Real-Time Computing

Professor John A. Stankovic
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

April 16, 1992

1 What is a Real-Time System

Real–time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. 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). Other examples of real-time systems include command and control systems, process control systems, flight control systems, the space shuttle avionics system, flexible manufacturing applications, the space station, space-based defense systems, intensive care monitoring, collections of humans/robots coordinating to achieve common objectives (usually in hazardous environments such as undersea exploration or chemical plants), intelligent highway systems, and multimedia and high speed communication systems. We are also beginning to see many of these real-time systems adding expert systems [22] and other AI technology creating additional requirements and complexities. Real-time systems technology is a key *enabling* technology for the future in an ever-growing domain of important applications. Because of this fact, it is important to understand the current technology and its limitations and to devote significant effort for improving the technology.

Typically, a real–time system consists of a *controlling system* and a *controlled system*. For example, in an automated factory (Figure 1), the controlled system is the factory floor with its robots, assembling stations, and the assembled parts, while the controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor. Thus, the controlled system can be viewed as the *environment* with which the computer interacts.

The controlling system interacts with its environment based on the information available about the environment from various sensors. It is imperative that the state of the environment, as perceived by the controlling system, be consistent with the actual state of the environment. Otherwise, the effects of the controlling systems' activities may be disastrous. Hence, periodic monitoring of the environment as well as timely processing of the sensed information is necessary.

Timing correctness requirements in a real–time system arise because of the *physical impact* of the controlling systems' activities upon its environment. For example, if the computer controlling a robot does not command it to stop or turn on time, the robot might collide with another object on the factory floor possibly causing serious damage. In many real–time systems even more severe consequences will result if timing as well as logical correctness properties of the system are not satisfied, e.g., consider nuclear power plants or air traffic control systems failing.

Timing constraints for tasks can be arbitrarily complicated but the most common timing constraints for tasks are either *periodic* or *aperiodic*. An aperiodic task has a deadline by which it must finish or start, or it may have a constraint on both start and finish times. In the case of a periodic task, a period might mean 'once per period T' or 'exactly T units apart'.

Low-level application tasks, such as those that process information obtained from sensors, or those that activate elements in the environment (through actuators) typically have stringent timing constraints dictated by the physical characteristics of the environment. A majority of sensory processing is periodic in nature. For example, a radar that tracks flights produces data at a fixed rate. A temperature monitor of a nuclear reactor core should be read periodically to detect any changes promptly. Some of these periodic tasks may exist from the point of system initialization while others may come into existence dynamically. The temperature monitor above is an instance of a permanent task. An example of a dynamically created task is a (periodic) task that monitors a particular flight; this comes into existence when the aircraft enters an air traffic control region and will cease to exist when the aircraft leaves the region.

More complex types of timing constraints also occur. For example, spray painting a car on a moving conveyor must be started after time t_1 and completed before time t_2. Aperiodic requirements can arise from dynamic events such as an object falling in front of a moving robot, or a human operator pushing a button on a console.

Time related requirements may also be specified in indirect terms. For example, a value may be attached to the completion of each task where the value may increase or decrease with time; or a value may be placed on the *quality* of an answer whereby an inexact but fast answer might be considered more valuable than a slow but accurate answer. In other situations, missing X deadlines might be tolerated, but missing $X+1$ deadlines can't be tolerated.

What happens when timing constraints are not met? The answer depends, for the

most part, on the type of application. A real–time system that controls a nuclear power plant or one that controls a missile, cannot afford to miss timing constraints of the critical tasks. Resources needed for critical tasks in such systems have to be preallocated so that the tasks can execute without delay. In many situations, however, some leeway does exist. For example, even on an automated factory floor, if it is estimated that the correct command to a robot cannot be generated on time, it may be appropriate to command the robot to stop (provided it will not cause other moving objects to collide with it and result in a different type of disaster), or to slowdown (thereby dynamically generating more time to produce a correct command). Another example is a periodic task monitoring the position of an aircraft; depending on the aircraft's location and trajectory, missing the processing of one or two radar readings may not cause any problems.

In a real-time system, the characteristics of the various application tasks are usually known *a priori* and might be scheduled statically or dynamically. While static specification of schedules is typically the case for periodic tasks, the opposite is true for aperiodic tasks. When the periodic temperature monitor of our previous nuclear reactor example senses a problem in the core, it can invoke another (aperiodic) task to activate the appropriate elements of the reactor to correct the problem, for example, to force more coolant into the reactor core. In this case, the deadline for the aperiodic task can be statically determined as a function of the physical characteristics of the reactions within the core. On the other hand, the deadline of a task that controls a robot in a factory floor can be determined dynamically depending on the speed, direction, etc. of the robot. The command to the robot forcing it to turn right, left, or stop should be generated before this deadline.

In a real-time system that is designed in a static manner, the characteristics of the controlled system are assumed to be known *a priori* and hence the nature of activities and the sequence in which these activities take place can be determined off-line before the system begins operation. Such systems are quite inflexible even though they may incur lower run-time overheads. In practice, most applications involve a number of components that can be statically specified along with many dynamic components. If handled appropriately, a system with high resource utilization and low overheads can be produced for such applications.

A large proportion of currently implemented real–time systems are static in nature, but by necessity, next generation systems will have to adopt solutions that are more dynamic and flexible. This is because such systems will be large and complex and they will function in environments that are uncertain while being physically distributed. More importantly, they will have to be maintainable and extensible due to their evolving nature and projected long lifetimes. Because of these characteristics, real–time systems in general, and systems with the above characteristics in particular, need to be *fast*, *predictable*, *reliable*, and *adaptive*.

An important ingredient in understanding what a real-time system is, is understanding what a real-time system is not. This issue has been carefully discussed in a prior publication [14] so will not be detailed here. However, one long-held misconceptions about real-time computing is that they only need to be fast. This misconception is so prevalent that we will briefly discuss it here. Basically, being fast is usually a necessary condition, but it is not sufficient. A real-time system has to meet explicit deadlines and being fast on the average does not guarantee that a deadline will be met. This reminds me of the famous quote "... then there is the man who drowned crossing the stream with an average depth of six inches...". If a real-time system can be shown to meet its deadlines (using a worst case behavior rather than average case behavior analysis), then we say that it is predictable. Predictability, itself, has many meanings and an entire journal article has been devoted to its meaning [15]. For purposes of this article it is sufficient to take a simplistic view of predictability. Consider that predictability means that when a task or set of tasks is activated it should be possible to determine its completion time subject to failure assumptions. This must be done taking into account the state of the system (including the state of the operating system and the state of the resources controlled by the operating system) and the tasks' resource needs.

In summary, real-time systems differ from traditional systems in that deadlines or other explicit timing constraints are attached to tasks, the systems are in a position to make compromises, and faults including timing faults may cause catastrophic consequences. This implies that, unlike many systems where there is a separation between correctness and performance, in real–time systems correctness and performance are very tightly interrelated.

2 Characterizing Real-Time Systems

Building a real-time system can vary from a simple task to an extremely complex task where the state of the art is still lacking. The difficulty depends on the characteristics of the real-time system along, at least, five dimensions which we discuss below.

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. In general, the tighter the deadline the more difficult the design task.

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 cases 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 (e.g., cyclic scheduling). Hard real-time tasks are more difficult to deal with than soft real-time tasks and systems which must deal with both types simultaneously are yet even more difficult.

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. Of course, many other reliability issues must be solved, but here we chose only to mention the key issue that deals with timing constraints and reliability.

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. Yet, typical solutions based on virtual memory are not acceptable because of the large degree of predictability injected by this technique.

Consequently, increased size and coordination give rise to many new problems that must be addressed and complicates the notion of predictability.

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 controlled and assumed 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 usually design and build 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 non-deterministic environment, and evolves over a long system lifetime. It is not possible to assume that this environment is deterministic nor to control it sufficiently well to make it 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 non-deterministic environment. Such systems are required in many (future) applications. Many new results are required before we can build reliable and safe systems of this type.

3 Achieving Real-Time Performance

Achieving quantifiable real-time performance requires integrated solutions across many areas. In this section we discuss real-time kernels, real-time scheduling, and real-time architectures and fault tolerance. Due to space limitations many other areas of real-time systems are not discussed such as programming languages, communication protocols [1], distributed systems [3, 8, 9, 11, 21] and design methodologies.

3.1 Real-Time Kernels

One focal point for next generation real-time systems is the operating system. The operating system must provide basic support for predictably satisfying real-time constraints, for fault tolerance and distribution, and for integrating time-constrained resource allocations and scheduling across a spectrum of resource types including sensor processing, communications, CPU, memory, and other forms of I/O. Towards this end, at least three major scientific issues need to be addressed.

- The *time dimension* must be elevated to a central principle of the system and should not be simply an afterthought. An especially perplexing aspect of this problem is that most system specification, design and verification techniques are based on abstraction — which ignores implementation details. This is obviously a good idea; however, in real-time systems, timing constraints are derived from the environment and the implementation. This dilemma is a key scientific issue.

- The basic paradigms found in today's general purpose distributed operating systems must change. Currently, they are based on the notion that application tasks request resources as if they were random processes; operating systems are designed to expect random inputs and to display good average-case behavior. The new paradigm must be based on the delicate balance of *flexibility* and *predictability*: the system must remain flexible enough to allow a highly dynamic and adaptive environment, but at the same time be able to predict and possibly avoid resource conflicts so that timing constraints can be met [17]. This is especially difficult in distributed environments where layers of operating system code and communication protocols interfere with predictability.

- A highly *integrated and time-constrained resource allocation approach* is necessary to adequately address timing constraints, predictability, adaptability, correctness, safety, and fault tolerance. For a task to meet its deadline, resources must be available *in time*, and events must be ordered to meet precedence constraints. Many coordinated actions are necessary for this type of processing to be accomplished on time. The state of the art lacks completely effective solutions to this problem.

Existing practices for designing, implementing, and validating real-time systems of today are still rather *ad hoc*. It is often the case that existing real–time systems are supported by stripped down and optimized versions of timesharing operating systems. To reduce the run-time overheads incurred by the kernel and to make the system *fast*, the kernel underlying a current real–time system

- has a fast context switch,

- has a small size (with its associated minimal functionality),
- responds to external interrupts quickly,
- minimizes intervals during which interrupts are disabled,
- provides fixed or variable sized partitions for memory management (i.e., no virtual memory) as well as the ability to lock code and data in memory, and
- provides special sequential files that can accumulate data at a fast rate.

To deal with timing requirements the kernel,

- maintains a real-time clock,
- provides a priority scheduling mechanism,
- provides for special alarms and timeouts, and
- tasks can invoke primitives to delay by a fixed amount of time and to pause/resume execution.

In general, the kernels perform multi-tasking; inter-task communication and synchronization are achieved via standard, well-known primitives such as mailboxes, events, signals, and semaphores.

In real-time computing these features are also designed to be *fast*. However, as mentioned above, fast is a relative term and not sufficient when dealing with real–time constraints. Nevertheless, many real-time system designers believe that these OS features provide a good basis upon which to build real–time systems. Others believe that such features provide almost no direct support for solving the difficult timing problems and would rather see more sophisticated kernels that directly address timing and fault tolerance constraints.

One key issue that comes up over and over again is the need to provide predictability. However, more than lip service must be supplied. Predictability requires bounded operating system primitives, some knowledge of the application, proper scheduling algorithms, and a viewpoint based on a *team* attitude between the operating system and the application. For example, simply having a very primitive kernel that is itself predictable is seen as only the first step. What is needed is more direct support for developing predictable and fault tolerant real-time applications. One aspect of this support comes in the form of scheduling algorithms. For example, if the operating system is able to perform integrated CPU scheduling and resource allocation in a planning mode so that collections of cooperating tasks can obtain the resources they need, at the right time, in order to meet timing constraints, this facilitates the design

and analysis of real-time applications [17]. Further, if the operating systems retains information about the importance of a task and what actions to take if the task is assessed as not being able to make its deadline, then a more intelligent decision can be made as to alternative actions, and graceful degradation of the performance of the system can be better supported (rather than a possible catastrophic collapse of the system if no such information is available). Kernels which support retaining and using semantic information about the application are sometimes referred to as *reflective* kernels [16].

Real-time kernels are also being extended to operate in highly cooperative multiprocessor and distributed system environments. This means that there is an end-to-end timing requirement (in the sense that a set of communication tasks must complete before a deadline), i.e., a collection of activities must occur (possibly with complicated precedence constraints) before some deadline. Much research is being done on developing time constrained communication protocols to serve as a platform for supporting this user level end-to-end timing requirement. However, while the communication protocols are being developed to support host-to-host bounded delivery time, using the current operating system paradigm of allowing arbitrary waits for resources or events, or treating the operation of a task as a *random process* will cause great uncertainty in accomplishing the application level end-to-end requirements. As an example, the Mars project [3], the Spring project [17], and a project at the University of Michigan [11] are all attempting to solve this problem. The Mars project uses an *a priori* analysis and then statically schedules and reserves resources so that distributed execution can be guaranteed to make its deadline. The Spring approach support dynamic requests for real-time virtual circuits (guaranteed delivery time) and real-time datagrams (best effort delivery) integrated with CPU scheduling so as to guarantee the application level end-to-end timing requirements. The Spring project uses a distributed replicated memory based on a fiber optic ring to achieve the lower level predictable communication properties. The Michigan work also supports dynamic real-time virtual circuits and datagrams, but their work is based on a general multi-hop communication subnet.

Research is also being done on developing real-time object oriented kernels [20, 21] to support the structuring of distributed real-time applications [12]. As far as we know, no commercial products of this type are available.

The diversity of the applications requiring predictable distributed systems technology will be significant. To handle this diversity, we expect the distributed real-time operating systems must use an *open system* approach. It is also important to avoid having to rewrite the operating system for each application area which may have differing timing and fault tolerance requirements. A library of real–time operating system objects might provide the level of functionality, performance, predictability, and portability required. We envision a Smalltalk like system for hard real–time, so that a designer can tailor the OS to his application without having to write everything from

scratch. In particular, a library of real-time scheduling algorithms should be available that can be plugged in depending on the run time task model being used and the load, timing, and fault tolerance requirements of the system.

3.2 Real-Time Scheduling

Real-time scheduling results in recent years have been extensive. Theoretical results have identified worst case bounds for dynamic on-line algorithms, and complexity results have been produced for various types of assumed task set characteristics. Queueing theoretic analysis has been applied to soft real-time systems covering algorithms based on real-time variations of FCFS, earliest deadline, and least laxity. We have seen the development of scheduling results for imprecise computation (a situation where tasks obtain a greater value the longer they execute up to some maximum value). More applied scheduling results have also been produced with an extensive set of improvements to the rate monotonic algorithm (this includes the deferrable server and sporadic server algorithms [13]), techniques to address the problem of priority inversion [10], and a set of algorithms that perform dynamic on-line planning [7, 23]. We have also seen practical application of *a priori* calculation of static schedules to provide what is called 100% guarantees for critical tasks. While these *a priori* analyses are very valuable, system designers better not be lulled into thinking that 100% guarantees mean that no scheduling error can occur. It is important to know that these 100% guarantees are based on many and often times unrealistic assumptions. If the assumptions are a poor match for what can be expected from the environment (more and more likely in a distributed environment), then even with 100% guarantees the system will indeed miss deadlines. Hence, a key issue is to choose an algorithm whose assumptions provides the greatest coverage over what *really* happens in the environment. For all these scheduling results outlined above, the trend has been to deal with slightly more and more complicated task set and environment characteristics (e.g., multiprocessing and distributed computing and task with precedence constraints). While many interesting scheduling results have been produced, the state of the art still provides piecemeal solutions. Many realistic issues have not yet been addressed in an integrated and comprehensive manner.

What is still required are analyzable scheduling approaches (it may be a collection of algorithms) that are comprehensive and integrated. For example, the overall approach must be comprehensive enough to handle:

- preemptable and non-preemptable tasks,
- periodic and non-periodic tasks,
- tasks with multiple levels of importance (or a value function),

- groups of tasks with a single deadline,
- end-to-end timing constraints,
- precedence constraints,
- communication requirements,
- resource requirements,
- placement constraints,
- fault tolerance needs,
- tight and loose deadlines, and
- normal and overload conditions.

The solution must be integrated enough to handle the interfaces between:

- CPU scheduling and resource allocation,
- I/O scheduling and CPU scheduling,
- CPU scheduling and real-time communication scheduling,
- local and distributed scheduling [2, 8, 19], and
- static scheduling of critical tasks and dynamic scheduling of essential and non-essential tasks.

3.3 Real-Time Architecture and Fault Tolerance

Real–time systems are usually special purpose. In the past, architectures to support such applications tended to be special purpose too. The current trend is one in which more "off-the-shelf" components (See Figure 2) are being used to produce more generic architectures [11, 18]. While considerable discussion could be given to real-time architectures, due to space limitations, we only briefly consider how architecture impacts the computation of worst case execution time and how it supports fault tolerance.

One aspect of architecture for real-time computing is the facility with which the worst case execution time can be calculated. Worst-case execution times of programs are dependent on the system hardware, the operating system, the compiler used, and the programming language used. Many hardware features that have been introduced to speed-up the average case behavior of programs pose problems when information about worst case behavior is sought. For instance, the ubiquitous caches, pipelining,

dynamic RAMs, and virtual (secondary) memory, lead to highly nondeterministic hardware behavior. Similarly, compiler optimizations tailored to make better use of these architectural enhancements as well as techniques such as constant folding contribute to poor predictability of code execution times. System interferences due to interrupt handling, shared memory references, and preemptions are additional complications. In summary, any approach to the determination of execution times of real-time programs has many complexities, but they must be solved for real-time computing.

Many real-time system architectures consist of multiprocessors, networks of uniprocessors, or networks of uni- and multi-processors. Such architectures have potential for high fault tolerance, but are also much more difficult to manage in a way such that deadlines are predictably met. Fault tolerance must be designed in at the start, must encompass both hardware and software, and must be integrated with timing constraints. In many situations, the fault tolerant design must be static due to extremely high data rates and severe timing constraints. Ultrareliable systems need to employ proof of correctness techniques to ensure fault tolerance properties. Primary and backup schedules computed off–line are often found in hard real–time systems. We also see new approaches where on–line schedulers predict that timing constraints will be missed, enabling early action on such faults. Dynamic reconfigurability is needed but little progress has been reported in this area. Also, while considerable advance has been made in the area of software fault-tolerance, techniques that explicitly take timing into account are lacking.

Since fault tolerance is difficult, the trend is to let experts build the proper underlying support for it. For example, implementing checkpointing, reliable atomic broadcasts, logging, lightweight network protocols, synchronization support for replicas, and recovery techniques, and having these primitives available to applications, then simplifies creating fault tolerant applications. However, many of these techniques have not carefully addressed timing considerations nor the need to be predictable in the presence of failures. Many real-time systems which require a high degree of fault tolerance have been designed with significant architectural support but the design and scheduling to meet deadlines is done statically, with all replicas in lock step. This may be too restrictive for many future applications. What is required is the integration of fault tolerance and real-time scheduling to produce a much more flexible system. For example, the use of the imprecise computation model [4], or a planning scheduler [7] gives rise to a more flexible approach to fault tolerance than static schedules and fixed backup schemes.

4 New Trends and Technologies

Achieving complex, real-time systems is non-trivial and will require research breakthroughs in many aspects of system design and implementation. For example, good design methodologies and tools (SEE ARTICLE IN THIS ISSUE) which include programming rules and constraints must be used to guide real–time system developers so that subsequent implementation and *analysis* can be facilitated. This includes proper application decomposition into subsystems and allocation of those subsystems onto distributed architectures. The programming language must provide features tailored to these rules and constraints, must limit its features to enhance predictability, and must provide the ability to specify timing, fault tolerance and other information for subsequent use at run time. Many language features are continuously being proposed although few of them are used in practice, to date. Execution time of each primitive of the kernel must be bounded and predictable, and the operating system should provide explicit support for all the requirements including the real–time requirements [17]. New trends in the OS area include the use of microkernels, support for multiprocessors and distributed systems, and real-time thread packages. The architecture and hardware must also be designed to support predictability and facilitate analysis. For example, hardware should be simple enough so that predictable timing information can be obtained. This has implications for how to deal with caching, memory refresh and wait states, pipelining, and some complex instructions which all contribute to timing analysis difficulties. The resulting system must be scalable to account for the significant computing needs initially and as the system evolves. An insidious aspect of critical real–time systems, especially with respect to the real–time requirements, is that the weakest link in the entire system can undermine careful design and analysis at other levels. Research is required to address all of these issues in an integrated fashion. To satisfy this need for systems integration various Centers for real-time computing have been established such as CRICCS (Center for Real-Time, Intelligent, Complex Computer Systems) at the University of Massachusetts. (SEE SIDEBAR ON CRICCS CENTER). Finally, two very new trends involve the development of real-time databases and real-time artificial intelligence. Since these areas are very new and less well known, we discuss them in more detail below.

4.1 Real-Time Databases

A real-time database is a database system where (at least some) transactions have explicit timing constraints such as deadlines. In such a system, transaction processing must satisfy not only the database consistency constraints, but also the timing constraints. Real-time database systems can be found, for instance, in program trading in the stock market, radar tracking systems, battle management systems, and computer

integrated manufacturing systems. Some of these systems (such as program trading in the stock market) are soft real-time systems. These systems are designated *soft* because missing a deadline is not catastrophic. Usually, research into algorithms and protocols for such systems explicitly address deadlines and make a best effort at meeting deadlines. In soft real-time systems there are no guarantees that specific tasks will make their deadlines. This is in contrast to *hard* real-time systems (such as controlling a nuclear power plant) where missing some deadlines may result in catastrophic consequences. In hard real-time systems *a priori* guarantees are required for critical tasks (or transactions).

Most current real-time database work deals with soft real-time systems. In this work, the need for an integrated approach that includes time constrained protocols for concurrency control, conflict resolution, CPU and I/O scheduling, transaction restart and wakeup, deadlock resolution, buffer management, and commit processing has been identified. Many protocols based on locking, optimistic, and timestamped concurrency control have been developed and evaluated in testbed or simulation environments. In most cases the optimistic approaches seem to work best.

Most hard real-time database systems are main memory databases of small size, with predefined transactions, and hand crafted for efficient performance. The metrics for hard real-time database systems are different than for soft real-time databases. For example, in a typical database system a transaction is a sequence of operations performed on a database. Normally, consistency (serializability), atomicity and permanence are properties supported by the transaction mechanism. Transaction throughput and response time are the usual metrics. In a soft real-time database, transactions have similar properties, but, in addition, have soft real-time constraints. Metrics include response time and throughput, but also include percentage of transactions which meet their deadlines, or a weighted value function which reflects the value imparted by a transaction completing on time. On the other hand, in a hard real–time database, not all transactions have serializability, atomicity, and permanence properties. These requirements need to be supported only in certain situations. For example, hard real-time systems are characterized by their close interactions with the environment that they control. This is especially true for subsystems that receive sensory information or that control actuators. Processing involved in these subsystems are such that it is typically not possible to *rollback* a previous interaction with the environment. While the notion of consistency is relevant here (for example, the interactions of a real-time task with the environment should be consistent with each other), traditional approaches to achieving consistency, involving waits, rollbacks, and aborts are not directly applicable. Instead, compensating transactions may have to be invoked to nullify the effects of previously committed transactions. Also, another transaction property, namely *permanence*, is of limited applicability in this context. This is because real-time data, such as those arriving from sensors, have limited *lifetimes* – they become obsolete after

a certain point in time. Data received from the environment by the lower levels of a real-time system undergoes a series of processing steps (e.g., filtering, integration, and correlation). We expect the traditional transaction properties to be less relevant at the lowest levels and become more relevant at higher levels in the system.

While the hard real-time system should guarantee all critical transaction deadlines and strive to meet all other transaction deadlines, this is not always possible. In this case it is necessary to meet the deadlines of the more important transactions. Hence, metrics such as maximizing the value imparted by completed transactions and maximizing the percentage of transactions that complete by their deadline are primary metrics. Throughput and response time are secondary metrics, if they are used at all. A new trend is the use of active database technology for real-time databases, but this is still in the research stage.

4.2 Real-Time Artificial Intelligence

Many complex real-time applications now require or will require knowledge-based on-line assistance operating in real-time [5, 6]. This necessitates a major change to some of the paradigms and implementations previously used by AI researchers. For example, AI systems must be made to run much faster (a necessary but not sufficient condition), allow preemption to reduce latency for responding to new stimuli, attain predictable memory management via incremental garbage collection or by explicit management of memory, include deadlines and other timing constraints in search techniques, develop anytime algorithms (algorithms where a non-optimal solution is available at any point in time), develop time driven inferencing, and develop time driven planning and scheduling. Rules and constraints may also have to be imposed on the design, models, and languages used in order to facilitate predictability, e.g., limit recursion and backtracking to some fixed bound. Coming to grips with what predictability means in such applications is very important.

In addition to these changes within AI, real-time AI (RTAI) techniques must be interfaced with lower level real-time systems technology to produce a functioning, reliable, and carefully analyzable system. Should the higher level RTAI techniques ignore the system level, or treat it as a black box with *general* characteristics, or be developed in an integrated fashion with it so as to best build these complex systems? What is the correct interface between these two (to this point in time) separate systems. Integrating RTAI and low level real-time systems software is quite a challenge because these RTAI applications are operating in non-deterministic environments, there is missing or noisy information, some of the control laws are heuristic at best, objectives may change dynamically, partial solutions are sometimes acceptable so that a tradeoff between the quality of the solution and the time needed to derive it can be made, the amount of processing is significant and highly data dependent, and the execution time

of tasks may be difficult to determine. These sets of demanding requirements will drive real-time research for many years to come.

Not only is it important to develop real-time AI techniques, but it is also necessary to determine what must change at the low levels to provide adequate support for the higher level, more application oriented tasks? Answers to these questions have been somewhat arbitrarily chosen on an application by application basis. It is too early to synthesize the general principles and ideas from these experiments as sufficient evaluative data is not available.

Competing software architectures for real-time AI include production rule architectures, blackboard architectures, and a process trellis architecture. Some real-time AI systems have been built by carefully and severely restricting how production rules and blackboard systems are built and used. Research is on-going to relax the restrictions so that the power of these architectures can be utilized, but at the same time providing a high degree of predictability. The process trellis architecture, used in the medical domain, is a highly static approach while the other two are much more dynamic. The trellis architecture (because it is static) has potential to provide static real-time guarantees for those applications characterized by enough time to completely compute results from a set of inputs before the next set of inputs arrive. This approach is suitable for certain types of real-time AI monitoring systems, but its generality for complex real-time AI systems has not been demonstrated.

In a distributed setting, high level decision support requires organizing computations with networks of cooperative, semi-autonomous agents, each capable of sophisticated problem solving. Theories of communication and organizational structure for groups of cooperative problem solving agents must be developed. These theories must include problem solving under uncertainty and under timing constraints.

5 Summary

The scientific research community has identified many *grand challenge* problems whose solution requires high performance computing. This list includes scientific problems in earth and space sciences, vision and robotics, etc. These are, of course, very important problems. However, one important grand challenge that is missing is the need for developing the science and technology of real-time computing. So many large, complex future applications depend on real-time computing that it is critical to focus efforts on its development. Real-time computing is an enabling technology and without its scientific development, the cost and dependability of these systems will be unacceptable. The problems will not be solved by more and more computing power.

6 Further Information

- J. Stankovic and K. Ramamritham, *Hard Real-Time Systems*, Tutorial Text, IEEE Computer Society Press, Wash. D.C., 1988.
- J. Stankovic, Misconceptions About Real-Time Computing: A Serious Problem for Next Generation Systems, *IEEE Computer*, Vol. 21, No. 10, pp. 10-19, October 1988.

References

[1] K. Arvind, K. Ramamritham, and J. Stankovic, "A Local Area Network Architecture for Communication in Distributed Real-Time Systems," *Real-Time Systems*, Vol. 3, No. 2, May 1991.

[2] C. Hou and K. Shin, "Load Sharing with Considerations of Future Task Arrivals in Heterogeneous Distributed Real-Time Systems," *Proceedings IEEE Real-Time Systems Symposium*, December 1991.

[3] H. Kopetz, A. Damm, C. Koza, and M. Mulozzani, "Distributed Fault Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, pp. 25-40, 1989.

[4] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, pp. 58-68, May 1991.

[5] V. Lesser, J. Pavlin, and E. Durfee, "Approximate Processing in Real-Time Problem Solving," *AI Magazine*, Vol. 9, pp. 46-61, 1988.

[6] C. Paul, A. Acharya, B. Black, and J. Strosnider, "Reducing Problem-Solving Variance to Improve Predictability," *CACM*, Vol. 34, No. 8, August 1991.

[7] K. Ramamritham, J. Stankovic, and P. Shiah. "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 1, No. 2, pp. 184-194, April 1990.

[8] K. Ramamritham, J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, pp. 1110-1123, August 1989.

[9] J. Schoeffler, "Distributed Computer Systems for Industrial Process Control," *IEEE Computer*, Vol. 17, No. 2, pp. 11-18, February 1984.

[10] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175-1185, 1990.

[11] K. Shin, "HARTS: A Distributed Real-Time Architecture," *IEEE Computer*, Vol. 24, No. 5, May 1991.

[12] S. Shrivastava and A. Waterworth, "Using Objects and Actions to Provide Fault Tolerance in Distributed Real-Time Applications," *Proceedings IEEE Real-Time Systems Symposium*, pp. 276-287, December 1991.

[13] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-Time Systems*, Vol. 1, pp. 27-60, 1989.

[14] J. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next Generation Systems," *IEEE Computer*, Vol. 21, No. 10, October 1988.

[15] J. Stankovic and K. Ramamritham, "What is Predictability for Real-Time Systems," *Real-Time Systems Journal*, Vol. 2, pp. 247-254, December 1990.

[16] J. Stankovic, "On the Reflective Nature of the Spring Kernel," invited paper, *Proceedings Process Control Systems '91*, February 1991.

[17] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real–Time Systems," *IEEE Software*, Vol. 8, No. 3, pp. 62-72, May 1991.

[18] J. Stankovic, D. Niehaus, and K. Ramamritham, "SpringNet: A Scalable Architecture for High Performance, Predictable, and Distributed Real-Time Computing," submitted for publication, also Univ. of Massachusetts, TR 91-74, October 1991.

[19] M. Takegaki, H. Kanamaru, and M. Fujita, "The Diffusion Model Based Remapping for Distributed Real-Time System," *Proceedings IEEE Real-Time Systems Symposium*, December 1991.

[20] H. Tokuda and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM OS Review*, Vol. 23, No. 3, July 1989.

[21] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time MACH: Towards a Predictable Real-Time System," *Proc USENIX MACH Workshop*, October 1990.

[22] M. Wright, M. Green, G. Fiegl, and P. Cross, "An Expert System for Real-Time Control," *IEEE Software*, Vol. 3, No. 2, pp. 16-24, March 1986.

[23] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real–Time Systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 5, pp. 567-577, May 1987.