Instruction Level Mechanisms for Accurate Real-time Task Scheduling *

Richard A. Vols
Trevor N. Mudge
The Robotics Research Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109

Abstract: The scheduling of timed tasks is generally based, at the hardware level, upon the use of time intervals. For example, most microprocessor families provide their only hardware support for timing control in the form of a programmable interval timer chip accessible as an I/O device over the system bus. In this paper we will argue that a more natural and elegant solution bases timing on a local (to a particular cpu) absolute timer. Further, we will show that the desired timing functions can be provided by simple extensions to existing cpu architectures. The widespread use of the "time interval" view has also influenced, in a negative way, the design of programming languages. An important example is Ada, a language designed with real-time multi-tasking explicitly in mind. We will describe the difficulty with the current timing methods used in Ada, and present a method for overcoming the timing weakness by using the proposed timing mechanisms, while still remaining within the definition of the Ada language.

1 Introduction

Real-time, multi-tasking processing requires that the activities of the processors be scheduled in accordance with both timing and external event requirements. Programming tools used for writing such applications should contain effective mechanisms for managing system resources to meet these kinds of requirements. Current time management tools have evolved from a separate development of hardware timers which may be added to the bus of a computer system and scheduling algorithms which use the timers. The consequence of this uncoordinated approach has been the development of timing systems which are largely interval based, inefficient, limited in resolution practically obtainable, difficult to use and not readily extendible to dealing with distributed systems. In this paper we argue that a unified approach to the development of software scheduling mechanisms and supporting hardware yields much more efficient, natural and easy to use tools for timing control. In particular, we suggest that timing control should be expressed in terms of absolute time at the language level and show that there exist simple extensions to cpu architectures which make the implementation particularly straightforward and efficient.

The real-time performance of a system is highly dependent upon the performance of the scheduler, which in turn, is highly dependent upon the timing mechanisms available. Accordingly, real-time computer systems nearly always contain an interval timer, with either fixed or programmable interval, and frequently, but not always, a time of day clock. If present, the time of day clock is usually of relatively low resolution, and not of value in scheduling tasks with a high repetition rate. Scheduling at the user level is typically accomplished by using some type of delay or wait function which puts the user process to sleep until a specified period of time has elapsed [1,2]. The use of a fixed interval clock limits the practically achievable timer resolution because of the software overhead associated with each timer interrupt. This is a limiting factor for many real-time applications. A programmable interval clock can be used to reduce unessential software overhead, but care must be taken in managing it to avoid loss of clock ticks [3].

Mechanisms for timing control in modern languages intended for concurrent and/or real-time applications are either absent or also rooted in the use of time intervals. Ada provides a "**delay** *interval*" construct to allow a process to delay its execution by the value of the variable *interval* [4, section 9.6]. Concurrent C and Concurrent Pascal utilize similar mechanisms [5,6]. Modula-2 [7] provides no intrinsic timing mechanism at all. OCCAM [8] appears to have a construct which references a desired absolute time. It maintains an internal variable which represents time and can apparently be used to delay until after a desired time. However, the syntax for using this feature is not very stable; at least two different versions have appeared in OCCAM documentation over the past three years. Moreover, OCCAM does not maintain a

*This work was sponsored by General Dynamics, contract no. DEY-601540.

To appear in the Proceedings of the IEEE Real-Time Systems Symposium (RTSS 1986)

global sense of time, and has only a very limited time resolution (16 bits), certainly far too limited for serious real-time applications.

Aside from being nonoptimal for real-time scheduling activities, interval based timing causes additional complications for distributed systems which must maintain time synchronism [9]. The adoption of an absolute sense of time at each node in the system simplifies user level synchronization problems by placing the burden at the point of maintaining synchronism among the individual system clocks.

The next section describes the difficulties with current timing mechanisms. Section 3 then presents a set of underlying support primitives based upon absolute timing which allow efficient implementation and illustrates their use in a simple scheduling algorithm. Section 4 describes the use of the new constructs to simplify and improve the efficiency of timing control in Ada.

2 Current Modes of Operation and Their Limitations

The problem of concern is the time scheduling of a set of tasks T1...,TN, so that they are made ready at the times $t_1,\ldots,t_n$. The times $t_1,\ldots,t_n$ are presumed to be known. Also of interest is the ability to schedule a task to occur after some interval of time has elapsed. Algorithms for selecting the times $t_1,\ldots,t_n$ or the interval are covered elsewhere [10] and are not of concern here. Rather, the interactions between the timer and the scheduler are considered. In most existing implementations, the timing hardware is designed without regard for the types of scheduling algorithms best suited for real-time operation and the schedulers must be written to accommodate existing hardware, with a resulting limitation in performance. The most common present mode of operation is to use a fixed interval timer which periodically interrupts the processor and invokes the scheduler. The scheduler maintains a list of scheduled tasks, the times at which they are to be made ready, and a software clock. Each time an interrupt occurs the software clock is updated and the list of scheduled tasks are checked to see if any of them should be made ready. The overhead associated with updating the clock and checking the task list after each interrupt places a lower bound on clock interval which may be used, as this must be incurred on each interval, regardless of whether or not there is a task to be scheduled.

The use of a programmable count down timer makes an alternative scheduling discipline possible. Such a timer always counts down at some basic rate (10–100 kHz are typical). Whenever zero is reached, an interrupt to the processor is generated and the scheduler invoked. Thus, overhead is incurred only when a scheduling operation is actually required, and, therefore, it is not necessary to choose a minimum interrupt rate on the basis of the fraction of processor time taken up by time management. This type of scheduling is preferred for real-time operations. In this case, however, the count in the timer must be updated by the scheduler to the interval required before the next task is to be made ready, and, as described below, this scheme is prone to errors which can result in a "drift" of the time kept by the system[1].

Many vendors offer programmable timers that operate as described above which may be added as a device on the bus of a computer configuration and used for scheduling operations (e.g., the Intel 8254 programmable interval timer [11]). A few processors, e.g., the Intel 8096, even offer on-chip timers. All of these, however, suffer one major deficiency; they allow a cumulative loss of time under some circumstances. If the timer receives a new value by a store operation, the time between the occurrence of the previous interrupt and the store operation is lost. This is of no consequence if no pulse from the underlying clock generator arrived at the timer during the store. However, if such a pulse did arrive during or before the store, its effect will be wiped out by the store of a new value in the timer, resulting in a slow drift of the time maintained on the system. The likelihood of such loss of time is accentuated by the presence of other external devices which might interrupt the processor or cycle steal from the processor during time management.

One solution to this problem is to *add* the new interval to the timer rather than store to it. That is, if *clk* is the value of the interval timer, and *new_interval* is the next delay interval, it is necessary to achieve

$$clk \leftarrow clk + new\_interval;$$

rather than $clk \leftarrow new\_interval$ if a drift in time is to be avoided. However, the add operation must also be performed without losing account of any clock pulses that may occur during the add. If it were implemented in software, there would be the possibility of the add blocking the pulses, particularly if interrupts or DMA occurred simultaneously. Unfortunately, existing commercial timers do not support the desired "add to timer" function in hardware, and one can only approximate the desired behavior by making the timer management function run at the highest priority possible. Then, however, clock pulses and nonmaskable interrupts can still intercede and lead to cumulative time loss. A programmable count down timer with an "add to timer"

[1] Throughout this paper we shall use the word "drift" to mean the cumulative loss of time due to missed clock pulses, rather than the fluctuations in the behavior of the oscillator producing the clock pulses that can often be caused by changes in the physical environment.

function has been built in a real-time computing systems laboratory at the University of Michigan and incorporated into the scheduler in a real-time operating system [3]. In that system, the timer was a device on the Q-bus of an LSI-11/1 system. Cyclic timing intervals as low as 1 millisecond have been realized with the system.

While the timing problems can be solved via an "add to timer" function or the use of a second register in conjunction with a programmable timer, as Digital Equipment Corporation does in the VAX [2], we believe the best solution to the problem of scheduling events in time is based upon a common sense of "absolute time." This is the approach that we shall present in this paper. Although it is not essential to the approach, the paper describes the placing of the necessary timing functions directly on the cpu chip. While perhaps infeasible a few years ago, most current LSI cpu chips have both extra space on the chip and unused instruction codes. These can be used in an upward compatible fashion to provide improved single chip real-time control processors.

3 Basic Timing Functions for Real-time Task Scheduling

3.1 Programmable Absolute Timer

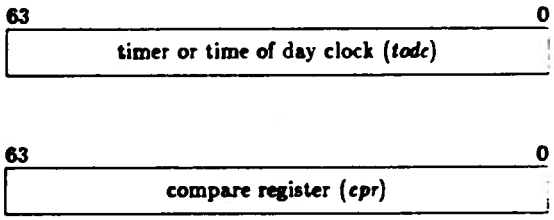
There are three basic questions to be answered in defining the new programmable absolute timer capability:

- How many bits should it have?
- Where should it be placed architecturally?
- How should it function?

The number of bits is a function of timer resolution and the maximum interval length to be measured. A 32 bit timer with a 10 microsecond resolution allows a maximum period of approximately a half a day. This is marginal for many applications. Since the number of bits is not critical to the remainder of the discussion, we assume a timer with 64 bits (this corresponds to a range of about 1/2 million years at a 1 microsecond resolution).

One of the key points to the proposed timer is that it be placed in the cpu. Proper timing operation requires several complex operations to be performed atomically, that is they must be performed in toto or not at all. This can be accomplished if the timer is part of the cpu, but is difficult to guarantee if the timer is only accessed through the bus. The need for atomic operation will become clearer as the necessary timer functions are described and illustrated below.

The proposed scheme requires the addition of two cpu registers, as shown below. The *todc* register holds the current time and is incremented at a fixed rate. The *cpr* register is used to hold the time of the next event to be scheduled. The arrival of the event is checked for by continuously comparing the contents of *cpr* with those



of *todc*. At every clock tick, the following operations are performed:

```
todc ← todc + 1;
if todc ≥ cpr then
      cpr ← 1...1;
      generate timer interrupt;
end if;
```

The comparison is atomic and is performed by hardware: clearly it should be performed within a clock tick. A convenient choice for the clock tick would be a multiple of the basic cpu clock. For example, in current 32 bit microprocessors a 2–4 phase clock would be appropriate and yield a tick period on the order of 1 μsecond. This degree of timer resolution is much finer than is currently typical. Just prior to generating a timer interrupt *cpr* is loaded with its maximum value (all ones). This prevents subsequent meaningless interrupts from occurring before new event times are loaded into *cpr*. The timer interrupt is generated on every tick where $cpr \leq todc$, even if *cpr* is, for whatever reason, loaded with a value less than the present time of day.

Given the above two registers and the continuous checking logic, four scheduling primitives can be defined that are sufficient for implementation of a number of scheduling algorithms:

- Set timer
- Set compare register
- Conditional set compare register
- Read timer

We next describe the operation of these instructions and then illustrate their use in a simple scheduling algorithm.

1. Set timer: *stodc(new_todc,A,B)*

```
c(A,B) ← todc;
todc ← new_todc;
```

where *new_todc* is a 64 bit register pair or memory location, and *A* and *B* are a pair of 32 bit registers (we will assume for the purposes of this discussion that we are dealing with a 32 bit machine). The value returned in the *A,B* register pair can be used for adjusting stored time values after *todc* has been reset to synchronize with an external clock. This operation should be atomic, i.e., no interrupts or DMA activity should intercede during the operation. Further, the set timer should be a privileged instruction. In addition to providing a way to initially set the value of the clock, this instruction will be used to synchronize two or more loosely coupled cpu's.

2. Set compare register: *scpr(new_compare_value,A,B)*

```
c(A,B) ← cpr;
cpr ← new_compare_value;
```

where *A* and *B* are two 32 bit registers used to save the old value of *cpr* and *new_compare_value* is a 64 bit quantity from cpu registers or memory. It is not essential that *scpr(A,B)* be atomic for the purposes of this paper, though other considerations are likely to make it desirable. If interrupts, DMA or other events result in *new_compare_value* being less than *todc* when *scpr* is executed, then a timer interrupt will be generated on the next clock tick. However, since the timing is based on "absolute time" this local anomaly will not contribute to a cumulative bias as it can in similar situations when the underlying timing mechanisms are based on interval timing. If initialization is performed correctly the accuracy rests solely with the ability to minimize the fluctuations of the oscillator that increments *todc*. The contents of *todc* are never modified except when it is set initially or reset to effect synchronization with other processors or some absolute time base.

3. Conditional set of compare register: *cscpr(new_compare_value, flag, A, B)*

```
flag ← 0;
if new_compare_value < cpr then
   c(A,B) ← cpr;
   cpr ← new_compare_value;
   flag ← 1;
end if;
```

This operation simply loads the compare register with a new event time if that time is earlier than the one presently in *cpr*. The 1 bit register *flag* is set to 1 if the exchange is made, and the old value is saved in a register pair. Typically, *flag* will be one of the cpu status bits. The instruction *cscpr* should be atomic. If the new value set into *cpr* is less than *todc*, that is, one has attempted to schedule something to occur prior to the present time, an interrupt will occur immediately after completion of this instruction and the item to be scheduled can be handled at the nearest possible point in time to that desired. Again, because the scheme relies only on absolute time, there will be no permanent or cumulative loss of time.

4. Read timer: *rtodc(save_timer_value)*

```
save_timer_value ← todc;
```

This operation saves a copy of the timer in *save_timer_value*, a 64 bit pair of cpu registers or memory locations.

We next demonstrate how the defined operations can be used as primitives in a simple model of a scheduling algorithm.

3.2 Simple Scheduling Model

Consider a set of tasks to be scheduled at various points in time. Order the tasks by the time at which they are to begin. For illustration purposes, we show the result as a linked list of task control blocks (TCB's) in Fig. 1. Each TCB contains, among other things, a pointer, *next*, to the TCB of the next task in the scheduling sequence, and a value, *scheduled_time*, at which the task is to begin. The variable *HEAD* points to the head of the list. Suppose that the timer already has been loaded with *HEAD.scheduled_time*, the time at which task T1 is to begin. *HEAD.scheduled_time*, the time at which task T1 is to begin. If *T* is a pointer to the TCB of a new

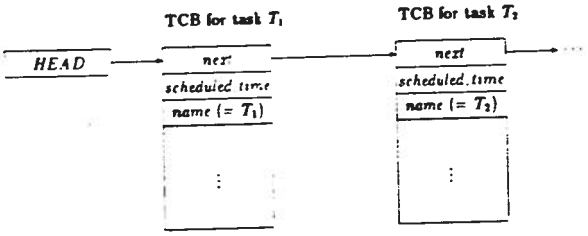


Figure 1: List of TCBs

task, that task can be added to the list of Fig. 1 as follows,

```
T.scheduled_time ← calculate scheduled time;
interrupts off;
   cscpr(T.scheduled_time,flag,A,B);
   insert T in schedule list;
interrupts on;
```

The action of the *cscpr* instruction permits the new task to be placed at the head of the queue of tasks to be processed if its *scheduled_time* is earlier than that of T_1. Of course, in such cases the *scheduled_time* for T_1 is not lost, and

it will be reloaded into *cpr* at the appropriate time, as we will see. The insertion of T into the list can be done in the usual way by following the chain of TCB's until $T.scheduled\_time \leq next.scheduled\_time$.

The actions performed by the scheduler upon the occurrence of a timer interrupt are equally simple. If the variable *NOW* is used to retain the *cpr* time which caused the timer interrupt, then the scheduler need only perform the following:

```
interrupts off ;
NOW ← HEAD.scheduled_time;
repeat
   add task pointed to by HEAD to set of ready tasks;
   HEAD ← HEAD.next;
until NOW ≠ HEAD.scheduled_time ;
scpr(HEAD.scheduled_time,A,B) ;
interrupts on ;
```

The repeat...until is executed at least once and moves all the tasks which where scheduled to run at the *cpr* time which caused the interrupt to the set of ready tasks. These tasks all have the same *scheduled_time*. The first task after the sequence of ready tasks then has its *scheduled_time* set in the compare register to await its start time.

In this simple example, neither the flag nor the registers which return the old value of *cpr* were used because all of the needed information was held in the TCB's. However, in more complex scheduling methods, a need for them may arise.

4 Language Level Issues

The pervasiveness of the "time interval" view has influenced the design of programming languages in a way that has introduced unnatural complexity into the scheduling operations. An important example is Ada, a language designed with real-time multi-tasking explicitly in mind. Ada provides a predefined date type, DURATION; objects of this type represent time intervals. A language construct, delay, provides a delay for at least the length of time given by an argument of type DURATION. In addition, a predefined package, CALENDAR, is specified which provides a data type TIME, and a set of mathematical operations for dealing with TIMEs and DURATIONs. For example, it provides a function "-" which subtracts two TIMEs and yields a DURATION.

To illustrate the influence of interval timing on Ada, we present an Ada version of a commonly used timing loop for repetitive (at a fixed interval) operations and note that even though the language has interval based timing mechanisms the user must still maintain an absolute sense of time (at least locally). We consider a simple control loop, which for instance might be controlling the motion of a robot arm, which must be executed repetitively with period 0.01 seconds. Denote the control action by the subprogram F. Then using the Ada language syntax the loop may be expressed as [12]:

```
with CALENDAR;
declare
   use CALENDAR;
   INTERVAL: constant DURATION := 0 .01;
   NEXT_TIME: TIME := FIRST_TIME;
begin
   loop
      delay NEXT_TIME - CLOCK;
      F;
      NEXT_TIME := NEXT_TIME + INTERVAL;
   end loop;
end;
```

where the package CALENDAR provides the data types DURATION and TIME, and the functions CLOCK (for returning the current value of TIME), "-" and "+" for operating on values of these types.

There are several observations to be made about this example. First, even though the example itself is intrinsically interval based and the language provides interval based timing, it is necessary for the user to implement an absolute (at least local to this problem) sense of time. This is necessary because of the unknown length of time required for the execution of F (indeed, it may not even be constant). Without the maintenance of an absolute sense of time, there could be a long term drift in the timing of the loop which could be harmful to the operation being performed.

The syntax of the language reflects the bias toward an underlying interval timer. The example illustrates the inefficiency of this. One must first convert from a time interval specification to an absolute time specification and then back again. Furthermore, arithmetic involving data objects of type TIME is not necessarily efficient. On one compiler tested, the times required for "+" and "-" are on the order of 200 μ seconds. Even worse, CLOCK function measurements on five Ada compilers showed times ranging from 94 to 3400 μseconds [13]. Furthermore, the underlying scheduling operations must suffer from one of the kinds of ills described in section 1.

The times required for timing operations is highly dependent upon the representation of the data types. The underlying use of an interval timer in conjunction with the requirement to provide functions returning the MONTH, DAY and YEAR of the TIME make a record representation appealing. It is this underlying record

implementation which leads to the large execution times.

The use of an underlying absolute timer would allow and encourage improvements in several ways. First, it would encourage an underlying representation of TIME objects as extended fixed point numbers, with conversion being performed for MONTH, DAY and YEAR as necessary. Then the TIME arithmetic and CLOCK functions could be performed in μseconds rather than hundreds of μseconds. Time could easily be kept in Greenwich Mean Time allowing compatibility across time zones; the time zone would then be simply a set-up parameter. Second, it would allow a more natural language expression of the timing function. Using the package facility a new package could be defined, ABS_TIME that supercedes the capabilities of CALENDAR and that is based on absolute time. Among other things it could provide a new procedure DELAY_UNTIL(T: TIME), which would delay until the timer reaches the time passed in as a parameter. This would typically be used in place of the current delay statement in Ada.

Then the above control loop may be expressed as:

```
with ABS_TIME;
declare
  use ABS_TIME;
  INTERVAL: constant DURATION := 0.01;
  NEXT_TIME: TIME := FIRST_TIME;
begin
  loop
    DELAY_UNTIL(NEXT_TIME);
    F;
    NEXT_TIME := NEXT_TIME + INTERVAL;
  end loop;
end;
```

Several points are worth noting. First, no reading of the clock is necessary. Second, no subtraction of time is necessary. Both of these contribute to being able the execute much faster loops. Also, the expression of the timed loop is more natural and easier to understand.

5 Summary and Conclusions

The management of time is critical in real-time embedded systems. We have shown that basing time on a (local) absolute timer is more natural, leads to simpler implementations, and is easier to use. The timing registers and primitives introduced can be easily implemented on a cpu chip or on a timer board that can be attached to the system bus. We have illustrated both simple time scheduling using the proposed instructions and new high level language functions for Ada to allow more efficient and natural expression of timed loops.

The use of (local) absolute timers also simplifies the maintenance of synchronism across a set of machines and isolates the real problem, that of providing correct synchronized values of time to each of the processors in the system. Though beyond the scope of this paper, there are a number of mechanisms possible for solving this latter problem, e.g., radio and satellite broadcasts of digitally encoded time (and geographical position which allows transmission time compensation).

References

[1] Intel Corporation, *iRMX-86 Reference Manuals*, Santa Clara California, 1980.

[2] Digital Equipment Corp., *VAX 11/780 Hardware Handbook*, Maynard, Mass., 1978.

[3] R.A. Volz, *The CRASH—Compiler for Real-time Applications SHop—Manual*, Electrical and Computer Engineering Dept., The University of Michigan, Ann Arbor, Michigan, 1978.

[4] *Ada Programming Language (ANSI/MIL-STD-1815A)*. Washington, D.C., 20301: Ada Joint Program Office, Department of Defense, OUSD(R&D), January, 1983.

[5] N.H. Gehani and W. D. Roome, *Concurrent C*, AT&T Bell Laboratories report, Murray Hill, New Jersey, 1985.

[6] Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[7] N. Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin, 1982.

[8] D. May, "OCCAM," *SIGPLAN Notices*, vol. 18, no. 4, pp. 69-79, April 1983.

[9] R.A. Volz, T. N. Mudge, A. W. Naylor, and J. H. Mayer, "Some Problems in Distributing Real-time Ada Programs Across Machines," *Ada in Use, Proc. of the 1985 Int'l Ada Conf.*,

[10] K. Ramamritham and J.A. Stankovic, "Dynamic task scheduling in distributed hard real-time systems," *IEEE Trans. Software Engineering*, vol. 1, no. 3, July 1984.

[11] Intel Corporation, *Microsystems Components Handbook*, Santa Clara California, 1984. pp. 72-84, May 1985.

[12] J.G.P. Barnes, *Programming in Ada*, (2nd ed.), Addison-Wesley: London, England, 1984.

[13] R.M. Clapp, L.J. Duchesneau, R.A. Volz, T.N. Mudge, and T. Schultze, "Toward real-time performance benchmarks for Ada," *Communications ACM*, to appear.