An Accurate Time-management Unit for Real-time Processors*

Krishnan Kailas
Department of Electrical & Computer Engineering
University of Maryland
College Park, MD 20742, USA
krish@cs.umd.edu

Ashok Agrawala
Department of Computer Science and UMIACS
University of Maryland
College Park, MD 20742, USA
agrawala@cs.umd.edu

Abstract

Time management is an important aspect of real-time computation. Commercial high performance processors provide little or no support for management of time. In this paper, we propose a time-management unit which can greatly help improve the performance of a real-time system. The proposed unit can be added to any processor architecture without affecting its performance. We also describe how the unit helps to solve the clock synchronization problems in a distributed real-time network.

Keywords: Real-time systems, Microarchitecture, Clock synchronization, Time-based scheduling.

1 Introduction

Accurate time management functions are required for scheduling real-time tasks to meet their deadlines. Time-based scheduling techniques [2] make use of the worst-case execution time estimates of the tasks to generate deterministic schedules for hard real-time systems. With the advent of gigahertz processors, considerable amount of computations can be done in a very short period of time, opening up new opportunities for realizing time-based systems with nanosecond accuracies. This, in turn, demands accurate timing mechanisms for scheduling real-time tasks. A fast and accurate time keeping mechanism, such as a system clock, with fine granularity is therefore essential to implement such precise time-based scheduling schemes. The fast internal clock of modern processors is a potential source to realize a system clock with such fine granularity. Unfortunately, modern high performance processors hardly provide any support to make use of the on-chip clocks to implement time-based systems. Some of the embedded microprocessors such as Intel 80x196, Intel386 EX, and microprocessors such as Pentium Pro [3] provide on-chip timers driven by the processor clock to implement system clocks with better time granularity. Systems that do not use such processors usually use an external hardware timer [4, 5] interfaced to the processor bus. However, these timers have been known to "lose time" during operation [6]. For example, in order to set a new value for the timer, usually a register has to be updated. Unfortunately certain events such as cache/TLB misses, DMA and high priority interrupts can preempt the execution of time management functions. This can cause a delay in updating the timer register and make the system clock lose time.

By moving all the time management functions into the processor microarchitecture, one can alleviate the above mentioned problems and provide a more flexible solution for the management of time. We believe that the time management functions should be made an essential feature of the processors used in real-time systems. In this paper, we propose a hardware architecture of an accurate time management unit for such processors. The following are the basic functionality required for such an accurate time management unit.

- A mechanism to implement a monotonic clock. Monotonicity of the clock is very important because distributed applications assume that time-stamps produced by a clock always increase monotonically [7].
- An automatic mechanism to register the time of occurrence of user-defined events. This property is essential to accurately time-stamp events in a real-time system.

*An earlier version of this paper appear as a UMIACS technical report [1].

- A deterministic and atomic mechanism to read and update the system time.
- A hardware register to hold system time to the required resolution and a mechanism to increment the system time without any software intervention.
- A mechanism to compensate for clock drifts (internal and external) to maintain a consistent global time.

A fast and accurate time keeping mechanism can also be of help in implementing robust real-time distributed computing systems. Solutions to several important problems in distributed computing can be simplified if a global time base is available in the system [8, 9]. Maintaining a consistent global notion of time in a distributed computing environment involves synchronization of all the local clocks. Clock synchronization problem has been studied extensively in the past and several solutions have been proposed [10, 11, 12, 13, 14, 15]. However, most of these solutions depend either on special purpose hardware or complicated protocols. This will add extra processing overhead and complexity to the system, thereby increasing the clock skews in a distributed system. The proposed time management unit can be used to achieve very accurate synchronization of local clocks with little overhead.

Distributed real-time systems such as process control and data acquisition systems often demand an additional capability to accurately time stamp sensor data and events. For example, in a distributed process control system, in order to deal with "alarm conditions", we often need to know the exact sequence of events that caused the alarm condition and the time of occurrence of the events. A time-stamping mechanism with fine granularity is therefore required to resolve the precedence of such close events. The proposed time management unit provides an automatic time-stamping mechanism for identifying and time-stamping events with high resolution.

The rest of this paper is organized as follows. In section 2, we explain the drawbacks of traditional time management techniques used in real-time systems. The architecture of the proposed time management unit is described in section 3. We explain how our solution helps to solve the clock synchronization problem in section 4. In section 5, we review the related work done by other researchers.

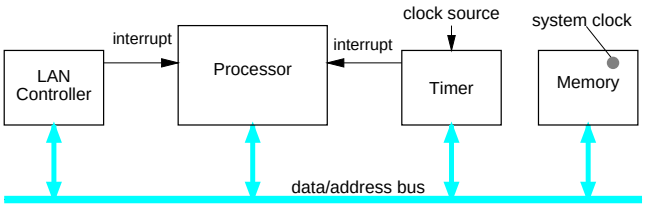

Figure 1: A node of a typical distributed real-time system

2 Time management in Real-Time Systems

The heart of a real-time system is the scheduler that makes important decisions such as, selecting a ready task for scheduling, and making decisions on when and how long each one of the tasks in the system are to be executed. Most of the real-time schedulers make use of a time-out mechanism to preempt the currently executing task and schedule the next task in the ready list, based on the scheduling algorithm used. Clearly, this timing mechanism should be very accurate, otherwise the tasks could be scheduled to run early or late and/or the task could be executed for longer or shorter amount of time than required, resulting in lower processor utilization and missed deadlines. In other words, the correctness of the scheduling scheme directly depends on the timing mechanism used to implement it. Often the same timing mechanism is used to maintain the local (time-of-day or -year) clock of the system.

A typical node of a distributed real-time system makes use of an external timer chip as shown in Figure 1 for time management. The most common timing technique is based on a system clock (software counter) in the memory updated by the *timer-tick* interrupts generated by a fixed interval timer [16, 17]. The main disadvantage of this method is the coarse granularity of the system clock, the software timing mechanism, which is typically of the order of milliseconds. This limitation arises because of the overheads associated with the *timer-tick* interrupt processing, which involves saving and restoring the processor registers, updating the system clock and checking the ready task list. Another disadvantage with this approach is that there is a possibility of missing the timer interrupts

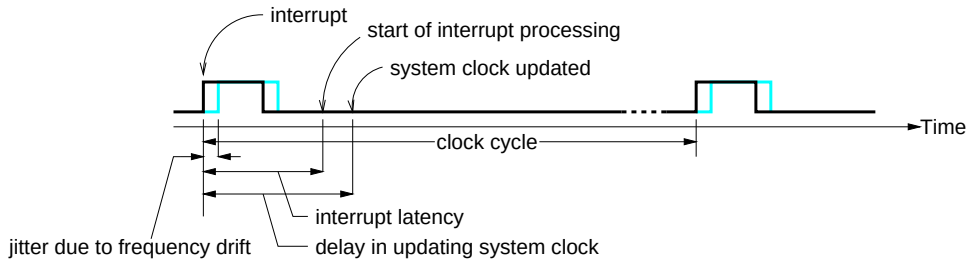

Figure 2: Timing diagram of external timer-based time management

if the interrupts were disabled or the interrupt processing gets delayed due to operations such as DMA and cache misses.

The main factors that constitute the delay in updating the system clock are the following:

1. jitter in the timer interrupt signal due to frequency drifts,
2. the interrupt latency of the processor, and
3. the execution time of the interrupt service routine.

Figure 2 shows the timer interrupt signal and these delays in a typical external timer-based system. Moreover, in modern high performance processor architectures, the interrupt latency and the execution time of the interrupt service routine itself are often hard to predict[18, 19].

An alternative approach is to use a programmable interval timer, which is commonly used for scheduling tasks in time-based real-time operating systems [2]. In this approach, an interval timer is loaded at each task scheduling instance with a new interval equal to the duration of task's execution time-slot. The timer generates an interrupt at the end of the interval to invoke the scheduler again to schedule the next task from the ready list. The main advantage of this method over the *timer-tick* approach is the better time granularity of the scheduling clock, because the granularity depends only on the frequency of the clock signal used to drive the timer and width of the timer register. However, this scheme also has a similar disadvantage of losing the time between the initiation of the timer interrupt service routine and reloading of timer register with new interval, due to DMA operations, cache misses or higher priority interrupt processing. A possible solution to this problem is to modify the timer to allow the new interval value to be added to the current count-down register contents [6]. Another solution is to use a second register to automatically load the next interval value to the timer register, as in the VAX-11 computer systems [20]. In addition to the errors that can occur in the time keeping mechanisms mentioned above, the basic source used to drive the timer itself can generate errors due to drift. Even though the add-timer and second register method apparently solves the problem of updating the system clock for scheduling purposes, these solutions do not address issues such as providing the current time to the applications at any instant for time-stamping purposes and automatic compensation for clock drifts.

It is clear from the above discussion that the existing time management approaches are not capable of providing the fine time granularity, accuracy and flexibility demanded by today's real-time computing systems. The problem with the existing solutions is that they address and solve the problems separately, resulting in solutions that often fail to provide the timing guarantees.

3 Time Management Unit Architecture

The proposed time-management unit works in parallel with the CPU without using any of the computational resources of the processor. The architecture of the time management unit is shown in Figure 3. It consists of a set of registers accessible to the CPU, a *Limit register* and associated logic. A drift-free clock signal is derived from the clock source using a *Rate Adjustment Unit*. The clock source can be the internal clock used in the processor itself or a stable external clock source such as a crystal oscillator. The Rate Adjustment Unit consists of a frequency divider (counter) for scaling down the input clock source frequency and a phase adjustment counter to apply corrections for small changes in frequency. The Rate Adjustment Unit makes necessary corrections to nullify any changes in the frequency of the clock source due to drift

(see section 4).

The system time is maintained in the *Physical Time register*, which is a 64-bit counter incremented at a rate specified by the output clock frequency of the Rate Adjustment Unit. The granularity of time maintained by the system is therefore defined by the output clock frequency of the Rate Adjustment Unit. For example, a 64-bit register can represent a time span of millions of years and provide a time granularity of the order of $1/10^{th}$ of a micro second with a 10 MHz clock derived using the Rate Adjustment Unit. The *Physical Time* register can be accessed by system software as a CPU register to read or modify its contents. The system time T_S is compared with a Limit register T_L at each processor clock cycle and an interrupt is generated when the condition $T_S \geq T_L$ is satisfied. This interrupt signal can be used for real-time task scheduling and precisely initiating time-based events. The interrupt signal will be reset only when the Limit register is modified.

The user accessible registers of the proposed time management unit is shown in Figure 4. All the registers except the *Rate Divisor* register and *Mode Selector* register are 64-bit registers. There are three modes of operation for the proposed time management unit, based on the way in which the *Limit* register content is updated. The modes can be selected by writing appropriate control words in the *Mode Selector* register. The modes of operation are described below:

Absolute time mode: This mode may be used for scheduling tasks precisely at a given absolute time. The time at which the interrupt is to be generated is specified in the *Absolute time* register, which is accessible to the CPU. In this mode, the Limit register is loaded with the contents of Absolute time register and at each processor clock cycle it is compared with the current physical time.

ΔT mode: This mode is intended for generating precise delays by emulating a one-shot timer. The desired delay time is specified in the ΔT register. In this mode, the Limit register will be loaded with the sum of the current contents of the Physical Time register and ΔT register, in the next clock cycle after the the ΔT register is updated. The idea is to prevent any loss of time as in timer tick-based approach.

Auto-reload mode: In this mode, after generating the interrupt, the Limit register is automatically loaded with a new value similar to the ΔT mode. The new value of the Limit register is generated by adding the current contents of Limit register and the ΔT register. For example, if T_S is the physical time (the contents of the Limit register) at the instant when the interrupt is generated, and ΔT is the delay time, then the comparator will generate the next interrupt after $T_S + \Delta T$ seconds. The main difference between this mode and the ΔT mode is that in this mode the changes in ΔT register will be effective in the cycle after the interrupt. The idea is to eliminate the delays (refer to Figure 2) that would have occurred in the ΔT mode if the same functionality is implemented making use of the timer interrupt service routine to load a new time interval after each interrupt. This mode can be used to implement very accurate time-based schedulers such as the one used in the Maruti hard real-time operating system [2].

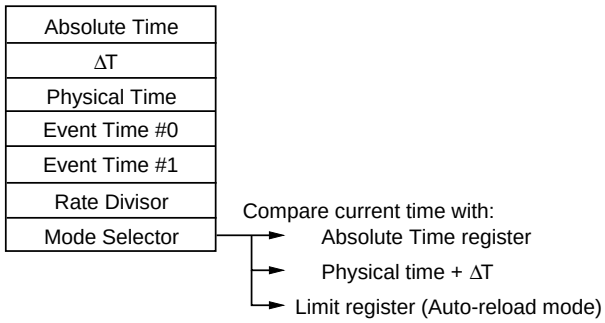

Figure 4: Time Management Unit Registers

The proposed time-management unit also provides an accurate mechanism to time-stamp events – the current physical time can be read into one of the *Event Time* registers using a single register transfer instruction in one cycle. The same operation can also be initiated by an external interrupt signal. This facilitates accurate time-stamping of external events without interfering with the CPU computations. In the next section, we explain how to make use of this feature to implement accurate

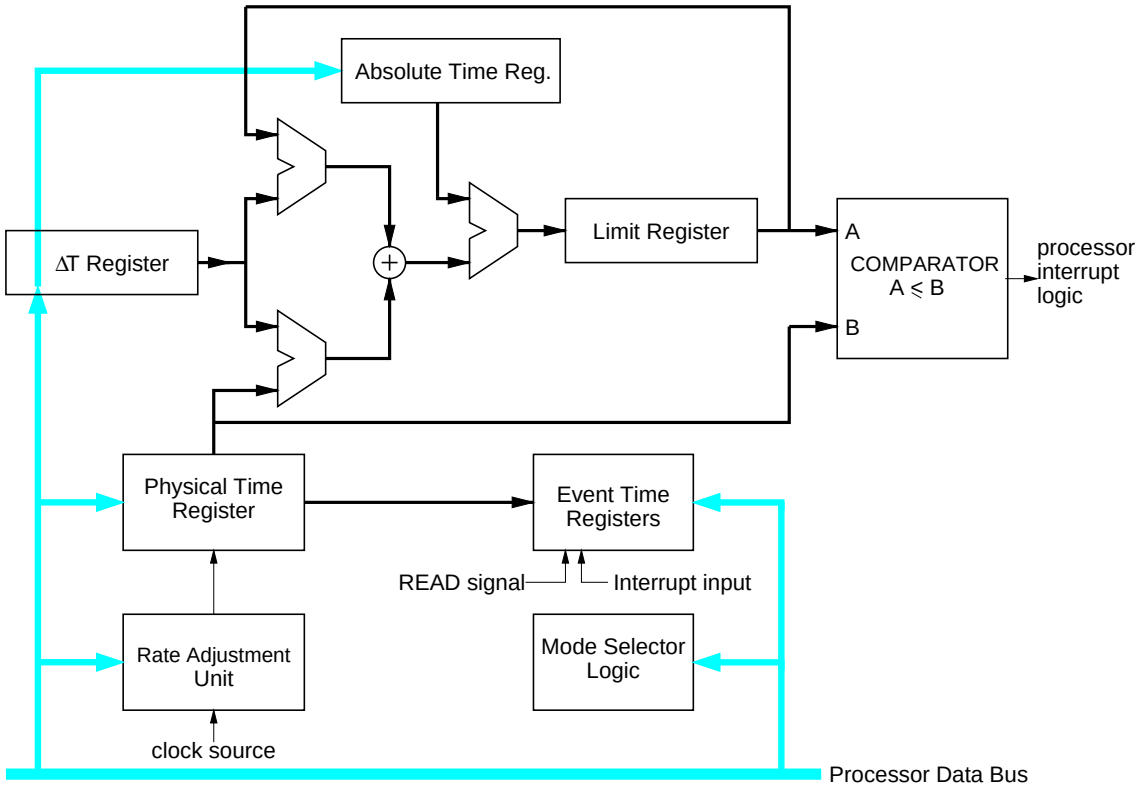

Figure 3: Time Management Unit Architecture

clock synchronization algorithms.

4 Support for Clock Synchronization

The essential requirements of a clock synchronization algorithm for distributed real-time systems can be found in [10, 6]. A common characteristic of all clock synchronization algorithms is that each node periodically computes the deviation of its local clock from a global time base [11]. These clock synchronization algorithms make use of the knowledge about the local clocks of other nodes or a master node in the system to compute the corrections to the local clock. Time stamped packets are used for exchanging the current time of the local clocks in the system. But, in a distributed system, there is a large variability in the amount of time taken between the instant at which a message is submitted for transmission at the sender node to the time at which it is processed at the receiver node. This jitter associated with the message passing may be attributed to the dynamics of the network and the processing delays at the sender and destination nodes. For example, the Ethernet protocol can introduce certain amount of uncertainty which increases with the network traffic [21]. However, by choosing protocols such as TDMA to pre-allocate slots for time message packets, the variability due to the network dynamics can be bounded [22, 2]. However, the jitter persists due to the unpredictable processing delays at the nodes resulting from operations such as non-preemptable interrupt processing, DMA and cache misses. It is clear from the above discussion that regardless of the algorithm used for clock synchronization, the accuracy of the technique is affected by the time stamping operation at the sender and receiver nodes. Clearly, a mechanism is needed to accurately time-stamp the packets just before they are transmitted at the sender and as soon as they arrive at the receiver. The proposed time management unit provides such a mechanism to solve this problem by automatically time-stamping the packets. The packets are time stamped on arrival making use of the interrupt signal from the network interface card without any processor intervention. The current physical time will be latched in the Event Time register by the interrupt signal, which

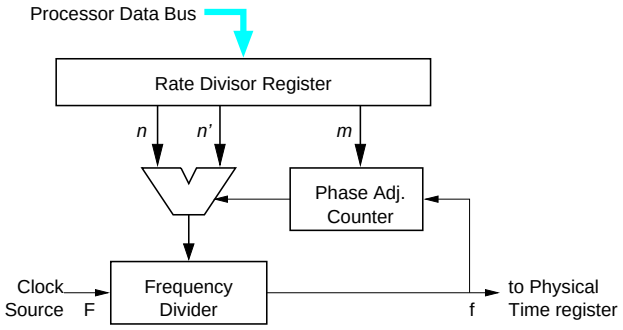

Figure 5: Rate Adjustment Unit

may be used to accurately time-stamp the arrival-time of the packets. Similarly, the Physical time register may be read for time stamping the packet just before they are sent. Thus, with the help of the proposed time management unit, without using any external hardware, the packets can be accurately time-stamped to implement clock synchronization algorithms. Moreover, the Absolute Time mode of the proposed time management unit can be used for precisely scheduling the send time of the time messages.

In addition to the errors due to the jitter in the message passing, the basic clock source itself can generate errors due to drift. This drift in the frequency of the clock source, due to temperature variations and aging, must be corrected. A discrete correction applied to the Physical Time register may cause the local clock to instantaneously leap forward or be set back and then run at the previous rate, thereby violating the monotonicity property. This problem can be avoided by *amortizing* (i.e., spreading out) the correction continuously over a time interval and this clock adjustment technique is called *amortization* [11]. The *Rate Adjustment Unit* implements this technique in hardware to avoid any abrupt "jumps" in the local clock.

A frequency divider making use of a simple binary counter may not be sufficient to derive the desired output frequency accurately from the clock source. This is because of the truncation errors in the approximation of the scaling factor to an integer value. However, it is possible to derive a fairly accurate output clock signal by changing the width of a few clock pulses out of a fixed number of clock pulses periodically, so that the average frequency of the output signal is very close to the required value. The Rate Adjustment Unit makes use of this technique to derive the desired clock frequency. The unit consists of a frequency divider (counter) for scaling down the input clock source frequency and a phase adjustment counter[1] to apply phase corrections as shown in Figure 5. Very small changes in output frequency are taken care of by re-loading the frequency divider with a slightly higher or lower count than the normal count at a rate specified by the phase adjustment counter. If F is the frequency of the clock source and f is the desired output clock frequency, then the normal scaling factor of the frequency divider is given by $n = \lceil F/f \rceil$. If F is not an integer multiple of f, then the frequency divider is loaded with a modified scaling factor $n' = n \pm k$ at every m^{th} output clock cycle, where k is an integer constant, and m is the phase adjustment rate. m may be computed using the following relationship.

$$(1 - m)n + mn' = \frac{F}{f}$$

Most of the time, at the time of re-synchronization, only the phase adjustment count n' and the phase adjustment rate m, needs to be updated. The software can access the *Rate Divisor* register to update the parameters n, n' and m.

5 Related Work

The importance of time management in real-time systems has been identified by researchers for quite some time. The problems with the interval based timing mechanisms and the lack of coordination between the hardware timers and software were discussed by Volz and Mudge in [6]. They suggested the use of absolute time as a solution and proposed an instruction level timing mechanism to maintain absolute time. They have also mentioned the idea of placing the timing functions on the CPU chip for scheduling applications. In contrast, the architecture we have proposed is more versatile and generic in nature – the support for instruction-level scheduling is one of the several features of the proposed architecture. Even though our architecture supports the absolute time representation mentioned in [6, 23], it is not restricted

[1]The counter is called the *phase* adjustment counter because the counter changes the width or the phase of the output signal by a small amount.

to any specific format in which the time is represented.

The Mars project [22, 10] makes use of proprietary network interface logic based on a clock synchronization unit chip to automatically generate time-stamps. Schossmaier and Loy proposed a similar ASIC for supporting external clock synchronization [24]. Both the schemes provide most of the functionality required by a time management unit. However, both of them suffer from the drawbacks of external timer-based approach and also are not portable because they are designed for a specific system. In contrast, our solution is transparent to the external hardware and the network interface logic used in the system, thereby making it easily portable to different platforms. The basic idea of the rate adjustment scheme for deriving clock signal proposed in this paper is similar to the adjustable rate clock scheme proposed by Volz *et al.* for clock synchronization in IEEE 896 Futurebus+ systems [23]. The differences are mainly of hardware implementation details.

Baek *et al.* proposed a hardware-based clock synchronization technique for synchronizing the clock signals [15]. Their technique implements a modified version of the interactive convergence algorithm CNV [13] and they assume that the actual clock signals are available for skew measurements. However, their scheme is not suitable for large distributed systems because of the problems (such as clock skews and signal-to-noise ratio) associated with distributing the high frequency clock signal over large distances. Recently, Fetzer and Cristian proposed an external 64-bit counter based scheme which make use of GPS signals to synchronize clocks [25].

The hardware-assisted software clock synchronization scheme proposed by Ramanathan *et al.*[14] makes use of an algorithm similar to CNV for a distributed system with point-to-point interconnection topology. They emphasize the algorithmic aspects than the implementation issues. The resolution of their technique for applying corrections to the logical clock at nodes is limited by the frequency of the clock source. Also, the scaling factor of their scheme can assume only a fixed integer value. As a result, their scheme cannot compensate for very small variations in frequency. Our architecture, in contrast, is aimed at providing mechanisms for efficient implementation of distributed clock synchronization algorithms with minimum software overheads and better accuracy, rather than implementing a specific algorithm in hardware.

6 Conclusion

In this paper we have proposed an accurate time-management unit architecture for real-time processors. We have shown that the proposed time management unit can be incorporated into any processor microarchitecture with little extra logic. The basic idea behind the proposed time management unit is to exploit the parallelism between the time management functions and the normal CPU operations of the processor, at the same time providing an instruction-level interface to a system clock of fine granularity. We believe that moving the time management functionality into the processor will greatly help to generate better solutions in terms of performance, simplicity and maintainability.

Future Work: In order to exploit the potential benefits of the proposed on-chip time-management unit, the microarchitecture of processor must provide a *deterministic* instruction-level mechanism to interact with the time-management unit hardware. Unfortunately, it is hard to predict the delay between instruction issue and retirement in modern out-of-order issue superscalar processors. Therefore, on such processors a special instruction, such as RDTSC of Pentium Pro [3], for reading the current physical time into an event time register is not sufficient, unless there is a mechanism to ensure that such special instructions can be executed within a deterministic time interval [19]. We plan to address this problem at the microarchitecture level in our future research.

Acknowledgment

We would like to thank a referee for the detailed feedback/suggestions, and Simon Hawkin and Bao Trinh for their comments.

References

[1] K. K. Kailas and A. K. Agrawala, "An Accurate Time-management Unit for Real-time Processors," Tech. Rep. UMIACS-TR-97-28, University of Maryland Institute for Advanced Computer Studies, March 1997.

[2] M. Saksena, J. D. Silva, and A. K. Agrawala, "Design and Implementation of Maruti-II," in *Principles of Real-Time Systems* (S. H. Son, ed.), Englewood Cliffs, N.J.: Prentice Hall, 1995.

[3] *Pentium Pro Family Developer's Manual*, vol. 1-3. Mt. Prospect, IL: Intel Corp., 1996.

[4] *Microprocessor and Peripheral Handbook*, vol. II, ch. 6. Santa Clara, CA: Intel Corp., 1989.

[5] *Product Data book*, ch. 6. Dallas, TX: Dallas Semiconductor Corporation, 1992-93.

[6] R. A. Volz and T. N. Mudge, "Instruction Level Timing Mechanism for Accurate Real-Time Task Scheduling," *ACM Transactions on Computers*, vol. C-36, pp. 988–993, August 1987.

[7] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed system," *Communications of ACM*, vol. 21, pp. 558–565, July 1978.

[8] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 6, pp. 254–280, April 1984.

[9] N. A. Lynch, *Distributed algorithms.* San Francisco, Calif.: Morgan Kaufmann Publishers, 1996.

[10] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Transactions on Computers*, vol. C-36, pp. 933–940, August 1987.

[11] F. Schmuck and F. Cristian, "Continuous clock amortization need not affect the precision of a clock synchronization algorithm," Tech. Rep. RJ 7290 (68547), IBM Almaden Research Center, San Jose, CA, 1990.

[12] Cristian, F., "Probabilistic Approach to Distributed Clock Synchronization," in *9th International Conference on Distributed Computing Systems*, pp. 288–296, IEEE, 1989.

[13] L. Lamport and P. Meilliar-Smith, "Synchronizing Clocks in the presence of Faults," *Journal of the ACM*, vol. 32, pp. 52–78, January 1985.

[14] P. Ramanathan, D. D. Kandalur, and K. Shin, "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems," *IEEE Transactions on Computers*, vol. 39, pp. 514–524, April 1990.

[15] Y. Baek, H.-K. Lee, and H. Yoon, "New hardware-based clock synchronization for the Byzatine fault," *Electronics Letters*, vol. 28, pp. 2018–2019, October 1992.

[16] A. S. Tanenbaum, *Modern Operating Systems*, ch. 5. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1992.

[17] D. Chakravarty, *POWER RISC System/6000: Concepts, facilities, and architecture*, ch. 14. New York: McGraw-Hill, Inc., 1994.

[18] P. Koopman, "Perils of the PC Cache," *Embedded Systems Programming*, vol. 6, pp. 26–34, May 1993.

[19] K. K. Kailas, B. Trinh, and A. K. Agrawala, "Temporal accuracy and modern high performance processors: A case study using Pentium Pro," Tech. Rep. UMIACS-TR-97-60, University of Maryland Institute for Advanced Computer Studies, August 1997.

[20] *VAX hardware handbook*, ch. 20. Maynard, Mass.: Digital Equipment Corporation, 1982.

[21] J. F. Kurose, M. Schwartz, and Y. Yemini, "Multiple-Access Protocols and Time-Constrainted Communication," *ACM Computing Surveys*, vol. 16, pp. 43–70, March 1984.

[22] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: the Mars approach," *IEEE Micro*, vol. 9, pp. 25–40, Feb. 1989.

[23] R. A. Volz, L. Sha, and D. Wilcox, "Maintaining Global Time in Futurebus+," *The Journal of Real-Time Systems*, vol. 3, pp. 5–17, 1991.

[24] K. Schossmaier and D. Loy, "An ASIC Supporting External Clock Synchronization for Distributed Real-Time Systems," in *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pp. 277–282, Jun 1996.

[25] C. Fetzer and F. Cristian, "Building fault-tolerant hardware clocks from COTS components," in *Proc. of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications*, pp. 59–78, Jan 1999.