Real-Time Systems, 2, 247–254 (1990)

Editorial

What is Predictability for Real-Time Systems?*

JOHN A. STANKOVIC AND KRITHI RAMAMRITHAM
Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, U.S.A

1. Introduction

Real-time systems span a broad spectrum of complexity from very simple microcontrollers (such as a microprocessor controlling an automobile engine) to highly sophisticated, complex and distributed systems (such as air traffic control for the continental United States). Some future systems will be even more complex (Stankovic 1988). These complex future systems include the space station, integrated vision/robotics/AI systems, collections of humans/robots coordinating to achieve common objectives (usually in hazardous environments such as undersea exploration or chemical plants), and various command and control applications. To further complicate the problem there are many dimensions along which real-time systems can be categorized. The main ones include:

- the granularity of deadlines and the laxities for tasks,
- the strictness of the deadlines,
- reliability requirements of the system,
- the size of the system and the degree of interaction (coordination) among components, and
- the characteristics of the environment in which the system operates.

The characteristics of the environment, in turn, seem to give rise to how static or dynamic the system has to be. As can be imagined, depending on the above considerations many different system designs occur. However, one common denominator seems to be that all designers want their real-time system to be *predictable*. But what does predictability mean? It means that it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made, for example, concerning failures and workloads. In other words, predictability is always subject to the underlying assumptions being made.

In this editorial we concentrate on predictability with respect to the timing requirements. We use examples to show that, in the current literature, predictability in real-time systems means different things depending on the requirements with respect to the above five dimensions. We then suggest two approaches for achieving predictability in complex real-time systems.

*This work was supported by ONR under contract N00014-85-K-0389 and NSF under grants DCR-8500332 and IRI-8908693.

We encourage readers to submit comments on this editorial, or to present their own definitions and views. Selected submissions will be published in future issues.

2. Categorizing real-time systems

Real-Time systems are those systems in which the correctness of the system depends not only on the logical results of computations, but also on the time at which the results are produced. However, the full meaning of this definition takes on various subtleties depending on the five system dimensions listed above. Let us now briefly discuss each of these dimensions in turn.

2.1. Granularity of the deadline and laxity of the tasks

In a real-time system some of the tasks have deadlines and/or periodic timing constraints. If the time between when a task is activated (required to be executed) and when it must complete execution is short, then the deadline is tight (i.e., the granularity of the deadline is small, or alternatively said, the deadline is close). This implies that the operating system reaction time has to be short, and the scheduling algorithm to be executed must be fast and very simple. Tight time constraints may also arise when the deadline granularity is large (i.e., from the time of activation), but the amount of computation required is also great. In other words even large granularity deadlines can be tight when the laxity (deadline minus computation time) is small. In many real-time systems tight timing constraints predominate and consequently designers focus on developing very fast and simple techniques to react to this type of task activation.

2.2. Strictness of deadline

The strictness of the deadline refers to the value of executing a task after its deadline. For a *hard real-time task* there is no value to executing the task after the deadline has passed. A *soft real-time task* retains some diminished value after its deadline so it should still be executed. Very different techniques are usually used for hard and soft real-time tasks. In many systems hard real-time tasks are preallocated and prescheduled resulting in 100% of them making their deadlines. Soft real-time tasks are often scheduled either with non-real-time scheduling algorithms, or with algorithms that explicitly address the timing constraints but aim only at good average case performance, or with algorithms that combine importance and timing requirements (for example, cyclic scheduling).

2.3. Reliability

Many real-time systems operate under severe reliability requirements. That is, if certain tasks, called critical tasks, miss their deadline then a catastrophe may occur. These tasks are usually guaranteed to make their deadlines by an off-line analysis and by schemes that reserve resources for these tasks even if it means that those resources are idle most of the

time. In other words, the requirement for critical tasks should be that all of them always make their deadline (a 100% guarantee), subject to certain failure and workload assumptions. However, it is our opinion that too many systems treat all the tasks that have hard timing constraints as critical tasks (when, in fact, only some of those tasks are truly critical). This can result in erroneous requirements and an overdesigned and inflexible system. It is also common to see hard real-time tasks defined as those with both strict deadlines and of critical importance. We prefer to keep a clear separation between these notions because they are not always related.

2.4. Size of system and degree of coordination

Real-time systems vary considerably in size and complexity. In most current real-time systems the entire system is loaded into memory, or if there are well defined phases, each phase is loaded just prior to the beginning of the phase. In many applications, subsystems are highly independent of each other and there is limited cooperation among tasks. The ability to load entire systems into memory and to limit task interactions simplifies many aspects of building and analyzing real-time systems. However, for next generation large, complex, real-time systems, having completely resident code and highly independent tasks will not always be practical. Consequently, increased size and coordination give rise to many new problems that must be addressed and complicates the notion of predictability.

2.5. Environment

The environment in which a real-time system is to operate plays an important role in the design of the system. Many environments are very well defined (such as a lab experiment, an automobile engine, or an assembly line). Designers think of these as deterministic environments (even though they may not be intrinsically deterministic, they are forced to be). These environments give rise to small, static real-time systems where all deadlines can be guaranteed a priori. Even in these simple environments we need to place restrictions on the inputs. For example, the assembly line can only cope with five items per minute; given more than that, the system fails. Taking this approach enables an off-line analysis where a quantitative analysis of the timing properties can be made. Since we know exactly what to expect given the assumptions about the well defined environment we can consider these systems to be predictable.

The problem is that the approaches taken in relatively small, static systems do not scale to other environments which are larger, much more complicated, and less controllable. Consider a next generation real-time system such as a team of cooperating mobile robots on Mars. This next generation real-time system will be large, complex, distributed, adaptive, contain many types of timing constraints, need to operate in a highly nondeterministic environment, and evolves over a long system lifetime. It is much more difficult to force this environment to look deterministic—in fact, that is exactly what you do not want to do because the system would be too inflexible and would not be able to react to unexpected events or combinations of events. We consider this type of real-time system to be a dynamic

real-time system operating in a nondeterministic environment. Such systems are required in many applications. It is much more difficult to define predictability for these systems and the typical semantics (all tasks make their deadlines 100% of the time) associated with the term for small static real-time systems is not sufficient. Many advances are required to address predictability of these next generation systems in a scientific manner. For example, one of the most difficult aspects will be in demonstrating that these systems meet both their overall performance requirements (which are generally average case statistics but with respect to meeting deadlines and maximizing value of executed tasks), as well as specific deadline and periodicity timing requirements of individual tasks or groups of tasks, or instances thereof. If both types of timing requirements can be demonstrated, then we can refer to the system as being predictable.

3. Predictability

For tutorial purposes, let us consider a series of examples of systems which are called predictable in the literature.

3.1. Predictability in a simple system

System A has a fixed worst case interrupt response time of 30 microseconds. This system is called predictable because you can guarantee that any new interrupt is *begun* to be serviced within 30 microseconds. Any overlapping interrupt is also guaranteed to begin to be processed in 30 microseconds, but this may impact the completion of the previous task. Consequently, there is no guarantee that a task will *finish* by its deadline. That requirement must be handled in some other manner. In other words, the system is predictable *only* with respect to the requirement that interrupts are to begin being processed in 30 microseconds. Consequently, in System A, the claim of predictability is an extremely tenuous one and offered at a microscopic level. However, assume that you have a very simple system where at most two overlapping interrupts are possible, where interrupts of the same type do not occur within 100 microseconds of each other, and all tasks that execute in response to an interrupt need five microseconds and that deadlines for these tasks are 100 microseconds. Then you can a priori show the macroscopic predictability, that is, the system level predictability. Note that here predictability means that all tasks make their deadlines with 100% guarantee. We emphasize again that often requiring 100% guarantees for all tasks is unnecessary. Implicit in this guarantee are assumptions such as no failures occur and various execution times and deadlines are correct.

3.2. Predictability in systems with only periodic tasks

In system B there are 100 periodic tasks, they are all hard real-time tasks and 25 are critical. The designers decide to treat all 100 as critical and produce a fixed cyclic schedule for these tasks. An underlying assumption in producing the schedule is that the worst case

execution times for the tasks are known which means that the hardware instruction times must be known. To facilitate this computation the designers eliminate caches and virtual memory from their design. All operating system primitives needed by these 100 tasks must also be accounted for, hence the execution time of each OS primitive must be bounded. Any precedence constraints between the tasks and conflicts over shared resources must be accounted for. At this point the system is completely fixed and inflexible, but there is a 100% guarantee that all deadlines will be met. The system meets its timing requirements. This is again subject to no failures and to the worst case times being accurate including maximum delays in waiting for shared resources.

In system C there are 25 independent periodic tasks. Suppose that the designers can show that the maximum cpu utilization ever required by these 25 tasks is less than 69%. Hence, they use the rate monotonic algorithm which guarantees that all tasks will make their deadlines if utilization is less than 69%. An underlying assumption behind the rate monotonic algorithm is that preemption costs are zero. However, since the number of preemptions is bounded it is possible to take the preemption costs into account. Consequently, we have 100% predictable performance even though a dynamic scheduling algorithm is used.

3.3. Predictability in a next generation, complex real-time system

System D is a large, complex, real-time system operating in a nondeterministic environment. It has both tight time constraints and loose time constraints; it has hard and soft real-time tasks; some of its tasks are critical; parts of the system may be highly static but many parts require a dynamic approach. In other words all dimensions of real-time systems are found simultaneously in this system. Designers cannot focus on just one predominant feature of the system such as tight time constraints, or simply bound the interrupt latency, or assume a fixed set of periodic tasks. It is this type of system for which it is difficult to define and demonstrate predictability. The notion that you obtain a 100% guarantee (as found in all the previous examples) must be relaxed.

We believe that you must demonstrate that the requirements are met at two levels of detail and for each class of task. At the macroscopic (system-wide) level we need to show first that all critical tasks will always make their deadline (100% guarantee) and that all noncritical tasks (both hard and soft deadline tasks) meet overall requirements. As an example, these overall requirements might be stated in the following way: 97% of noncritical hard real-time tasks, and 95% of soft real-time tasks must make their deadlines. Alternatively, the requirements might be to maximize the value of noncritical hard and soft real-time tasks executed by the system. At the microscopic level (on an individual task or task group basis) we also would like some level of predictability. For the critical tasks we already know that they will always make their deadline. For the other real-time tasks their performance will depend on the current state of the system. However, using scheduling algorithms in a planning mode (as done in Stankovic and Ramamritham 1989) can provide an instantaneous snapshot of the current predictability of this set of tasks. In other words, at any point in time the system can identify exactly which tasks will make their deadlines. This provides many advantages including graceful degradation, the ability to handle overloads, and the ability to make more intelligent decisions concerning the overall operation of the system (Stankovic and Ramamritham 1989).

4. Achieving predictability

We now consider two alternative ways to achieve predictability in complex real-time systems. The first we call the *layer-by-layer* approach and the second we call the *top layer* approach. In the first case we require both microscopic and macroscopic predictability, while in the second case we only require macroscopic predictability. It is important to note that these approaches can be merged.

Before we discuss each of these approaches we have a few preparatory remarks. A real-time system can be considered to be composed of entities at various hardware and software layers. Broadly speaking these levels are: semiconductor components, the hardware/architecture layer, the operating system layer, and the application layer. The layer-by-layer method assumes that a higher layer is predictable, if and only if, the lower layer is predictable. The top layer approach challenges this statement.

4.1. Layer-by-layer approach

In this approach, in order to obtain a predictable system, it is necessary to have a tight interaction between all aspects of the system starting from the design rules and constraints used, to the programming language, to the compiler, to the operating system, and to the hardware. Then, based on a careful software and hardware design we believe that it is possible to achieve both microscopic and macroscopic predictability. In the microscopic view, we can compute the worst case execution time of any task. This is not as simple as it first may seem. First, we require a simplified architecture so that instruction times are well-defined. Second, we must be able to account for resource requirements and calls to system primitives made on behalf of this task. This can be accomplished via various techniques including a *planning* scheduler such as found in the Spring system (Ramamritham, Stankovic and Shiah 1990; Stankovic and Ramamritham 1987; Stankovic and Ramamritham 1989; Zhao, Ramamritham and Stankovic 1987). In this way, the execution time of a particular invocation of a task with its resource needs can be accurately computed. In many other approaches predictability breaks down here because they have no good method for dealing with delays for resources.

Further, the layer-by-layer approach enables a macroscopic view of predictability. That is, first, we require the *macroscopic* view that *all* critical tasks will *always* make their deadlines (subject to the assumptions of the analysis). In other words, for critical tasks the requirement is a 100% guarantee. As mentioned above some systems force all their tasks to be critical. This has a number of disadvantages and will not scale to next generation, large, and dynamic systems. Second, by planning and through microscopic predictability, at any point in time we know *exactly* which noncritical but hard real-time tasks in the entire system will make their deadlines given the current load. In other words we have a dynamic and macroscopic picture of the capabilities of the current state of the system with respect to timing requirements. This has several advantages with respect to fault tolerance and graceful degradation. Third, it is also possible to develop an overall quantitative, but probabilistic assessment of the performance of noncritical hard real-time tasks given expected normal and overload workloads. For example, via simulation one might compute

the average percentage of noncritical tasks that make their deadlines or the expected value of tasks that make their deadline. We then would need to show that on the average these tasks meet the system requirements or add resources until this is true. Fourth, we require the macroscopic view of the capabilities of the I/O front ends. For example, it may be possible to state that the tasks on the I/O processor, scheduled according to the rate monotonic algorithm, will always make all their deadlines because the load is less than 69% and because there are no resource conflicts.

In some circles this four pronged macroscopic view may seem unsatisfying because *everything* is not 100% guaranteed. However, we believe that this is necessary and unavoidable given that we are operating in a complex, nondeterministic environment. In these environments it seems necessary to *carefully* develop the requirements as actually needed, and then to employ different means to meet the different types of requirements.

4.2. Top layer approach

In the top layer approach we concentrate on the fact that what is really important is the *application layer* predictability requirements. Deadlines imposed on activities other than those that occur at the application layer are artifacts. The lower layers need not be predictable, but only provide services so that the application layer is predictable.

This approach also recognizes that in the layer-by-layer approach, even if everything has been predicted—at each of the layers—to work as required, error handlers have to be provided for fail-safe and fail-soft behavior. This is just in case assumptions made to derive predictability figures do not hold, or the algorithms used to derive predictability are in error.

Let us first consider the requirements for hard real-time tasks. Suppose the requirement imposed is that either an activity with a hard deadline D is done completely by its deadline, or some alternative action is taken that is guaranteed to take the system to a safe state. The alternative action should also be completed by the deadline. The alternative action can be thought of as an error handler.

Assuming that the computation time C for the error handler is easier to determine than for the original activity, the system can do the following: Before the activity begins, it guarantees with 100% that the error handler, when started at time D′ ($D' \leq D - C$) will run and complete prior to D. The lower layers run as fast as they can to provide the services required so that the activity is completed by D′. If this fails, the error handler is invoked.

To use this approach, it must be possible to demonstrate that the error handler can execute by its deadline with 100% guarantee. To guarantee this, some of the techniques used in the layer-by-layer approach may be necessary, although applied to a small subset of the system (the error handlers) rather than to the entire system. It might even be possible to allocate a separate subsystem dedicated for the error handlers, so that the various layers of this subsystem are carefully designed using the layer-by-layer approach to provide the needed guarantees.

Now consider the requirements for the critical tasks. Critical activities, by the semantics given earlier, must be completed by their deadlines. So, for such activities, the layer-by-layer approach has to be used to provide 100% guarantees.

The top-layer approach is most suited when the activities are complex. For instance, it may not be possible to "break" a complex activity at one layer into activities/tasks assumed

at the lower layers. Further, the uncertainties caused by the environment in which an activity is being performed may make it impossible to quantitatively describe the behavior of components of that activity.

Another situation where the top layer approach will be called for is where the bounds found at one layer, based on bounds at the lower layers, may be so high so as not to be of any practical value. Let us look at an example. Suppose an activity A is made up of A1 that happens on node p, A2 that happens on node q and communication C between them. Say A1 and A2 are subproblems such that if A1 takes more time, then A2 takes less, and vice versa. If A1 does more work, C takes more time. Whether or not A1 takes more time depends on various factors not all of which are known at the time the activity begins. However, A1 and A2 are bounded. So is C. Here all the component activities are bounded. So the overall activity is also bounded. However, this bound may be too large to be useful in making any prediction, say, whether A will meet its timing requirement.

5. Conclusion

In summary, predictability in real-time systems has been defined in many ways. For static real-time systems we can predict the overall system performance over large time frames (even over the life of the system) as well as predict the performance of individual tasks. If the prediction is that 100% of all tasks over the entire life of the system will meet their deadlines, then the system is predictable without resorting to any stochastic evaluation. In dynamic real-time systems we must resort to a stochastic evaluation for part of the performance evaluation. Predictability for these systems should mean that we are able to satisfy the timing requirements of critical tasks with 100% guarantee over the life of the system, be able to assess overall system performance over various time frames (a stochastic evaluation), and be able to assess individual task and task group performance at different times and as a function of the current system state. If all these assessments meet the timing requirements, then the system is predictable with respect to its timing requirements.

References

Ramamritham, K., Stankovic, J., and Shiah, P. 1990. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, 184–194.

Stankovic, J. and Ramamritham, K. 1987. The design of the Spring kernel. *Proc. 1987 Real-Time Systems Symposium*, Dec.

Stankovic, J. and Ramamritham, K. 1989. The Spring kernel: A new paradigm for real-time operating systems. *ACM Operating Systems Review*, vol. 23, 54–71.

Stankovic, J. 1988. Misconceptions about real-time computing. *IEEE Computer*, vol. 21.

Zhao, W., Ramamritham, K., and Stankovic, J. 1987. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*.