David P. Anderson
Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, California 94720 USA

Ron Kuivila
Music Department
Wesleyan University
Middletown, Connecticut 06547 USA

Accurately Timed Generation of Discrete Musical Events

Introduction

Most computer-controlled synthesizers have interfaces that are defined in terms of discrete real-time "events" (e.g., commands sent from the computer to the synthesizer). In designing a programming system for such an environment, some likely goals are:

- Allow a program to interact with a human performer (via input devices, such as keyboards, attached to the computer) and to respond quickly to these interactions.
- Support "on-line" algorithmic music generation, that is, allow the computer to act as a composer, accompanist, or improviser, rather than simply as a sequencer or message forwarder.
- Provide accurately timed event performance, for precise rhythmic control.
- Support concurrent (multiprocess) programming. Concurrency is essential for many musical applications; for example, multiple concurrent note-playing processes are useful for polyphony.

This paper explores the interdependencies among these goals and proposes mechanisms by which the goals can be simultaneously met. These mechanisms are *event buffering*, which reduces or eliminates event timing errors, and *deadline scheduling*, which is an appropriate way to schedule event-generating processes.

Access to these low-level mechanisms can be as procedure or function calls from a language such as C or Lisp. Hence the mechanisms can be provided without writing or even necessarily modifying a compiler. An initial implementation using Forth has been completed and achieves the goals listed above.

Computer Music Journal, Vol. 10, No. 3, Fall 1986,

Timing Accuracy of Event Performance

Terminology

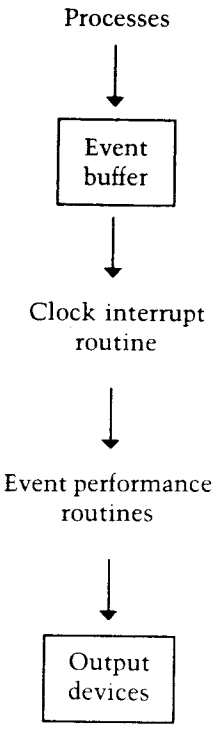
An *event*, as the term is used in this paper, is the execution of an *event performance routine* that performs an output action. An example is a synthesizer "key-down" event, which might involve writing a few bytes out on a bus or MIDI channel.

An event routine can take parameters; for example, the key-down routine might need pitch, waveform, and envelope information. The routines are assumed to use negligible central processor unit (CPU) time; in practice, this means that their CPU time per call should be less than the required temporal resolution. However, the computation of an event (the algorithmic selection of the performance routine and its parameters) may take significant CPU time. The distinction between event *computation* and event *performance* should be kept in mind.

Problems with Simple Scheduling Approaches

Conventional operating systems do not provide a service for accurately timed event performance but may supply a timed "sleep" service. A simple approach to event generation would have a process perform each event immediately after it is computed, then sleep for the duration of the event. Were processing infinitely fast this approach would be correct. As it is, each event is delayed by the time

Fig. 1. Event path, from processes to output devices, e.g., synthesizers.



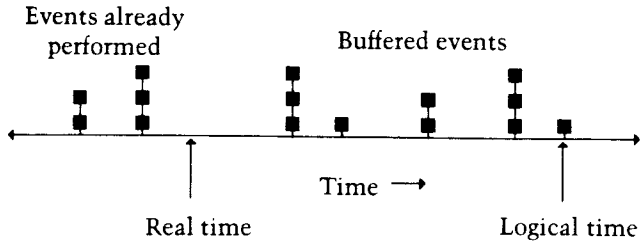
required for its computation. These delays are cumulative and become noticeable when there are groups of simultaneous or nearly simultaneous events, each of which takes significant CPU time to compute.

Intervention scheduling (Abbott 1984) and Moxie (Collinge 1980) starts each event computation at its scheduled performance time. This approach reduces error accumulation but doesn't address the problem of simultaneous events.

Event Buffering

Event buffering is a method that decouples computation from performance. Events are computed slightly ahead of their performance, and *event descriptors* (records containing the address of an event performance routine together with whatever parameters it requires) are stored in an *event buffer* while they await performance. A clock interrupt routine performs the events buffered at their scheduled times (see Fig. 1).

Fig. 2. Logical time and real time, with logical time in advance of real time.



Event buffering can reduce or eliminate timing inaccuracy. As an example, suppose a composition program uses 0.2 CPU sec to compute each note and that most notes are 1 sec apart, but occasionally there is a burst of 10 notes only 0.1 sec apart. If the CPU can buffer 2 sec of notes, it is able to keep ahead of schedule, allowing accurately timed event performance.

For accurately timed on-line performance to be possible, the average CPU time used to compute each event must be less than the average delay between events. Conversely, event buffering is of little value unless the CPU time used in computing events is nonnegligible compared to the desired timing accuracy.

An Event Buffer Abstraction

To clearly separate event buffering from process scheduling, we describe the event buffer as a module that is accessed by only a single process. *Logical time* is the (real) time at which the event currently being computed is scheduled to be performed. Because of event buffering, logical time is usually ahead of real time (see Fig. 2).

The event buffer module provides two services:

TIME_ADVANCE(dt)

advances the logical time by *dt*, and

SCHEDULE_EVENT(procedure_address, parameters)

schedules an event for performance at the current logical time.

The model provided by these two services re-

quires that events be computed in the order of their performance. An extension that tempers this requirement is described in the section "Future Actions."

Processes and Deadline Scheduling

In this section we discuss the idea of *process scheduling* and describe a particular policy called deadline scheduling that is appropriate for scheduling event-generating processes.

Terminology

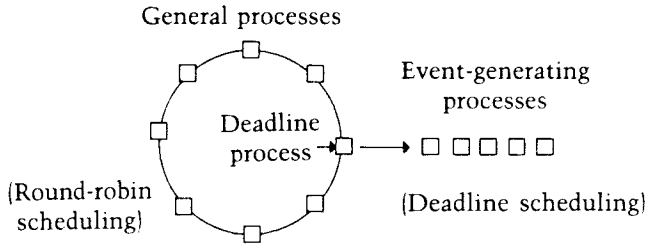
A *process* is the abstraction of a processor executing a single instruction stream. Multiple *concurrent* (simultaneously executing) processes can be simulated on a single processor by occasionally switching between processes (e.g., Andrews and Schneider 1983). Each such *context switch* saves the machine state of the exiting process in memory and loads the machine state of the new process. The area in memory where the state of a process is stored is called its *context block*.

The question of when to do a context switch and what process to switch to must be decided by a *scheduling policy* in the underlying system. A policy in which processes are switched only at known points in their execution, such as during system calls, is called *nonpreemptive*. Nonpreemptive scheduling policies have the advantages that (1) context switching is fast because little state information need be saved, and (2) access to shared data structures is simpler because a critical section mechanism is not needed. Nonpreemption was a goal in designing the scheduling mechanisms discussed here.

Event-generating versus General Processes

It is useful to distinguish two classes of processes: *event-generating processes*, which generate timed output events, and *general processes*, which have tasks such as monitoring input devices. They can

Fig. 3. Process scheduling mechanisms.



explicitly interact; for example, a general process can create an event-generating process. However, the two classes are scheduled by different (but interacting) policies. Event-generating processes are scheduled by a mechanism called deadline scheduling, to be described later.

General processes do not generate events and so are not scheduled by the deadline mechanism. In our implementation, general processes are arranged in a circular queue and are scheduled in a round-robin, nonpreemptive fashion. Three services are available to general processes:

pause() does a context switch to the next process in the queue
sleep() removes the calling process from the queue
wakeup(P) restores a sleeping process to the queue

A general process can be *interrupt-driven* (it sleeps until an interrupt from an input device wakes it up) or *polling* (it runs an infinite loop in which each iteration checks for input, then calls **pause**).

The connection between deadline and round-robin scheduling (or whatever mechanism is used for general processes) is a *deadline process*, a general process that manages deadline scheduling. This relationship is depicted in Fig. 3.

Deadline Scheduling

At any point, each event-generating process is positioned at a particular logical time. A process can advance in time by calling

time_advance(dt)

and it can schedule an event for its current logical time by calling

schedule_event(procedure_address, parameters)

These are generalizations of, and are defined using, the lower-level services **TIME_ADVANCE** and **SCHEDULE_EVENT** mentioned previously.

Given a set of event-generating processes, the *soonest process* is that whose logical time is earliest. Deadline scheduling is the nonpreemptive policy in which the executing event-generating process is always the soonest process (i.e., the process with the earliest deadline).

Deadline scheduling has previously been used for discrete simulation languages, such as Simula, that use the model of concurrent event-generating processes; see (Leeming 1981) for a survey. It is studied analytically in (Liu and Layland 1973). Several computer music language systems have also apparently used some form of deadline scheduling (Loy 1981; Schottstaedt 1983). The same idea has been used for scheduling interrupt handlers with fixed maximum latencies (Clark 1985).

An executing event-generating process eventually calls **time_advance,** say with argument *dt*. Three steps are then performed:

1. Its logical time is advanced by *dt* units. There is now a new soonest process. Let *x* be the difference between the old logical time and the logical time of the new soonest process.
2. **TIME_ADVANCE** is called with argument *x*, since this is net advance in logical time on switching from the old to the new soonest process.
3. Control is transferred to the new soonest process.

This algorithm is specified in more detail later.

Deadline scheduling is the natural choice for scheduling event-generating processes for the following reasons:

Timing errors due to missed deadlines (i.e., an event not computed at its scheduled performance time) are avoided if possible, since the process with the earliest deadline is always given highest priority.

Events in different processes are computed in a deterministic order, namely the order in which they occur in real time. This provides an implicit synchronization that simplifies the programming of interacting processes.

Event Buffering Revisited

We now consider event buffering in more detail.

Buffer Delay and Response Latency

The *buffer delay* at any point is the difference between logical and real time. When a user interacts with an existing process the audible effect of this interaction is delayed by the buffer delay. This is an inevitable side-effect of event buffering; for interaction with a process to have an immediate effect on the stream of event performances, it would be necessary to "back up" the computation to the current point in real time, which is not generally feasible.

An upper bound on response latency can be provided by limiting the buffer delay, i.e., by imposing a limit on the temporal (not the physical) size of the event buffer. A bound of a fraction of a second typically provides adequate responsiveness together with enough buffering for the CPU to stay ahead of schedule. This bound can be enforced by having **TIME_ADVANCE** block (put its caller to sleep) until the buffer delay falls below the limit.

Fairness to General Processes

If the scheduling of general processes is nonpreemptive, it is possible for event-generating processes to monopolize the CPU. This occurs if they don't compute fast enough to fill the event buffer to its limit, in which case **TIME_ADVANCE** never sleeps.

A reasonable solution to this problem is to force **TIME_ADVANCE** to call **pause** (thereby giving all general processes a chance to execute) whenever it has advanced more than some fixed amount of logical time without sleeping.

Implementation of Event Buffering

Conceptually, the event buffer can be viewed as a queue of event descriptors and delay records. It could be implemented as such. In practice, it has been advantageous to implement it as an array in which each entry corresponds to a clock tick, and points to a list of event descriptors to be performed at that tick.

Only the ticks within the current event buffer (i.e., between real time and logical time) are relevant, and a bound can be placed on this interval. Therefore the event buffer array can be a fixed-size circular buffer, together with two points that indicate the current real and logical times.

The following variables are involved in the event buffer implementation:

event_buffer:	an array of pointers to lists of event descriptors.
buffer_size:	the size of *event_buffer*
logical_index:	index into *event_buffer* for logical time
real_index:	index into *event_buffer* for real time
buffer_delay:	logical time minus real time
max_delay:	response latency bound
delay_since_pause:	time advance since **TIME_ADVANCE** last called **pause** or **sleep**
nice_time:	*limit on delay_since_pause*

An algorithm for **TIME_ADVANCE** incorporating the latency-limiting and fairness mechanisms can now be given:

```
procedure TIME_ADVANCE(dt)
  logical_index=(logical_index+dt)
    mod buffer_size
  buffer_delay+=dt
  if buffer_delay>max_delay+1
    sleep()
    delay_since_pause=0
  else
    delay_since_pause+=dt
    if delay_since_pause>nice_time
      pause()
      delay_since_pause=0
    end if
  end if
end
```

Note: The use of t= and −= operators derives from the C programming language. In C, if *e1* and *e2* are two expressions, then *e1* op= *e2* is equivalent to *e1*=*e1* op *e2*.

If an interrupt routine wakes up a general process and wants to ensure that it runs soon, it can set *delay_since_pause* to *nice_time* + 1. The general processes will run as soon as **TIME_ADVANCE** is called. A preemptive scheme could be developed to make response to interrupts quicker; however, this would eliminate the advantages of nonpreemption mentioned earlier. In our experience the benefits of preemption are not worth the added trouble.

Late Events

If the CPU cannot generate events fast enough to keep the buffer nonempty, events are of necessity performed late. Initial lateness can be avoided by giving the CPU a "head start"; that is, allowing the event buffer to become partially or completely full before enabling the clock interrupt.

There are several choices in dealing with lateness. In some applications it might be best to always perform events as close to the original schedule as possible. However, this approach can change the temporal relationships of events in musically undesirable ways, such as by breaking up chords. Musical requirements suggest the following design decisions:

Events scheduled to occur simultaneously are always performed on the same clock interrupt. This is realized by having the clock interrupt routine return without performing any events if *buffer_delay* = 0.

The interval between two event performances is never allowed to be less than the scheduled interval. When an event is performed late, the entire remaining schedule is postponed.

Given these decisions, the algorithm for the clock interrupt routine is as follows:

```
procedure clock_interrupt()
  if buffer_delay==0 return
  perform the event list in
    event_buffer[real_index]
  real_index=(real_index+1)
    mod buffer_size
  buffer_delay-=1
  if buffer_delay==max_delay
   wakeup(deadline process)
  end if
end
```

Deadline Scheduling Revisited

In this section we describe one way of implementing deadline scheduling, and discuss the issues that arise in the creation and termination of event-generating processes.

Delay Queues

Deadline scheduling can be implemented by organizing the context blocks of event-generating processes as a *delay queue*. The soonest process is always at the head of the queue. Each context block has a *delay field* giving its delay beyond the previous process in the queue (i.e., the difference between the logical times of the two processes).

As an example, suppose the queue is initially as follows:

	head			*tail*
process:	A	B	C	D
delay:	4	8	5	10

If process *A* calls **time_advance** with a *duration* argument of 16, the result is:

	head			*tail*
process:	B	C	A	D
delay:	8	5	3	7

Note that the delay field of *B* now contains its delay beyond the original logical time of *A*.

The Deadline Process

Recall that the deadline process is a general process that manages deadline scheduling. If the delay queue structure is used, the algorithm executed by the deadline process is as follows (*PQH* stands for *process queue head*, i.e., the soonest process, and *PQH.delay* is its delay field):

```
procedure deadline_process()
  loop
    if the deadline process queue
      is non-empty
    switch to PQH
      TIME_ADVANCE(PQH.delay)
    else
      pause
    end if
  end loop
end
```

and the algorithm for **time_advance** is:

```
procedure time_advance(dt)
  if dt≠0
    advance this process by dt in
      delay queue
    switch to deadline process
  end if
end
```

The deadline process provides a logical separation between deadline scheduling and the other system components (the event buffer and general process scheduling). There are alternative designs that don't use a separate deadline process. These may be slightly more efficient because they use fewer context switches, but conceptual simplicity is sacrificed.

Creation of Event-generating Processes

The semantics of creating an event-generating process include initializing its logical time. The choice

of "starting time" can depend how the process is created. If an event-generating process *P* creates a process *Q*, then *Q* starts at *P*'s logical time. In other words, *Q* is placed in the process queue at zero delay after *P*. This satisfies the obvious musical requirement that child processes should be initially synchronized with their parent.

If an event-generating process is created by a general process (e.g., by a user-level command interpreter), there are potentially two choices for its initial logical time:

1. The new process starts at the logical time of the soonest process, that is, it is inserted with zero delay after the process queue head.
2. The new process starts with its logical time equal to the current real time. This is done by storing *buffer_delay* in the delay field of the process queue head, placing the new context block with zero delay at the head of the queue, and setting *logical_index* = *real_index* and *buffer_delay* = 0.

The latter alternative is referred to as *fast-starting* the process. It has the property that if the process schedules an event before it calls **time_advance,** that event is heard as soon as possible after the process is created. The event is delayed only by its computation time, rather than by the buffer delay. Clearly this is musically useful, for it means that processes created by input gestures can be heard almost instantaneously and are rhythmically synchronized with existing processes.

Structuring the event buffer as an array instead of a linked list simplifies the implementation of fast-starting. This was the main reason for that choice.

Fast-starting violates the principle that events are computed in order of their occurrence. This can create anomalies while a new process is "catching up" to the existing processes. For example, if the new process reads a frequently modified shared variable, its view of the value is inconsistent with that of existing processes during this period. In practice, this has not been a problem.

Process Termination

A problem arises when the exiting process is the only process in the queue, and it exits after scheduling its final event. The clock interrupt routine will not perform this event until it is assured that there are no more events simultaneous with it, i.e., until another **TIME_ADVANCE** is done. This is fixed by calling **time_advance**(1) in the process exit routine.

Future Actions

The event generation model supplied by **schedule_event** and **time_advance** requires that processes compute events in their temporal order. At another extreme is a model in which events can be computed in any order; that is, **schedule_event** takes an absolute event time as an additional parameter. This "random-access" model is convenient for certain musical applications, such as distributing events randomly over an interval. However, it renders on-line event performance impossible; there is no alternative but to store all events and perform them after the entire computation is finished.

An intermediate model of event generation, based on *future actions*, has the advantages of both the sequential and random-access models. As before, a process has a logical time that it can (and, periodically, must) advance. However, instead of restricting a process to scheduling events only at its current logical time, we allow it to schedule *actions* at non-negative delays from this time. An *action* is a call to an arbitrary parameterized procedure that is stationary in time, i.e., does not call **time_advance.** This procedure may, for example, be **schedule_event.**

When a process advances, actions scheduled in the time interval between its old and new logical times can safely be performed, since no more actions can be scheduled for that interval.

For musical purposes, this model provides significant advantages over the sequential model. A process might position itself at successive measure boundaries and generate the events within each measure randomly. The model also provides a

framework for legato, where key releases are expressed as future actions.

Scheduling a future action is similar to the idea in Moxie (Collinge 1980) of *causing* a procedure call. The difference is that here it is used as an *adjunct* to a process mechanism, whereas Moxie uses it to *provide* a processlike facility (i.e., a procedure can do some work, then recursively cause itself to be called at a future time).

This distinction is worth elaborating. Processes and self-causing procedures can both express musical activities in time. With processes, information such as program counter, loop indices, etc., is part of the process state and is implicitly propagated over time advances. With self-causing procedures, the activity must start executing at the beginning of a procedure whenever it advances in time, and state information must be explicitly propagated by the programmer.

Implementation of Future Actions

Future actions can be implemented as an extension of the sequential model by associating with each process a *future action queue* (FAQ), a delay queue of *action descriptors*. An action's total delay in the FAQ is the interval between the process's logical time and the time the action is to be performed.

schedule_future_action(procedure_address, parameters, delay)

inserts an action descriptor with the given delay in the FAQ of the calling process. **time_advance** must be redefined to take the FAQ into account. The algorithm for the new version of **time_advance** is as follows (it is defined in terms of the old version, referred to below as **old_time_advance**).

```
procedure time_advance(dt)
  time_left=dt
  loop
    if FAQ is empty
      old_time_advance(time_left)
      return
    end if
    if time_left<(FAQ head).delay
      (FAQ head).delay-=time_left
      old_time_advance(time_left)
      return
    end if
    old_time_advance((FAQ head).delay)
    time_left-=(FAQ head).delay
    remove FAQ head and perform its
    action
  end loop
end
```

When a process exits, its FAQ can be nonempty. To ensure that these actions get performed, the process exit routine computes x = the total delay in the FAQ, and calls **time_advance**(x).

Conclusion

We have proposed models for output events (parameterized performance routines) and for event generation by processes (sequential, and with future actions). We have posed the problems of process scheduling and accurately timed on-line event performance for these models, and have given efficient and easily implemented solutions.

The mechanisms have been implemented on several microprocessor configurations. These include a single-processor system and dual-processor systems, both with and without shared memory, in which one processor buffers and performs events while the other executes the event generating processes. These implementations are in Forth, and are the basis for a computer music language called FORMULA (Anderson 1984).

This work addresses issues in the design and implementation of a high-level control language for musical areas such as real-time algorithmic composition and programmable interactive performance. The proposed mechanisms can be added to an existing procedural language. The capabilities provided by these mechanisms are relevant mainly at the level of controlling note initiations. At lower levels such as envelope generation, computational

models such as those of FORMES (Rodet and Cointe 1984) and Arctic (Dannenberg 1985) are more applicable.

References

Abbott, C. 1984. "Intervention Schedules for Real-Time Programming." *IEEE Transactions on Software Engineering* SE-10(1):268–274.

Anderson, D. P. 1984. "A Forth Computer Music Programming Environment Design." *Proceedings of the 1984 Rochester Forth Conference.* Rochester: Institute for Applied Forth Research, pp. 217–227.

Andrews, G. R., and F. B. Schneider. 1983. "Concepts and Notations for Concurrent Programming." *ACM Computing Surveys* 15(1):3–44.

Clark, D. D. 1985. "The Structuring of Systems using Upcalls." *Proceedings of the 10th Symposium on Operating System Principles,* pp. 171–180.

Collinge, D. 1980. "The Moxie User's Guide." Internal report. Victoria: School of Music, University of Victoria.

Dannenberg, R. B. 1984. "Arctic: A Functional Language for Real-Time Control." *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming,* pp. 96–103.

Leeming, A. M. C. 1981. "A Comparison of Some Discrete Event Simulation Languages." *Simuletter* 12:1–4.

Liu, C. L., and J. W. Layland. 1973. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." *Journal of the ACM* 20(1):47–61.

Loy, G. 1981. "Notes on the Implementation of MUSBOX." *Computer Music Journal* 5(1):34–50.

Rodet, X., and P. Cointe. 1984. "FORMES: Composition and Scheduling of Processes." *Computer Music Journal* 8(4):32–48.

Schottstaedt, B. 1983. "PLA Reference Manual." Stanford: Center for Computer Research in Music and Acoustics, Stanford University.