A Real Time Scheduler/Dispatcher*

Roger B. Dannenberg
Computer Science Department and
Center for Art and Technology
Carnegie Mellon University
Pittsburgh, PA 15213 USA

December 7, 1996

Abstract

Real-time systems often spend an inordinate amount of time getting ready to do things in the future and deciding what to do next. Designating a task to be performed at some time in the future, or *scheduling*, and finding the next task to be run, or *dispatching*, typically take a total time which is linear in the number of waiting tasks. A new algorithm is presented in which the time for both scheduling and dispatching is bounded by a small constant. An additional constant load is placed on the processor, and a modest background processing load is also imposed. The new algorithm is compared to other popular real-time scheduler/dispatcher strategies.

1 Introduction

Most real-time systems have some mechanism for saying "perform this action at this future time" and many music systems spend a lot of their computation power in providing this capability. In this short paper, I will show how to dramatically improve the performance over that of the typical implementation.

The typical scheduler/dispatcher keeps a list of the tasks to be performed in the future. These tasks are sorted chronologically so that the next action to perform is the first action on the list. Systems typically inspect the list at regular intervals. When the current time equals or exceeds the performance time of the first item on the list, the action is performed and the first item of the list is discarded. Thus, the operation that starts performing a task, which we will call *dispatching*, can be accomplished in a fixed amount of time. (If several tasks can be scheduled for the same instant, the dispatching time is proportional to the number of ready tasks plus a constant.)

Let us now consider the *scheduling* operation which puts tasks into the list for removal by the dispatcher. In the worst case, the scheduler must search the entire list to find the right insertion point. Thus, the cost of scheduling is proportional to the total number of pending tasks. In music systems where this number can grow large, this is a serious problem. To make things worse, the worst case is a common one in which a task is scheduled at a time later than the time of any other tasks. One can easily treat this as a special case or use a doubly-linked list and search in decreasing time order, but the cost in the worst case is not reduced.

In the next section, I will look at a few interesting variations on schedulers and dispatchers. Then, I will present a scheduler/dispatcher that performs its operations in constant time (with a little additional work going on in the background).

2 Some Alternatives

What can we do to make scheduling more efficient? The typical implementation described above is an example of *linear search*, but faster methods, using other data structures, are known. One method is the use of balanced binary trees [1, 3]. A balanced binary tree allows items with time tags to be inserted in random order and removed in the order indicated by the time tags.

Unlike the list-based implementation, the insertion

[0]Published as: Dannenberg, "A Real Time Scheduler/Dispatcher," in *Proceedings of the International Computer Music Conference*, Computer Music Association, (September 1988), pp 239-242. P.S. If you know how to properly attach a footnote to a title in LaTex, please let me know. - rbd@cs.cmu.edu

time for a binary tree is proportional to the *logarithm* of the number of items (scheduled tasks) in the tree. If the number of tasks doubles, the list-based scheduler doubles its execution time in the worst case, but the tree-based scheduler execution time only increases by a small constant. Removing an item from the tree is just as expensive as inserting one, so dispatching is more costly than in constant-time list-based dispatchers. This is bad, considering that one may want to dispatch simultaneous events as efficiently as possible. On the other hand, every dispatch is preceded by a schedule operation, and the total cost in the tree-based implementation is significantly less than in the list-based one.

Can we do better? The answer is a qualified "yes." To do better we have to schedule tasks only at an integer number of clock ticks, and we must spend a small (and constant) amount of processor time on each tick. This is usually not a problem for music applications, and many programs already do this in the form of a "polling loop" that looks for ready-to-run tasks. (With the previous dispatchers, a programmable timer can be used to eliminate polling by interrupting the processor when it is time to dispatch the next task.)

To do better, we use a technique called *hashing*[1] [3] in which a table of lists rather than a simple list is used to remember scheduled tasks. If the table has N locations and we want to schedule a task for time T, then the task is put on the list at table location T *mod* N (the remainder of T divided by N). This technique is described Varghese and Lauck [5] and is used in the current implementation of Formula [2].

At each clock tick, the dispatcher only looks at one of the N lists. For this scheme to work, either the scheduler must sort the lists or the dispatcher must inspect every element on the list. If the scheduler sorts the lists, then scheduling time is proportional to the length of the list in the worst case and dispatching time is a constant. Alternatively, we can make scheduling take constant time by not sorting and have the dispatcher take time proportional to the number of list elements.

In any case, if N is larger than the number of pending tasks, and the scheduled times of those tasks is random, then the expected time for scheduling and dispatching is constant. The worst case, however, is quite bad. Suppose all tasks are scheduled at some multiple of N ticks. Then only one list will be used and the scheduler/dispatcher degenerates to the simple (and expensive) one described in the introduction. The tree-based system would work better.

[1] The technique I am about to describe is a specialized use of hashing. The technique could also be referred to as a modified bucket sort.

3 Improving the Worst Case

Once again, can we do better? As before, the answer is a qualified "yes." To do better, we must spend some additional processing time in a low-priority background task. The total time spent by the background task will be (in the worst case) proportional to the log of the number of pending tasks at each scheduling operation. Thus, the processing time is proportional to that of the tree-based implementation, but *all of this processing is done in the background.* The real-time processing required for scheduling and dispatching is constant, even in the worst case.

To achieve this performance, a combination of the tree-based and table-based implementations is used. The basic scheduling operation simply puts a task on a list of tasks to be scheduled. A background process takes tasks from the list and inserts them into a balanced binary tree. As tasks become almost ready to run, they are moved from the binary tree into a table as in the table-based implementation. The dispatcher advances through the table on each clock tick and performs all the tasks it encounters. This approach works for tasks that are scheduled far enough into the future to allow the background process time to handle them. To schedule a task in the near future, the scheduler bypasses the background processing and puts the task directly into the table. Having outlined the basic approach, I will now present a more detailed description of the implementation.

The most important aspect of this design is to make sure all scheduling and dispatching operations take only a constant amount of time. No time-critical operations can be required of the background task. Scheduling, background processing, and dispatching naturally lend themselves to a pipelined implementation as shown in Figure 1.

At time interval[2] i, the scheduler inserts tasks into a list which is passed to the background process at the beginning of time interval i+1. During interval i+1, the background process must insert the entire list into the tree structure and also remove any items in the tree scheduled for period i+2. These items are inserted into a table as in the table-based implementation. During period i+2, the tasks are performed by the dispatcher. In practice, all three stages of the

[2] Assume that a pipeline time interval takes many clock ticks.

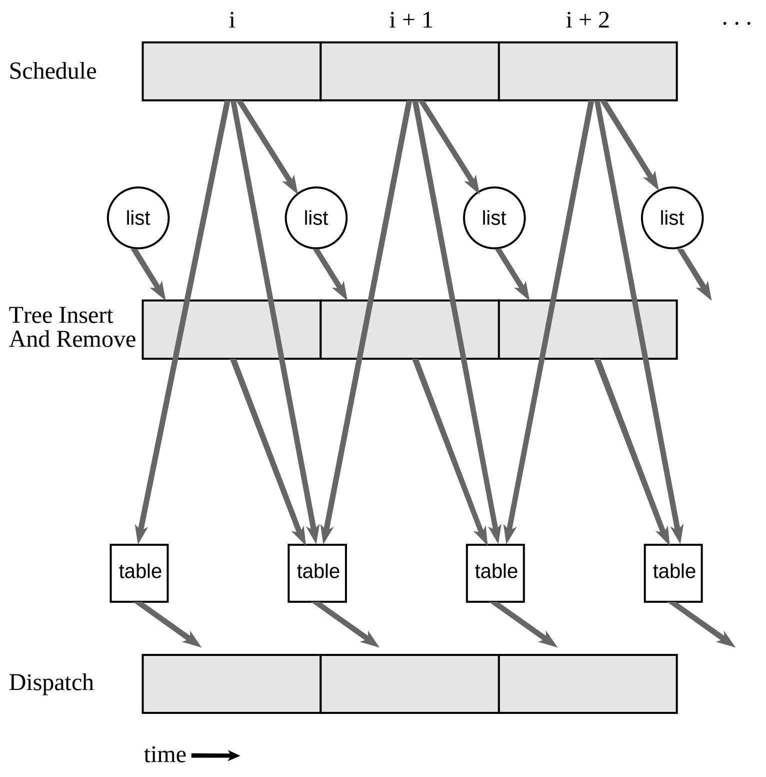

Figure 1: The pipelined implementation.

pipeline are run concurrently; that is, during interval i+1 the scheduler begins to create another list of tasks for the background process, and the dispatcher handles tasks that were previously scheduled.

There must be two active lists at any given time: one to accumulate new tasks from the scheduler and one with tasks to be moved into the tree. The latter will be empty at the end of each interval and can be reused by the scheduler.

There must also be two tables at any given time: during interval i, the dispatcher must read from one table holding tasks scheduled for interval i while the background process inserts tasks scheduled for interval i+1 into another table. Meanwhile, the scheduler takes any task scheduled in time interval i or i+1 and inserts it directly into the proper table. Tasks for intervals i+2 and beyond are stored on a list and passed on to the background process.

Tables can also be reused. At the end of interval i, the table being used by the dispatcher will be empty. This table can be used for tasks scheduled for interval i+2. Thus, only storage for two tables is required. The table size must equal the number of ticks in an interval.

4 Conclusion

Scheduling and dispatching can be performed in constant time even in the worst case, provided a small amount of time is available in the background. This compares favorably to list-based implementations (linear in the number of scheduled tasks), tree-based implementations (logarithmic in the number of tasks), and table-based implementations (constant time only in the expected case, linear time in the worst case). The dispatcher is extremely fast in that it does not even need to compare the current time with the scheduled time.

The background process must, in each interval, insert tasks scheduled in the previous interval and move tasks for the next interval from the tree structure to a table. The only time constraint is that the background process must complete all of its work within each interval. Bursts of scheduling activity can place extra load on the background process, but this load is spread evenly across the interval, which can be made as large as memory permits. (Memory requirements are essentially two pointers per tick of interval size because there are two tables of task lists.)

5 Acknowledgments

This scheduler was conceived in conversations with Ron Kuivila at the Second STEIM Symposium on Interactive Composition in Live Electronic Music. A more extensive presentation of this algorithm and a few related ones will appear in a forthcoming book [4]. This research was partially sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the: Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio, 45433-6543.

References

[1] Adel'son-Vel'skii, G. M. 1962. "An Algorithm for the Organization of Information." *Dokl. Akad. Nauk SSR* 146: 263-6 (in Russian). English translation in (1962) *Soviet Math. Dokl.* 3:1259-62.

[2] Anderson, D. and Kuivila, R. 1986. "Accurately Timed Generation of Discrete Musical Events." *Computer Music Journal* 10(3): 48-56.

[3] Horowitz, E. and Sahni, S. 1987. *Fundamentals of Data Structures in Pascal.* Computer Science Press.

[4] Matthews, M. and Pierce, J., editors. (to appear). *System Development Foundation Computer Music Benchmark.* M.I.T. Press.

[5] Varghese, G. and Lauck, T. 1987. "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility." *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* pp. 25-38, published as *Operating Systems Review* 21(5), ACM Order No. 534870.