {
      fprintf(rt_file, "%9.4f\n",
              rt_delay[i]);
   }
   fclose(rt_file);

   /* Output clk offset to a file */
   strcpy(fname, prefix);
   strcat(fname, CLKOFF);
   c_off_file = fopen(fname, "w");

   for (i = 0; i < num_itr; i++)
   {
      fprintf(c_off_file, "%f\n",
              c_off[i]);
   }
   fclose(c_off_file);

   /* Output drift to a file */
   strcpy(fname, prefix);
   strcat(fname, DRIFT);
   drift_file = fopen(fname, "w");

   for (i = 0; i < num_itr; i++)
   {
      fprintf(drift_file, "%9.4f\n",
              o_drift[i]*1e9);
   }
   fclose(drift_file);



   /* Close up and clean up */
   error = PABUTIL_Unmap_board ();
   if (error != PABUTIL_SUCCESS)
   {
      printf ("Error unmapping PAB board\n");
      exit (1);
   }

   free(cpu_count);
   free(pab_count);
   free(rt_delay);
   free(c_off);
   free(o_drift);

   return 0;
}

/* Rounds a floating point number to the nearest integer */
int round(double number)
{
   int rval;

   if (number > floor(number) + 0.5)
      rval = ceil(number);
   else
      rval = floor(number);

   return rval;
}
```

```
/* Computes the mean value for a data set */
double mean(double *data_vec, int data_len)
{
   int i = 0;
   double data_mean = 0.0;

   for (i = 0; i < data_len; i++)
      data_mean += data_vec[i];

   data_mean /= (double)data_len;

   return data_mean;
}

/* Computes the variance for a data set */
double variance(double *data_vec, int data_len)
{
   int i = 0;

   double data_mean = 0.0;
   double mean_sq = 0.0;
   double var = 0.0;

   double temp = 0.0;

   data_mean = mean(data_vec, data_len);

   for (i = 0; i < data_len; i++)
   {
      temp = (double)data_vec[i] - (double)data_mean;
      temp = fabs(temp) * fabs(temp);

      var = var + temp;
   }

   var = var / (double)(data_len - 1);
   return var;
}

/* Using the CPU and PAB timestamps, computes the roundtrip delay */
/* For each "packet" of timestamps                                */
double *roundtrip_delay(CPU_COUNT *cpu_count, PAB_COUNT *pab_count, int data_len, double
factor)
{
   double *rt_delay = NULL;
   int i = 0;

   rt_delay = (double *)malloc(data_len * sizeof(double));

   for (i = 0; i < data_len; i++)
   {
      rt_delay[i] = ((double)cpu_count[i+data_len].f - (double)cpu_count[i].f) -
                    ((double)pab_count[i+data_len]   - (double)pab_count[i]) * factor;
   }
   return rt_delay;
}

/* Using the CPU and PAB timestamps, computes the clock offset    */
/* between the PAB and CPU clocks for each "packet" of timestamps */
double *clock_offset(CPU_COUNT *cpu_count, PAB_COUNT *pab_count, int data_len, double
factor)
{
   double *c_off = NULL;
   int i = 0;
```

```
   double temp;

   c_off = (double *)malloc(data_len * sizeof(double));

   for (i = 0; i < data_len; i++)
   {
      temp = (double)(pab_count[i]) * factor;
      temp -= (double)(cpu_count[i].f);
      temp += (double)(pab_count[i+data_len]) * factor;
      temp -= (double)(cpu_count[i+data_len].f);
      temp /= 2.0;

      c_off[i] = temp;
   }
   return c_off;
}

void overall_drift(CPU_COUNT *cpu_count, PAB_COUNT *pab_count, int data_len, double
factor,
                   double pab_hz, double *drift, int *drift_len)
{
   double X1, X2;

   double time_int = 0.9;
   double e_time = 0.0;
   double drift_mean = 0.0;

   int n = 0;
   int m = 0;
   int i = 0;
   int num_drift = 0;

   X1 = (double)cpu_count[0].f - (double)pab_count[0]*factor;

   for (i = 0; i < data_len; i++)
   {
      n++;

      e_time += (double)(cpu_count[i].f - cpu_count[m].f) / (pab_hz*factor);
      if (e_time >= time_int)
      {
         X2 = (double)cpu_count[i].f - (double)pab_count[i]*factor;

         /* Do drift calculations */
         drift[num_drift] = (X2 - X1) / (e_time * pab_hz * factor);

         X1 = X2;
         n = 0;
         m = i;
         num_drift++;
         e_time = 0.0;
      }
   }

   /* Remove mean value */
   drift_mean = mean(drift, num_drift);
   for (i = 0; i < num_drift; i++)
      drift[i] -= drift_mean;

   (*drift_len) = num_drift;
}

void histogram(double *data_vec, int data_len, int num_bins)
{
   int i = 0;
   int j = 0;
```

```
   int k = 0;

   double data_min = data_vec[0];
   double data_max = data_vec[0];
   double int_size = 0.0;

   int *bins;
   double start;
   double end;

   bins = (int *)malloc(num_bins * sizeof(int));

   /* Establish range */
   for (i = 0; i < data_len; i++)
   {
      if (data_vec[i] > data_max)
         data_max = data_vec[i];

      if (data_vec[i] < data_min)
         data_min = data_vec[i];
   }

   int_size = (data_max - data_min) / num_bins;
   for (i = 0; i < num_bins; i++)
      bins[i] = 0;

   /* Fill bins */
   start = data_min;
   end = data_min + int_size;
   for (i = 0; i < num_bins; i++)
   {
      for (j = 0; j < data_len; j++)
      {
         if (data_vec[j] >= start && data_vec[j] < end)
            bins[i]++;

         if (i == num_bins - 1 && data_vec[j] >= end)
            bins[i]++;
      }

      start += int_size;
      end   += int_size;
   }

   printf("\t\tHistogram\n");
   printf("Data min:  %9.4f\tData max:  %9.4f\n\n",
          data_min, data_max);

   for (i = 0; i < num_bins; i++)
      printf("%d  ", bins[i]);

   printf("\n\n\n");
   free(bins);
}

void wait_sec()
{
   int status;
   ONCORE_RESPONSE oncore_response;
   ONCORE_PSD_RESPONSE *psd_response;

   psd_response = (ONCORE_PSD_RESPONSE *) &oncore_response;

   /* Wait for 1pps */
   status = PABUTIL_Get_msg (&oncore_response);
   while ((status == PABUTIL_SUCCESS) &&
```

```
          (psd_response->type != PSD_RESPONSE_TYPE))
   {
      status = PABUTIL_Get_msg (&oncore_response);
   }
}

int compute_factor()
{
   float cpu_factor = 0.0;
   int factor = 0;

   volatile PAB_COUNT current_count;
   volatile CPU_COUNT cpu_speed[1];

   //cpu_speed[0].f = read_tsc();
#ifdef INTEL
   PABUTIL_GET_CPU_COUNT(cpu_speed[0].hl[1], cpu_speed[0].hl[0]);
#else
   PABUTIL_GET_CPU_COUNT(cpu_speed[0].hl[0], cpu_speed[0].hl[1]);
#endif

   sleep(10);
   //cpu_speed[1].f = read_tsc();
#ifdef INTEL
   PABUTIL_GET_CPU_COUNT(cpu_speed[1].hl[1], cpu_speed[1].hl[0]);
#else
   PABUTIL_GET_CPU_COUNT(cpu_speed[1].hl[0], cpu_speed[1].hl[1]);
#endif

   cpu_factor = (float)(cpu_speed[1].f - cpu_speed[0].f) / 10.0;
   cpu_factor /= PCI_CLK;
   factor = round(cpu_factor);

   return factor;
}

int read_timestamp(volatile CPU_COUNT *cpu_count, volatile PAB_COUNT *pab_count, double
factor,
                   int sleep_mode, int sleep_time, int num_itr)
{
   int i = 0;

   double diff_cpu = 0.0;
   double diff_pab = 0.0;

   PAB_COUNT temp;
   volatile PAB_COUNT current_count;

   for (i = 0; i < num_itr; i++)
   {
      if (sleep_mode == PPS_ACTIVE)
         wait_sec();
      else if (sleep_mode == SLEEP_ACTIVE)
         usleep(sleep_time);

      /* Read CPU count */
      //cpu_serialize();
      //cpu_count[i].f = read_tsc();
#ifdef INTEL
      PABUTIL_GET_CPU_COUNT(cpu_count[i].hl[1], cpu_count[i].hl[0]);
#else
      PABUTIL_GET_CPU_COUNT(cpu_count[i].hl[0], cpu_count[i].hl[1]);
#endif

      /* To try to force sequential order */
      current_count = cpu_count[i].f + cpu_count[i].f;
```

```
      /* Read PAB count */
      //cpu_serialize();
      temp = *(pab_regs + LO_PAB);
      pab_count[i] = *(pab_regs + HI_PAB);
      pab_count[i] = (pab_count[i] << 32) + temp;

      /* To try to force sequential order */
      current_count = pab_count[i] + cpu_count[i].f;

      /* Read PAB count */
      //cpu_serialize();
      temp = *(pab_regs + LO_PAB);
      pab_count[i+num_itr] = *(pab_regs + HI_PAB);
      pab_count[i+num_itr] = (pab_count[i+num_itr] << 32) + temp;

      /* To try to force sequential order */
      current_count = pab_count[i+num_itr] + cpu_count[i].f;

      /* Read CPU count */
      //cpu_serialize();
      //cpu_count[i+num_itr].f = read_tsc();
#ifdef INTEL
      PABUTIL_GET_CPU_COUNT(cpu_count[i+num_itr].hl[1], cpu_count[i+num_itr].hl[0]);
#else
      PABUTIL_GET_CPU_COUNT(cpu_count[i+num_itr].hl[0], cpu_count[i+num_itr].hl[1]);
#endif
   }
}

int get_sleep_mode(int argv, char *argc[])
{
   if (argv > 2)
   {
      if (strcmp(argc[2], "-1pps") == 0)
         return PPS_ACTIVE;
      else if (strcmp(argc[2], "-sleep") == 0)
         return SLEEP_ACTIVE;
   }
   else
      return NO_WAIT;
}

int get_sleep_time(int argv, char *argc[])
{
   if (argv > 3)
      return atoi(argc[3]);
   else
      return 0;
}

int get_num_itrs(int argv, char *argc[])
{
   return atoi(argc[1]);
}

void print_usage()
{
   printf("USAGE:\n");
   printf("ts n [-1pps | -sleep m] f_prefix\n");
   printf("   n       :  the number of timestamp iterations\n");
   printf("   -1pps   :  wait 1 second between timestamps using GPS 1pps\n");
   printf("   -sleep  :  wait for m microseconds between timestamps\n");
   printf("   f_prefix:  prefix to add to output files\n");
   printf("\n");
   printf("If no sleep method is specified, ts will collect\n");
```

```
   printf("timestamps as fast as possible\n");
   printf("\n");

   exit(-1);
}
```

The final program used to collect timestamp data is the *time_seq* program. The *time_seq* program is collected timestamps using modified version of NTP. This program creates a timestamp sequence by reading the CPU counter, C_0, then the PAB counter, P_0, and then the CPU counter again, C_1. This method is illustrated in Fig. 3.2. The unprocessed timestamp data is dumped to a file for later analysis. The temperature at each timestamp sequence is recorded and dumped to another file. This allows the temperature dependence of the timestamp sequence to be investigated. Section C.3 provides a full listing of the source code for *time_seq*.

C.3 Full source code listing for *time_seq*

```
#include <stdio.h>
#include <fcntl.h>
#include <math.h>
#include <string.h>

#ifndef INTEL
#define SPARC
#endif

#include "perfmon.h"
#include "pabutil.h"
#include "time_seq.h"

#define SECITR 15
#define NITR 5
#define NUM_TS 30

int main(int argv, char *argc[])
{
   /* Timestamp arrays */
   /* 1st timestamp in transaction at i */
   /* 2nd timestamp in transaction at i+num_itr */
   CPU_COUNT *cpu_count;
   PAB_COUNT *pab_count;
   double *temperature;

   /* Output file variables */
```

```
   FILE *data_file = NULL;
   FILE *temp_file = NULL;


   /* General and device variables */
   int fd, rc;
   int error, status;
   int i, j, k, l;

   int num_itr = NUM_TS;

   /* Allocate space for variables */
   cpu_count = (CPU_COUNT *)malloc(2*num_itr * sizeof(CPU_COUNT));
   pab_count = (PAB_COUNT *)malloc(2*num_itr * sizeof(PAB_COUNT));
   temperature = (double *)malloc(num_itr * sizeof(double));


   /* Open perfmon */
   fd = open("/dev/perfmon", O_RDONLY);
   if (fd == -1)
   {
      perror("open(/dev/perfmon)");
      exit(1);
   }

   /* Enable reading of the TSC register and PerfCtrs */
#ifdef INTEL
   rc = ioctl(fd, TSC_PERFCTRS_EN);
   if (rc < 0)
   {
      perror("ioctl(TSC_PERFCTRS_EN)");
      exit(1);
   }
#endif

   /* Map in PAB Board */
   error = PABUTIL_Map_board ();
   if (error != PABUTIL_SUCCESS)
   {
      printf ("Error mapping PAB board\n");
      exit (1);
   }

   /* Get PAB registers ptr */
   PABUTIL_Get_regs_ptr(&pab_regs);

   /* Empty GPS message fifo */
   for (i = 0; i < 5; i++)
      wait_sec();

   data_file = fopen("new_data.dat", "w");
   temp_file = fopen("new_temp.dat", "w");

   printf("\n");
   printf("Number of packet iterations: %d\n", num_itr);
   printf("\n");

   /* Collect timestamps*/
   for (i = 0; i < NITR; i++)
   {
      for (k = 0; k < SECITR; k++)
         wait_sec();

      read_timestamp(cpu_count, pab_count, temperature, num_itr);
```

```
      for (j = 0; j < num_itr-1; j++)
      {
         fprintf(data_file, "%llu\t%llu\t%llu\t\t",
                 cpu_count[j].f, cpu_count[j+num_itr].f,
                 pab_count[j]);

         fprintf(data_file, "%llu\t%llu\t%llu\n",
                 cpu_count[j+1].f, cpu_count[j+1+num_itr].f,
                 pab_count[j+1]);

         fprintf(temp_file, "%3.4f\n", temperature[j]);
      }
      fprintf(data_file, "\n");
      fprintf(temp_file, "\n");

      fflush(NULL);
   }


   /* Close up and clean up */
   error = PABUTIL_Unmap_board ();
   if (error != PABUTIL_SUCCESS)
   {
      printf ("Error unmapping PAB board\n");
      exit (1);
   }

   fclose(data_file);
   fclose(temp_file);

   free(cpu_count);
   free(pab_count);
   free(temperature);

   return 0;
}

void wait_sec()
{
       volatile unsigned int current_status;

       /* Wait for 1PPS Low */
       current_status = *(pab_regs + STATUS_REG) & PPS_MASK;
       while(current_status != 0x00000000)
              current_status = *(pab_regs + STATUS_REG) & PPS_MASK;

       /* Wait for 1PPS High */
       while (current_status == 0)
       {
              current_status = *(pab_regs + STATUS_REG) & PPS_MASK;
       }

}

int read_timestamp(volatile CPU_COUNT *cpu_count, volatile PAB_COUNT *pab_count,
                   volatile double *temperature,  int num_itr)
{
   int i = 0;
   int j = 0;


   PAB_COUNT temp;
   volatile PAB_COUNT current_count;


   for (i = 0; i < num_itr; i++)
```

```
   {
      wait_sec();

      /* Read CPU count */
//      cpu_serialize();
//      cpu_count[i].f = read_tsc();
#ifdef INTEL
      PABUTIL_GET_CPU_COUNT(cpu_count[i].hl[1], cpu_count[i].hl[0]);
#else
      PABUTIL_GET_CPU_COUNT(cpu_count[i].hl[0], cpu_count[i].hl[1]);
#endif

      /* To try to force sequential order */
      current_count = cpu_count[i].f + cpu_count[i].f;

      /* Read PAB count */
      temp = *(pab_regs + LO_PAB);
      pab_count[i] = *(pab_regs + HI_PAB);
      pab_count[i] = (pab_count[i] << 32) + temp;

      /* To try to force sequential order */
      current_count = pab_count[i] + cpu_count[i].f;


      /* Read CPU count */
//      cpu_serialize();
//      cpu_count[i+num_itr].f = read_tsc();
#ifdef INTEL
      PABUTIL_GET_CPU_COUNT(cpu_count[i+num_itr].hl[1], cpu_count[i+num_itr].hl[0]);
#else
      PABUTIL_GET_CPU_COUNT(cpu_count[i+num_itr].hl[0], cpu_count[i+num_itr].hl[1]);
#endif

      /* To try to force sequential order */
      current_count = pab_count[i+num_itr] + cpu_count[i+num_itr].f;

      /* Read temperature */
      temperature[i] = (*(pab_regs+TPTR) & 0x00000080) ?
         (*(pab_regs+TPTR) | 0x00000100) & 0xffffff7f :
         (*(pab_regs+TPTR) | 0x00000080);
      temperature[i] = temperature[i] / 8.0;
   }
}
```

APPENDIX D

DATA ANALYSIS SOFTWARE

Matlab scripts were used to analyze the timestamp data returned by the data collection programs. The Matlab scripts listed in this Appendix are located at: */ccs/issl/general/ca/UltraScope/software/rbarnes/async/matlab*.

The first script, listed in Section D.1, uses the properties of a synchronous computing system to estimate the performance of the algorithm used for the asynchronous case. This script, *time_analysis.m*, investigates the errors associated with the PCI bus correction given by (3-6). The errors due to an approximation of the clock frequency ratio, the operating system delays, and the delays due to reading the CPU counter can be estimated and investigated in this script.

D.1 Full source code listing for *time_analysis.m*

```
clear;
clc;

% Open data file
fid = fopen('H:\users\ooglesby\erc\thesis\data\pci_busy_pcdata.dat', 'r');

% Read data in from file
% Each line of the timestamp file has two seperate and consequtive timestamps
tstamp = fscanf(fid, '%f\t%f\t%f\t\t%f\t%f\t%f\n', [6, inf]);
fclose(fid);

cpu1 = tstamp(1:2, :);
cpu2 = tstamp(4:5, :);
pab1 = tstamp(3, :);
pab2 = tstamp(6, :);

% Compute the time offset
offset = (2*pab1(1, :)*7) - (cpu1(1, :) + cpu1(2, :));
offset = mean(offset / 2);

p1 = (pab1(1, :) * 7) - offset;
c1 = cpu1(1, :) + offset;
c2 = cpu1(2, :) + offset;

% Compute the correction with R approximation
E = pab1(1, :) .* (cpu1(1, :) ./ cpu1(2, :));
E = (E .* 7) - offset;

% Compute the correction with R exactly
Ep = pab1(1, :) - ((cpu1(2, :) - cpu1(1, :)) ./ 7);
Ep = (Ep .* 7) - offset;

% Create regions x, y, z
```

```
x = cpu1(1, :) - Ep;
y = (pab1(1, :) .* 7) - offset - cpu1(1, :);
z = cpu1(2, :) - (pab1(1, :) .* 7) + offset;

figure;
subplot(3, 1, 1); plot(x); grid
title('');
xlabel('Samples (Tevent)');
ylabel('CPU counts');
subplot(3, 1, 2); plot(y); grid
title('');
xlabel('Samples (TPCI)');
ylabel('CPU counts');
subplot(3, 1, 3); plot(z); grid
title('');
xlabel('Samples (Tend)');
ylabel('CPU counts');

figure;
plot((Ep - E) ./ 33 .* 100); grid
title('');
xlabel('Samples');
ylabel('Time (ns)');

figure;
subplot(3, 1, 1); hist(x, 25); grid
subplot(3, 1, 2); hist(y, 25); grid
subplot(3, 1, 3); hist(z, 25); grid

[mean(x) cov(x)]
[mean(y) cov(y)]
[mean(z) cov(z)]

figure;
hist(Ep-E);

[mean(Ep-E) cov(Ep-E)]



figure;
subplot(3, 1, 1); psd(x);
subplot(3, 1, 2); psd(y);
subplot(3, 1, 3); psd(z);
```

The other data analysis script used simulated the effects of PCI bus arbitration on the performance of the asynchronous timestamp and correction algorithm. The *pci_sim.m* script simulates two bus master devices on the PCI bus. The parameters for the arbitration delays were the typical values that are discussed in the PCI bus specification. The simulation adds the arbitration delays randomly to actual PCI bus timestamp data. The amount of arbitration can be set as a percentage of the overall number of transactions. The output of this script is a set of graphs that show the delays

associated with the timestamp sequence with and without PCI bus arbitration. The source code to this simulation is provided below.

D.2 Full source code listing for *pci_sim.m*

```
% PCI Arbitration simulation
%
% Lightly loaded system
% Arbitrates for 10% of the PCI acessses
% Arbitration is based on examples in PCI spec (pg 69)
% Latency timer of 22 PCI clocks

clear;
clc;

ArbAccess = 3;
LatencyTimer = 22;
NumMaster = 2;
TtlTime = LatencyTimer + 2;

FIGURES = 1

% Open data file
fid = fopen('H:\users\ooglesby\erc\thesis\data\pci_idle_pcdata.dat', 'r');

% Read data in from file
tstamp = fscanf(fid, '%f\t%f\t%f\t\t%f\t%f\t%f\n', [6, inf]);
fclose(fid);

cpu1 = tstamp(1:2, :);
cpu2 = tstamp(4:5, :);
pab1 = tstamp(3, :);
pab2 = tstamp(6, :);

offset = (2*pab1(1, :)*7) - (cpu1(1, :) + cpu1(2, :));
offset = mean(offset / 2);

DLen = length(tstamp);
%RSeed = floor(rand(1, 1) .* 10.0);

% Add arbitration delay
count = 0;
for k = 1:DLen,
   RSeed = floor(rand(1, 1) .* 100.0);

   if (RSeed >= (100 - ArbAccess))
      arb_pab1(k) = pab1(1, k) + (TtlTime .* NumMaster);
      arb_cpu1(k) = cpu1(2, k) + (TtlTime .* NumMaster .* 7);
      count = count + 1;
   else
      arb_pab1(k) = pab1(1, k);
      arb_cpu1(k) = cpu1(2, k);

   end;
end;


% Apply correction
arb_E = arb_pab1 .* (cpu1(1, :) ./ arb_cpu1);
```

```
arb_E = (arb_E .* 7) - offset;

E = pab1(1, :) .* (cpu1(1, :) ./ cpu1(2, :));
E = (E .* 7) - offset;

arb_x = cpu1(1, :) - E;
arb_y = ((arb_pab1 .* 7) - offset) - cpu1(1, :);
arb_z = arb_cpu1 - ((arb_pab1 .* 7) - offset);

x = cpu1(1, :) - E;
y = ((pab1(1, :) .* 7) - offset) - cpu1(1, :);
z = cpu1(2, :)-((pab1(1, :) .* 7) - offset);

% Plot Results
if (FIGURES == 1)
   figure;
   subplot(3, 1, 1); plot(x); grid
   title('');
   xlabel('Samples (Tevent)');
   ylabel('CPU counts');

   subplot(3, 1, 2); plot(y); grid
   title('');
   xlabel('Samples (TPCI)');
   ylabel('CPU counts');

   subplot(3, 1, 3); plot(z); grid
   title('');
   xlabel('Samples (Tend)');
   ylabel('CPU counts');

   figure; hold on;
   subplot(3, 1, 1);
   hold on;
   plot(arb_x, 'r');
   plot(x, 'b'); grid
   hold off;
   title('');
   xlabel('Samples (Tevent)');
   ylabel('CPU counts');


   subplot(3, 1, 2);
   hold on;
   plot(arb_y, 'r');
   plot(y, 'b'); grid
   hold off;
   title('');
   xlabel('Samples (TPCI)');
   ylabel('CPU counts');


   subplot(3, 1, 3);
   hold on;
   plot(arb_z, 'r');
   plot(z, 'b'); grid
   hold off;
   title('');
   xlabel('Samples (Tend)');
   ylabel('CPU counts');

end;

count ./ DLen .* 100

figure;
```

```
subplot(3, 1, 1); hist(arb_x, 25); grid
subplot(3, 1, 2); hist(arb_y, 25); grid
subplot(3, 1, 3); hist(arb_z, 25); grid

[mean(arb_x) cov(arb_x)]
[mean(arb_y) cov(arb_y)]
[mean(arb_z) cov(arb_z)]


figure;
subplot(3, 1, 1); psd(arb_x);
subplot(3, 1, 2); psd(arb_y);
subplot(3, 1, 3); psd(arb_z);
```