A Survey of Real-Time Operating Systems – Draft

Kaushik Ghosh (kaushik@cc.gatech.edu)*
Bodhisattwa Mukherjee (bodhi@cc.gatech.edu)
Karsten Schwan (schwan@cc.gatech.edu)

GIT–CC–93/18

February 15, 1994

Abstract

This paper describes current research in real time operating systems. Due to its importance to real-time systems, we begin this survey with a brief summary of relevant results in real-time scheduling and synchronization. Real-time operating systems are described in terms of the primitives and constructs offered to application programs. In addition, the effects of underlying computer architectures on real-time operating systems are discussed, followed by a description of benchmarks and evaluation methods for real-time systems.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

*Readers' comments and suggestions for improvement are solicited. Please direct them to `kaushik@cc.gatech.edu`.

1 Functionality and Characteristics of Real-time Systems

The embedded computer hardware of modern robots and industrial control systems is becoming increasingly complex. Typically, it consists of many interconnected computers operating at multiple levels of control or supervising different mechanical or electronic system peripherals. Given such hardware, the efficient execution of a real-time application requires that programmers deal with issues that arise for other high-performance, parallel and distributed application programs, such as efficient resource management, task and communication scheduling, load balancing, and programmed dynamic reconfiguration and recovery. Therefore, as with parallel and distributed operating systems, a real-time operating system must provide programmers with primitives for task control, interprocess communication, device operation, etc. However, the operating systems used in such settings must also offer support that is specific to the domain on 'real-time systems'. Such support is the focus of this survey paper.

This article surveys system principles and sample operating systems for real-time applications. We attempt to provide insights into current and future research topics in real-time systems, concentrating on research rather than industrial systems.

Early Real-time Systems. Early work on real-time operating or runtime systems focussed on the needs of single embedded systems[Sta92], such as flight control[Car84] or manufacturing systems[To73]. While most such work involved the construction of operating system kernels for specific target applications and architectural platforms[Cor90], two good summaries of the functionality typically offered by such operating systems are provided by current industry and government efforts to define a real-time operating system standard based on the Unix system in the U.S. (the POSIX 1003.4 standard[1] – also, see section 5.1), or based on the evolutionary TRON effort in Japan (see section 5.1). For example, the real-time functions in POSIX 1003.4 include:

- both synchronous and asynchronous I/O,
- IPC primitives (shared memory, semaphores and asynchronous events),
- the ability to lock and hold memory,
- priority scheduling,
- real-time files, and
- timers.

Other general characteristics of a real-time kernel [SBWT87, SR87, SR91, Ass92] include a size that may be adjusted to each application's needs, multitasking with low overhead for task context switches, fast response to external interrupts, limited or no use of virtual memory, support for time-constraints in tasks such as priority scheduling, support for real-time clocks, special alarms and time outs, and primitives to delay, pause, and resume tasks. Specific issues concerning the problems with making the Unix operating system suitable for real-time applications are described in [MHM$^+$90, FGG$^+$91, Sal93].

Predictability in Real-time Systems. The real-time functionality offered in POSIX, TRON, and elsewhere deals with with one of the main functions of a real-time operating system: to manage system resources in a timely manner. Thus, while traditional operating systems attempt to ensure

[1] POSIX 1003.1 standards are for the OS interface; POSIX 1003.4 for real-time extensions, and POSIX 1003.4a for threads extensions. While POSIX 1003.1 has been released, the other two are still in draft form.

the efficient and fair allocation of resources among multiple application programs, real-time operating systems must ensure that such allocation is performed in a timely fashion, and that it is performed such that existing application-level timing requirements can be met. Therefore, two basic requirements on real-time operating systems are: (1) high performance coupled with consistent execution times of real-time operating system functions, and (2) the provision of primitives with which application programs can run both efficiently as well as perform their operations in a timely and consistent fashion, thereby able to guarantee the applications' timing requirements. (1) and (2) are typically referred to as *predictability* in operating system or application program execution.

Predictability also implies that the metrics with which real-time systems are evaluated can differ from those used for non-real-time systems. For Unix implementations on workstation platforms, for example, one is concerned with the *average* performance of its OS primitives, whereas an important metric in a real-time version of Unix is that the performance of a primitive not degrade below a certain, acceptable threshold. At the same time, the desired system predictability is not easily measured and evaluated, in part because the timing requirements of application programs can differ widely – ranging from *guaranteed* or *hard deadlines* that must not be missed (otherwise causing catastrophic system failures), to *soft deadlines* that may be missed occasionally, to *recoverable deadlines* that cause programmed recovery actions when missed[GS93], to *weak deadlines*[LNL87, Loc86], which specify that partial or incomplete results are acceptable when the deadline is missed. Furthermore, spending more time on some computations can be useful when seeking 'better' results while at the same time causing possible timing violations. A rich body of work has investigated this relationship (see section 2).

Concerning the attainment and appropriate definition of predictability, real-time systems may be categorized using the following parameters[SR90]): (1) granularity of deadlines and task laxities, (2) strictness of deadlines, (3) the reliability requirements of the system, (4) system size and degree of interaction among system components, and (5) the characteristics of the system's operating environment. *Granularity* and *laxity* jointly determine the lengths of tasks, and the amount of time available for making scheduling decisions. For a system with low task granularities (which often coincides with low task laxities), fast, best effort [Loc86] scheduling decisions may be more suitable than slower, optimal decisions. *Strictness* of deadlines concerns timing semantics, such as hard vs. soft deadlines, and determines the utility of continuing to perform a task after its deadline has passed. Clearly, definitions of system predictability must be concerned with these semantics, and they will vary in accordance with the system specifications that must be met. In this context, system *reliability* may be defined as the system's ability to perform critical tasks within their timing constraints. The system's operating environment determines the degree to which systems must deal with dynamic effects, such as mode changes in response to drastic environmental changes (e.g., changing from walking to running mode in an autonomous robot[SBWT87]).

Complex Systems. While much of the earlier work in real-time operating systems focusses on single, embedded target architectures and applications, current computer systems and applications are addressing increasingly large and complex systems, including entire air traffic control systems[Chr93], single-or multi-site theater battle management systems[MS93a], multiple autonomous and cooperating robots[DJW93], distributed and real-time simulations[GFS93a]. and real-time communications in large-scale, distributed information systems. The complexity of such systems requires several essential and novel attributes from modern real-time systems:

- *Multiple task and communication granularities* – complex real-time applications consist of parallel application tasks of differing sizes, ranging from small tasks executed at high rates and by necessity consisting of a small number of instructions, to large tasks executed infrequently. Therefore, real-time operating systems must offer support for multiple task sizes, granularities of parallelism and associated communication latencies and throughput.

- *Varying timing semantics* – while many tasks are time-critical, their deadlines may vary in semantics and in laxity, including indications of task periodicity, recoverability, criticality for periodic tasks, and ranging from hard deadlines for certain tasks in embedded systems to 'desired timings' in those multi-media applications where high performance in network communications is more important than timeliness[Ten90, CT93]. As a result, while tasks must be executed within application-specific timing constraints, no single task scheduler is likely to satisfy all real-time applications. This constitutes one reason for operating system configurability, even at the lowest operating system levels.

- *Multiple task and communication models* – time constraints in task execution also imply time constraints in task communication[MST89, CW94], and equally important, any single model of task communication offered by a real-time operating system (e.g., RPC) is unlikely to be efficient and appropriate for all real-time applications. This is because tasks often make different assumptions regarding the model of communication used, where some tasks may tolerate the loss of individual readings from a sensor in order to perform an operation asynchronously at the highest rate possible, whereas other tasks may assume that individual messages never get lost[SBWT87]. As a result, real-time operating systems should support multiple models of time-critical task communication.

- *Configurability* – the current and future target architectures of real-time applications vary widely in size and complexity, ranging from the small embedded systems in the autonomous Mars Rover to future drill hole sensor processors of the power of a Thinking Machines CM-5. As a result, the machines' real-time operating system kernels must be highly configurable in size and functionality[Sak89a].

- *Adaptability* – complex systems can experience a large number of possible changes in their external operating conditions. As a result, operating systems and applications must be written to deal with such uncertainty, including permitting systems and applications to *adapt*[SR84, KM85] (i.e., change at runtime) in performance[SBB87] and functionality to external changes. Current research differentiates between two types of adaptations, those that anticipate changes in the operating environment – termed *preventive adaptations* – and those that deal with unexpected changes – termed *reactive adaptations*. Preventive adaptations attempt to guarantee certain levels of performance or functionality in operating software by making assumptions regarding future system behavior based on past behavior[SBB87, SGB87, BS91a, BS88, GS89b]. *Reactive adaptations*, on the other hand, are performed in response to external or exceptional events like failures, temporary overloads, etc.[Be85].

- *Fault tolerance* – real-time and embedded systems are often used in critical missions. Several kinds of hardware and software fault-tolerance mechanisms are incorporated in such systems. With the increase in complexity in the other parts of future real-time systems, the fault-tolerance and testability aspects have to investigate increasingly larger design spaces. Fault-tolerance in real-time systems have to test not only the value domain, but also the time domain [KG94, KKG$^+$90, GS93].

The remainder of this paper surveys recent research in the area of real-time operating systems.

The survey focusses on the novel research performed during the last few years, rather than discussing commercially available systems. We first describe some results in real-time task scheduling, because this area is of critical importance to real-time operating systems. Section 3 next elaborates on synchronization in real-time systems. Section 4 describes the salient features of experimental real-time operating systems constructed during the last few years.

2 Real-Time Scheduling

Research on real-time scheduling has experienced a major shift during the last few years, from static (off-line) to dynamic (on-line) scheduling. Therefore, this section begins with only a brief review of well-known static scheduling methods, followed by a more extensive discussion of dynamic scheduling. Thorough reviews of research in real-time scheduling appear in [Law83, CSR88, SR93b]. In addition, Kopetz [Kop93a] provides a general introduction to distributed, real-time scheduling and Casavant and Kuhl [CK88] provide a taxonomy of general (not real-time) scheduling approaches in distributed computing.

All work presented below uses the same measures of 'success' or 'quality' for real-time scheduling algorithms: (1) algorithm complexity (worst case running time), (2) an algorithm's ability to meet the deadlines of a given set of tasks to be scheduled (in the static case), or (3) an algorithm's ability to perform as well as any other available algorithm regarding the deadlines being met (in the dynamic case).

2.1 Well-known Scheduling Algorithms

Algorithms. Early algorithm work focusses on relatively small-scale or static real-time systems, where task execution times can be estimated prior to task execution (i.e., data dependencies are limited), and where the resulting task schedules can be determined off-line. Algorithms address both periodic tasks and sporadic tasks; periodic tasks typically arise from sensor data and control loops, and sporadic tasks can arise from unexpected events caused by the environment or by operator actions. A scheduling algorithm jointly schedules all periodic and sporadic tasks such that their timing requirements are met.

The most commonly used static methods are **cyclic schedulers** and more recently, **Rate Monotonic (RM)** scheduling algorithms[Lei80], in part because they are easily mapped to low-level priority-based task schedulers. The basic idea of the rate monotonic algorithm is to assign different and fixed priorities to tasks with different execution rates, highest priority being assigned to the highest frequency tasks, lowest priority to the lowest frequency task. At any time, the low-level scheduler simply chooses to execute the highest priority task. By specifying the period and maximum computation time of each task, the behavior of the system can be categorized a priori[SLR86].

RM algorithms (and static priority algorithms) can schedule a set of tasks to meet their deadlines if total resource utilization is lower than a certain *schedulable bound*[2] [LL73]. This bound may be less than 100% utilization. It is 0.693 for RM algorithms in general (with task set size approaching infinity), 0.88 when the periods are uniformly distributed [LSD89], and 1.0 only when the periods are harmonics of the smallest period. The scheduling of aperiodic tasks when using RM algorithms is addressed in [SLS88, SSL89]. The *Extended Priority Exchange* algorithm described in [SLS88] utilizes unused time allotted to periodics for better aperiodic response, and the *Sporadic Server* algorithm creates a 'server'

[2] The schedulable bound of a task set is defined as the maximum CPU utilization for which the set of tasks can be guaranteed to meet their deadlines.

task with a given period and utilization. All soft-deadline aperiodics use a single sporadic server, while each hard-deadline aperiodic uses a distinct server. This approach will meet the system goal of keeping the deadlines of periodics and hard-deadline aperiodics, while minimizing the response time for soft-deadline aperiodics.

In addition to schedulable bounds that are less than 1.0, two problems exist for RM algorithms: (1) RM algorithms provide no support for dynamically changing task periods and/or priorities, and (2) tasks may experience priority inversion. The first problem is addressed in [HKL91], where the authors consider fixed priority scheduling of periodic tasks with varying task execution priorities. Specifically, tasks may have subtasks of various priorities. The method presented for determining task schedulability involves identifying a critical instant[3] for the periodic task by analyzing its worst-case phasing with respect to other tasks, the priority of each task with respect to the priorities of the subtasks of the other tasks (to find the number of subtasks whose schedulability must be checked), and to check for completion of each such subtask. *Priority inversion* arises when a high priority job must wait for a lower priority job to execute, typically due to other resources being used by executing tasks (e.g., tasks waiting on critical sections). In [MT92], the authors consider the nature of the non-preemptable critical regions that give rise to such priority inversions in the context of a soft real-time operating system, where average response time for different priority classes is the primary performance metric. An analytical model is used to illustrate how non-preemptable critical regions may affect the time-constrained jobs in a multi-media (soft real-time) task set. Chen et. al.[CL91] study a priority ceiling protocol for multiple-instance resources, and present an optimal resource allocation algorithm which can be used to improve the schedulability of a real-time system.

Earliest Deadline First (EDF) scheduling algorithms can be used for both dynamic and static real-time scheduling[ZRS87b, DM89, CC89, SZ92]. These algorithms use the deadline of a task as its priority. Since the task with the earliest deadline has the highest priority, the resulting priorities are naturally dynamic and the periods of tasks (represented by their deadlines) can be changed at any time. A variant of EDF scheduling is **Minimum-Laxity-First (MLF)** scheduling [DM89], where a laxity is assigned to each task in the system, and minimum laxity tasks are executed first. Laxity measures the amount of time remaining before a task's deadline will pass if the task uses its allotted maximum execution time. Essentially, laxity is a measure of the flexibility available for scheduling a task. The main difference between MLF and EDF is that unlike EDF, MLF takes into consideration the execution time of a task[SK91].

While EDF is superior to RM in the sense that its schedulable bound is 100% for all task sets, a problem with EDF is that there is no way to guarantee which tasks will fail in transient overload situations. This has resulted in another variant of EDF scheduling, called the **Maximum-Urgency-First (MUF)** algorithm[SK91], where each task is given an explicit description of urgency. This *urgency* is defined as a combination of two fixed priorities, and a dynamic priority, which is inversely proportional to the task's laxity. One of the fixed priorities, called task criticality, has precedence over the task's dynamic priority. The other fixed priority, called user priority, has lower precedence than the task's dynamic priority. The idea is to use user-specified notions of 'priority' (i.e., criticality) to help on-line algorithms distinguish more important from less important tasks.

EDF algorithms have also been extended to deal with resources other than the CPU. In [Jef92] an optimal algorithm is presented for scheduling a set of sporadic tasks that share a set of serially reusable, single unit resources such that tasks complete their execution before a deadline, and such that resources are accessed sequentially. The algorithm combines EDF scheduling with a synchronization scheme for access to shared resources. More information on real-time synchronization appears in section 3.

[3] A Critical Instant for a task is an instant such that if the task is activated at that instant, its completion time will be the longest.

Effects of Cycle-Stealing and Overloads on Scheduling Algorithms. Additional effects of non-CPU resources on scheduling are discussed in [RSL87], where the authors investigate the effects of cycle stealing on scheduling algorithms in a hard real-time environment. Specifically, since an I/O device can transfer data by direct memory access (DMA) and therefore, steal cycles from the processor and the currently running task, such cycle-stealing can cause unpredictable delays and lead to missed task deadlines (even at low degrees of processor utilization). In addition, I/O devices are often designed such that FIFO is the only possible way to schedule I/O activities. As a result, the benefits of 'intelligent' real-time CPU scheduling may be negated by inappropriate I/O scheduling[SLR86]. This issue is addressed in [RSL87] by proposing a remedy consisting of two steps: (1) I/O devices should be constructed such that they can be scheduled in the same fashion as the CPU device, (2) the degree of memory interleaving necessary to effectively counter the effects of cycle stealing is determined. For instance, when both I/O in the DMA controller and computation tasks in the processor are schedulable, the provision of only 4 banks of low-bit interleaved memory can lead to significant improvements in performance. With 8 banks near-optimal performance is obtained.

Theoretical schedulability results under conditions of system overload are discussed in [KS92], where the authors present an optimal on-line scheduling algorithm able to operate in overloaded systems. Further, in [KSH93], the authors derive an inherent bound on the best competitive guarantee that a multiprocessor on-line scheduler can afford, and present an algorithm that achieves a worst-case guarantee within a small factor of this bound. In addition, [KM92] discusses the semantics of data-intensive real-time applications. Specifically, by examining the semantics of these applications, the concept of similarity is formed, which has been used on an ad hoc basis by application engineers to provide more flexibility in concurrency control. An efficient real-time scheduling algorithm exploiting similarity is proposed. In [BKM+92], the authors show that no on-line scheduling algorithm can guarantee a performance better than 1/4th of a clairvoyant scheduler[4], and present a uniprocessor algorithm that performs to this 1/4th factor of the clairvoyant scheduler. Generalizing to a dual-processor case, the authors prove that in the dual-processor case, no on-line algorithm can perform better than 1/2 of the clairvoyant scheduler.

Evaluation and Algorithm Improvements. Researchers have extensively studied both preemptive [LL73, MD78, LM80, Law81, Mok83, RS84, SLR86, RSZ89, ZS91] and non-preemptive [Lei80, LY82, ZRS87a, RSS90] real-time scheduling algorithms. In [LL73], the authors show that the rate-monotonic and earliest deadline scheduling algorithms are optimal static priority and dynamic priority scheduling algorithms, respectively, in a uni-processor preemptive scheduling environment. In [Mok83], the author proves that the slack-time algorithm is optimal, as well.

In [MD78], an optimal runtime scheduler for hard real-time environments is defined as one that is able to meet all task deadlines, provided that such an algorithm exists. With this definition, both earliest due date (EDD) and least laxity first (LLF) sequencing of arrivals result in optimal run-time schedulers. It is also shown that an optimal scheduler cannot be found for multiprocessors unless a priori knowledge exists of the deadlines, computation times and arrival times of all tasks. In [GJ77], the authors show that even for a single processor, constructing an optimal schedule for tasks with arbitrary arrival and computation times and arbitrary laxity is an NP-complete problem.

The performance study in [JLT85] of various classical scheduling algorithms also considers situations where computation times are not exactly known at the time of task arrival, but have some given, known distributions. The study introduces the notion of a *value function* which specifies the value of completing a task at any time after arrival, thereby attempting to unify the treatment of both hard and soft real-time environments. The idea is that tasks with value functions that become negative

[4] This is done by quantifying the benefit of clairvoyance, and assigning a scheduler a 'value' equal to the task's execution time if it can schedule a task to completion.

while waiting or during execution are aborted or discarded from the task queue. The best algorithms in terms of maximizing the value of completed tasks for multiprocessor take into account the expected value of a task at completion. This value is the probability that a task completes prior to its critical time or deadline. The study does not address how suitable value functions may be found for practical scheduling problems.

Imprecise Computations Recent work in real-time scheduling has begun to consider changes in the semantics of timing constraints to be used and enforced in actual systems. Several specific formulations have been advanced by the 'imprecise computation' community and more recently, by work in 'anytime' algorithms being performed by AI researchers.

The basic idea in imprecise computation is that there are some algorithms (e.g., iterative algorithms) that can return results at almost any time during their execution. The longer they run, the more precise their results. Ideally, a process executes until a result with a desirably small tolerance has been obtained. However, when time is limited, the process can be terminated prematurely, producing a result that may be acceptable but not as precise as desired.

In [LLN87], the authors discuss a formulation of and algorithms for this problem which take into the account the quality of the overall result. In [SL92], the authors describe three algorithms for scheduling preemptive, imprecise tasks on a processor such that total error is minimized. Each *imprecise* task consists of a *mandatory* followed by an *optional* portion, and some of the tasks may arrive after the processor begins execution. The algorithms assume that when each new on-line task arrives, its mandatory portion and all mandatory portions of all tasks yet to be completed at the time can be feasibly scheduled before their deadlines. The algorithms produce for such tasks feasible schedules whose total errors are minimized. Three algorithms are presented for three types of task systems: (1) when every task is on-line and is ready upon its arrival, (2) when every on-line task is ready upon arrival but there are also off-line tasks with arbitrary ready times, and (3) when on-line tasks have arbitrary ready times.

Recent research in anytime algorithms[DB88], applied to real-time robotics, does not assume mandatory task portions. As with other research in autonomous robotics, researchers instead assume the existence of alternative tasks or task sets (e.g., exception handling tasks) to be used when the deadlines of the original tasks cannot be met[BS91a, GS93].

2.2 Schedulers for Multiprocessor and Distributed Real-Time Systems

Multiprocessor Scheduling. Since embedded system architects are increasingly turning to multiprocessor computer architectures (for reasons of reliability and/or performance), scheduling algorithms and scheduler designs have begun to address parallel execution platforms, typically assuming dynamic task arrivals and on-line scheduling for both sporadic and periodic tasks. One efficient on-line multiprocessor algorithm is the any fit algorithm proposed by Blake and Schwan[BS91b], which offers a distributed scheduler implementation consisting of global schedulers performing the assignment of tasks to processors in cooperation with processor-local schedulers that carry out deadline scheduling. This work is extended further in [ZSA91]. The resulting multiprocessor schedulers have some similarities to those developed for distributed systems and described in [RS84, RSZ89].

The quality of on-line scheduling is determined not only by the algorithms implemented by schedulers, but also by the efficiency of the parallel scheduler (and its data structures) implementing those algorithms. In [ZSA91], the authors address two problems regarding the performance of multiprocessor schedulers: (1) how are the latency and the quality of scheduling affected by different degrees of completeness in the information shared among multiple, potentially concurrent schedulers, and (2) how can scheduling information be represented so that it is efficiently and concurrently accessible? The authors present a real-time scheduling algorithm for multiprocessors that is scalable in the number of tasks performing scheduling and in the maximum amount of computation time consumed by those tasks. In addition, a flexible representation for shared information within the distributed scheduler facilitates variations in the degree of shared information completeness and consistency. It is shown that the sharing of incomplete (vs. complete) information can lead to increased performance regarding scheduling latency with few or no losses in scheduling quality.

A second topic receiving increased recent attention due to complex systems' needs for on-line scheduling is the construction of special purpose hardware for execution of schedulers or for assistance in memory management in real-time applications. Such work began by simply dedicating a single processor in a multiprocessor system to execute the scheduler and contain scheduling information[SR91], but recent work concerns hardware support for scheduling or scheduling co-processors and similar support for dynamic memory management. This interrelationship between architecture and real-time operating system is discussed further in section 4.6.

The remainder of this section presents some detail on distributed real-time scheduling, emphasizing scheduler performance and structures over scheduling algorithms.

Distributed Real-time Scheduling. An important topic in real-time scheduling is how to enforce globally important performance properties in distributed systems. Toward this end, most distributed scheduling algorithms have two common features[WC87]: (1) a *global* task sharing strategy between nodes and (2) a *local* scheduling policy for individual nodes. The local scheduling policy is often based on heuristics that efficiently determine which tasks to accept or reject. Some distributed scheduling algorithms are compared in [CL86], based on the means utilized for sharing scheduling information (i.e., the global part). In [RS84], a heuristic, called a 'guarantee routine', is proposed for local scheduling in a distributed system. Here, an arriving task is inserted into the queue if it is possible to guarantee that both the arriving task and all other tasks currently in the queue do not miss their deadlines. Rejected tasks are then passed on to the task sharing algorithm for distribution to other processors.

Local scheduling algorithm performance for hard real-time distributed systems is determined not only by the rejection ratio but also be the number of tasks able to be shared. This in turn depends on the rejected task having sufficient laxity remaining at the time of rejection to enable it to be sent to other processors. For tasks with hard real-time deadlines, results are limited to FCFS service. The earliest work in this area appears in [GK68], where analytic results are presented for the ratio of rejected tasks arriving to a multiprocessor system, assuming Poisson arrivals and tasks with known exponential computation and laxity requirements. In [BBH84], the authors present transform solution results for tasks with arbitrary computation, laxity and arrival distributions, where FCFS service is used and all task parameters are known on arrival to the queue. In [KSC86], analytic results are extended for an elementary load sharing algorithm with tasks of fixed laxity and a fixed delay in load sharing, again assuming a FCFS local scheduling algorithm. In [WC87], the commonly used local non-preemptive algorithms are examined, and their performance is compared with regard to rejection ratio and expected task laxity at rejection. The policies compared are the standard sequencing methods of FCFS, SJF (sequencing by shortest computation time first), LLF, EDD, the local 'guarantee' routine (GM), and a run-time selection algorithm (called MM algorithm) based on the Moore ordering rule[Moo68]. The criteria considered for selection of a local scheduling algorithm for hard real-time systems is that of minimum rejection ratio, maximum number of rejected tasks with positive laxity and greatest task laxity at rejection for tasks with positive laxity. Simulation results show that MM algorithms exhibit the best performance for task rejection for a given example of task computation and laxity. For the other criteria, the FCFS algorithm results in the largest number of rejected tasks with positive laxity, while the LLF algorithm results in the greatest laxity at rejection. CPU utilization appears to be similar for the LLF, EDD, GM and MM algorithms and for the FCFS and SJF algorithms.

In [HS91b], Hou, Shin, et. al. propose a load sharing algorithm for real-time applications which takes into account the effect of future task arrivals on locating the best receiver for each un-guaranteed task in a heterogeneous distributed environment. This work is extended in [HS92], which addresses the problem of allocating (assigning and scheduling) periodic task modules to processing nodes (PNs) in distributed real-time systems subject to task precedence and timing constraints. They propose a module allocation algorithm (MAA) to find an 'optimal' allocation that maximizes the probability of meeting task deadlines using the branch-and-bound technique. The task system within a planning cycle is first modeled with a task flow graph (TG) which describes computation and communication modules as well as the precedence constraints among them. To incorporate both timing and logical correctness into module allocation, the probability of meeting task deadlines is used as the objective function. The MAA is then applied to find an optimal allocation of task modules in a distributed system. The timing aspects embedded in the objective function drive the MAA not only to assign task modules to PNs, but also to use a module scheduling algorithm (MSA) (with polynomial time complexity) for scheduling all modules assigned to each PN so that all tasks may be completed in time. Several numerical examples are presented to demonstrate the effectiveness and practicality of the proposed algorithms.

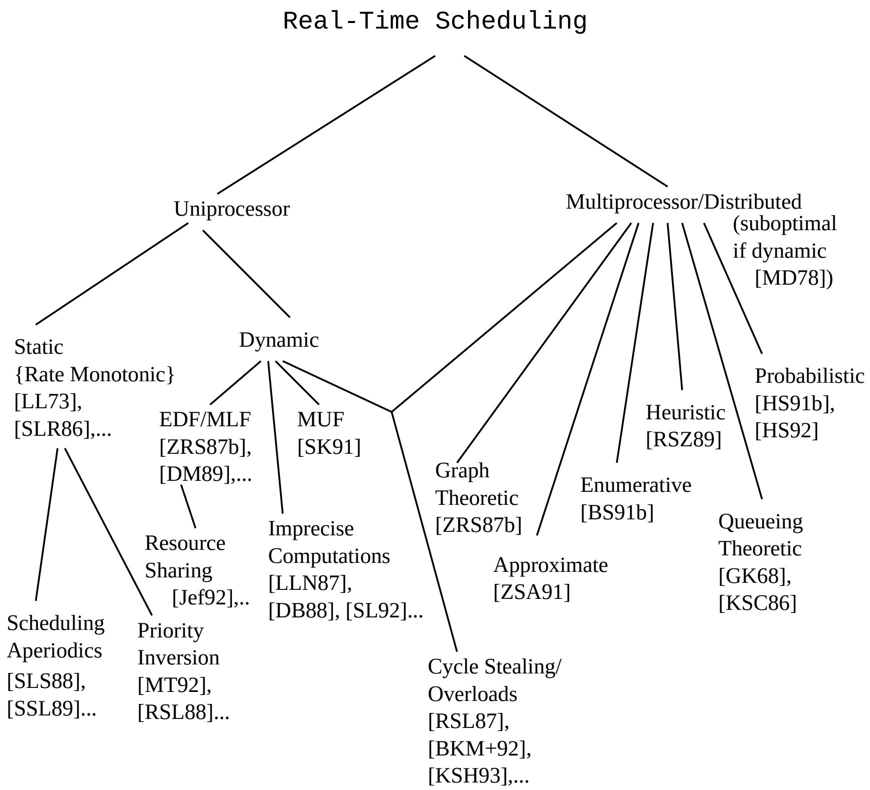

Figure 1: A simple taxonomy of some of the scheduling algorithms discussed here.

Figure 1 presents a classification of the some of the scheduling literature presented in this survey. Scheduling approaches can be roughly divided into uniprocessor and multiprocessor scheduling algorithms, with multiprocessor approaches often being used in distributed real-time scheduling (after proper consideration of communication scheduling). The Rate-Monotonic approach was the only static uniprocessor algorithm we discussed. Among the dynamic approaches, we mentioned Earliest-Deadline-First, and its variants: Minimum Laxity First and Maximum Urgency First. We also discussed work on imprecise computation/Anytime algorithms in the uniprocessor domain. In multiprocessor scheduling, we mentioned some approaches to deal with overloads (some of these approaches are also applicable in uniprocessor systems), and then discussed some strategies of distributed real-time scheduling, and the structure of distributed schedulers. Dynamic multiprocessor real-time schedulers are by nature sub-optimal. The algorithms cited here may be roughly classified into graph-theoretic, queueing-theoretic, enumerative, approximate, probabilistic and heuristic approaches. We provide examples of each approach in figure 1.

2.3 Future Work

Dynamic Systems. Current and future work in real-time scheduling must address the highly dynamic, complex environments of current and future large-scale real-time systems, such as national networks carrying time-constrained communications (e.g., multi-media applications like real-time video compression and transmission[Wal91, Gal91, TTCM92]), or collaboration systems[Gre88, BHI93, FKRR93], or theater battle management systems[MS93a], or distributed real-time simulations[GFS93a]. Many open issues in the area of real-time scheduling are due to the novel characteristics such future real-time applications, including the presence of (1) multiple task and communication granularities associated with, (2) multiple task and communication models, (3) varying timing semantics used in application programs, (4) increased system configurability, and (5) dynamic system adaptability to different operating environments and user requests. As a result, researchers are now primarily addressing on-line scheduling, scheduling for parallel and distributed systems, and the semantics of timing constraints to be enforced in such future systems (e.g., hard deadlines are simply neither needed nor feasible as a formulation of timing constraints in multi-media applications[Bum93, MB93]).

On-line Scheduling. Researchers are also addressing the actual overheads experienced by on-line scheduling algorithms[ZSG92, SZ92], and even performing the actual scheduling in hardware [NRS$^+$93]. This is resulting in more attention being paid to scheduling algorithm and scheduler implementation rather than considerations of algorithm optimality and complexity (the latter only captures worst case performance, whereas we are often interested in average case performance in actual systems). In addition, new topics like reliability coupled with timeliness must be explored for large-scale systems. The topic of soft deadlines, and alternative notions for deadlines also needs further investigation.

Scheduler Implementation. Another important future topic of realistic real-time scheduling is the issue of scheduler integration and configuration in operating systems offering multi-user support and the required operating system protection facilities. While some work has been done on user/operating system interfaces for threads-based schedulers in non-real-time systems[ABLL92, PM93], only recent research is addressing how user-level requirements may be shared with or affect the scheduler or scheduling algorithms integrated into a real-time operating system. Typically, such research is assuming object-oriented user/operating system interfaces, where additional scheduling information is shared as attributes passed at the time of thread creation or thread scheduling for dynamic schedulability analysis[ZS91, Jon93a], or where such attributes are passed earlier for static scheduling. In addition, operating system or more specifically, scheduler configurability may concern the definition of additional 'configuration methods' on scheduler objects[PM93] or 'configuration attributes' submitted with object invocations and processed by 'policies' associated with operating system objects[GS89a].

Research is also beginning to address the structuring of flexible operating system schedulers for large-scale parallel or distributed machines. Such schedulers must make provisions for use of different scheduling algorithms, alternative representations of scheduling information (to attain various cost/performance and scalability tradeoffs), and permit the use of different timing semantics. Current work has attempted to define flexible frameworks within which alternative scheduler implementations and scheduling algorithms can be implemented. One such framework is described in [Bla89], where alternative scheduler configurations are defined for distributed memory machines based on notions similar to the fragmented or distributed objects discussed for high performance computing or distributed applications [SB90, Sha86]. An implementation of such a framework for a shared memory machine is described in [Bla89] and is now underway for modern shared memory multiprocessors[ZSA91]. The object-based Chaos operating system also offers users the ability to implement and use alternative low-level real-time threads schedulers[GS93]. Furthermore, Stankovic and Ramamritham [SR93a] have identified a set of problems that any comprehensive scheduling approach should be able to handle. These include dealing with both preemptable and non-preemptable tasks, periodics and non-periodics, multiple task criticalities, groups of tasks with a single shared deadline (co-scheduling or gang scheduling), precedence constraints, resource and communication requirements, task migration and placement, fault tolerance requirements, soft and hard deadlines, and normal and overload conditions.

In section 4, we discuss salient features of several well-known real-time operating system kernels, which should illustrate how some current systems address the issues raised in this section.

3 Synchronization in Real-time Systems

Synchronization is important in real-time systems for two reasons: (1) tasks may experience unpredictable delays due to blocking on shared resources to which they require exclusive access, and (2) solutions attained for synchronization may also help in constructing solutions for the multi-resource task scheduling important in several future real-time applications. Examples of applications in which multi-resource scheduling must be performed are multi-media applications in which video streams and associated output devices must be scheduled in conjunction with each other, since users would like synchronized voice and video[AC86], and military applications in which CPU processing must be synchronized with sensor input processing[Fer93].

Theoretically, Mok[Mok83] showed that the addition of mutual exclusion requirements in real-time programs makes the general scheduling problem an NP-hard problem. In practice, a number of algorithms have been devised and evaluated. Several algorithms are presented next.

Uniprocessor Systems. For uniprocessor systems running periodic tasks, two recent protocols provide effective solutions to the scheduling problem with resource sharing. They are the *kernelized monitor* protocol[Mok83] and the *priority ceiling* protocol[SRL90]. In the kernelized monitor protocol, the *earliest deadline first* scheduling policy is used for task scheduling. All executions in critical sections are non-preemptable. However, schedulability analysis performed in this protocol requires the use of upper bounds on the execution times of all critical sections appearing in tasks. Since such upper bounds may be overly pessimistic, using the kernelized monitor protocol may result in low processor utilization.

The priority ceiling protocol is designed for systems where each task has a fixed priority and the *rate monotonic* scheduling algorithm is used. With this protocol in the worst case, each task only has to wait for at most one lower priority task to finish in a critical section, and deadlocks cannot occur. Assuming that the longest possible waiting time is known for each task in the system, sufficient conditions for scheduling sets of periodic tasks can also be derived[SRL90]. However, the priority ceiling protocol cannot be directly used when priorities are dynamic, which is addressed in the protocol described in [CL90].

For uniprocessor systems, Jeffay[Jef89b, Jef93] develops schedulability conditions for a set of sporadic tasks that each consist of a sequence of *phases*, where at most one shared resource can be accessed in each phase. In his analysis, tasks' timing constraints as well as resource requirements are assumed known a priori. It is shown that optimal synchronization and scheduling disciplines exist for restricted patterns of resource usage.

Multiprocessor Systems. Predictable synchronization on multiprocessor real-time systems offers a new challenge compared to existing work on uniprocessors. In [MSZ90], predictable algorithms are described for semaphores with linear waiting. Although the proposed algorithms are predictable, they do not take into account the priorities of the processes that wish to acquire the semaphore. In [RSL88], the authors present a multiprocessor extension of the priority ceiling protocol [RSL89]. The priority ceiling protocol minimizes priority inversion for a set of periodic real-time processes that access exclusively some shared data. The multiprocessor priority ceiling protocol generalizes the uniprocessor protocol by executing all critical regions associated with a semaphore on a particular processor called the synchronization processor. As a result, the critical regions in a program are replaced by an invocation to a remote synchronization server. Unfortunately, the description of a single centralized synchronization server limits the scalability of the solution, and may increase the cost of executing fine grain real-time applications.

Mutual exclusion and synchronization for *dynamic* hard real-time multiprocessor applications are discussed in [ZSG92]. As with any dynamic parallel program, a dynamic real-time application's execution can result in the on-line creation of additional tasks, and the creation of such time-constrained tasks cannot be predicted or accounted for prior to program execution. The research results presented in the paper concern task synchronization such that guarantees can be made regarding the synchronized tasks' timing constraints. Such guarantees cannot be made without performing on-line schedulability analysis and on-line analysis concerning the maximum time that a task will wait for some resource being acquired with a synchronization primitive. A novel real-time locking scheme is shown to prevent deadlocks and ensure time-bounded mutual exclusion. The maximum waiting time for a task attempting to acquire a resource is computed with an $O(1)$ algorithm. Two attributes of the algorithm are: (1) previously made guarantees regarding resource accesses are always maintained, and (2) failures regarding accesses to shared resources are reported immediately. As a result, the application program or higher-level operating system software can deal with such failures in a timely manner, by acquisition of alternative resources, by execution of exception handling code, etc.

Research on synchronization in non-real-time systems derives efficient spin-lock implementations on multiprocessor systems [And90, MCS91]. Since these implementations service lock requests in FIFO order, they may be usefully employed in real time systems which just want to bound (not minimize) priority inversion. Mechanisms for constructing locks that may be configured for non-real-time use are described in [MS93c]. In [Mar91], Markatos enhances existing algorithms by Burns[Bur78] and Mellor-Crummey and Scott[MCS91] in order to define a priority spin lock with implementations that involve only local spinning. A priority spin lock has a priority ordering property. Each processor competing for a priority spin lock has a unique dynamic priority that reflects the importance of the process it runs. The processes that request, hold or release such locks are not pre-emptable during lock operations.

4 Real-Time Kernels and Run Time Systems

Research on real-time operating systems in the U.S. has been driven by three primary concerns: (1) support of the Ada language (section 4.1), (2) the efficient and predictable execution of embedded systems (section 4.2) and (3) dealing with the complexity of large-scale and dynamic real-time applications. Furthermore, recent research is becoming more concerned with the provision of some platform on which diverse real-time systems may be constructed (e.g., real-time, configurable threads and micro-kernels). Commercial systems, on the other hand, have typically provided either Ada support (e.g., environments supported by Honeywell[MS82] or TRW[MS93a]), or some fixed set of primitives (i.e., micro-kernels) at the process level (e.g., pSOS), or they have focussed on building real-time extensions to or alterations of Unix, the latter now resulting in the POSIX real-time standards for Unix. A real-time version of the Mach operating system is also being developed (see section 4.3.3).

4.1 Ada-supporting Runtime Systems

A continuing, DoD-induced thrust in current research on real-time systems is to design and build run-time support for real-time Ada.

A brief description of some of the Ada 9X proposals addressing hard real-time requirements appears in [BP91]. Ada 9X is a revision to the Ada programming language standard[Wel92]. The reports [Inc92, QD92, Sof92] present some results of real-time implementation studies of Ada. In [BJ87, Bak87, BJ86], an interface to a real-time Ada run-time environment is presented. Different possible time representations and their utilization in real-time Ada systems are discussed in [Bri92].

Corset and Lace[BJ87, Bak87, BJ86] are runtime environment interfaces. Corset is an interface specification for a compact runtime support environment for Ada tasking. Lace is an interface specification for a low level adaptable common executive that implements a model of real-time, lightweight tasks. Compiled Ada tasks and programs request Corset and Lace services via normal Ada procedure calls.

Corset hides details of the runtime support environment (RSE) from the compiler. Lace, in turn, hides the details of processor allocation from the Ada RSE. This permits tailoring the dispatching policy to fit the application. In addition to information hiding, Lace also supports multiprogramming of simple Ada procedures without involvement of the Ada RSE, thereby eliminating unnecessary inefficiency and unpredictability. Such multiprogrammed procedures can be executed simultaneously by other tasks that make use of the full Ada RSE. Therefore, it is easy to construct hybrid systems. Execution timing remains under control of the Lace dispatcher. Lace does not provide directly for intertask communication or memory management. Such services are provided separately, possibly using the Lace operations. The Corset interface and the Lace interface are described in detail in [BJ87].

Additional work on distributed Ada runtime systems focusses on on-line support for program monitoring and load balancing at Honeywell[MS82] and language-based support for on-line adjustments to task scheduling at TRW[MS93a]. Both such efforts are attempting to address the future, dynamic execution environments presented by large-scale real-time applications like theater battle management, distributed simulation, or distributed interactive manufacturing control systems.

4.2 High-Performance, Predictable Real-time Programs

Basic attributes of many real-time operating systems or system kernels are (1) the provision of functionality for implementation of real-time requirements of application programs with (2) operating system constructs that are both efficient and offer consistent or predictable performance. Unfortunately, many optimization techniques that enhance performance in non-real-time systems reduce timing predictability and must therefore, be either eliminated or used sparingly in real-time operating systems [Rea90]. Examples include caching and virtual memory [NNS91], lazy evaluation and lazy copying[AT87], and FIFO-queueing, instruction-pipelining, and delays associated with locks[TNR90b]. When such techniques are used in real-time systems, system predictability must be explicitly preserved by separating time-critical from other application code, perhaps using the layering techniques suggested in [SR90]. The basic idea of this technique is that a 'higher layer' can be implemented to be predictable if all its lower layers are predictable. Specifically, once the worst case computation times of lower layers are known, worst case computation times can be computed for higher layers, as well.

The remainder of this section presents the design of a few well-known real-time kernels. It should serve to illustrate the techniques used to achieve efficient and predictable real-time execution in some of the well-known real-time systems today.

4.2.1 Predictability on RISCs: VxWorks and pSOS

While RISC architectures increase the *average* execution rates (by instruction pipelining, with processor caches, by using special on-board floating-point co-processors, using latency-hiding techniques in memory operations or large sets of registers, etc.[Can91]), they also introduce uncertainties in program instruction execution times. Incorrect branch-prediction may result in flushing of pipelines (RISC processors often have several pipelines, each with several stages). This leads to delays due to the instruction cache being refilled. Furthermore, register-conflicts may occur if identical resources are required by instructions close together in a pipeline [Tho90, Can91]. Similarly, while register windows speed up subroutine invocation, saving all of the windows at a context switch can be quite time-consuming. In addition, low-level details like instruction execution times (and hence, the worst-case execution time of a code segment) change with each implementation of an architecture. Some of these issues are addressed by the VxWorks and pSOS operating systems discussed next.

VxWorks. The VxWorks real-time operating system [FSW91, FSW90, Ing91, NEN+92] addresses context switch overheads by saving only those register windows (on a Sparc) that are actually in use (as determined by the *window invalid mask* (WIM) bit of the particular window on a Sparc) by a task being switched. When the task's context is restored, only a single register window must be restored (its contents are restored from memory, and its WIM bit is set). The other windows are restored later at the appropriate returns-from-subroutine (the window-underflow trap causes the contents to be restored). VxWorks also allows the register windows (typically there are several such windows in Sparc implementations [Can91]) to be used as 'register-caches', whereby a window is saved during a context switch only if loading the new context requires so. This has the advantage of eliminating the time for saving and restoring register windows if the 'outgoing task' runs again soon, and if the number of register windows used by each task is small. VxWorks also allows a set of register windows to be allocated to a task in 'dedicated' mode: these particular windows need not be flushed when the task is context-switched out. Since interrupt-processing is often a high priority task, VxWorks allows the application designer to reserve one or more register windows for this purpose. Under these circumstances, interrupts need not restore any register window to switch in the interrupt-handler's context, unless the level of nesting of interrupts gets too large.

pSOS. The pSOS operating system offers 'standard' building blocks for real-time operating systems (e.g., kernels, debuggers, monitors etc.) on several hardware platforms[Tho90]. It is intended as a 'plug-in operating system' that can be readily used by real-time software implementors. pSOS is position-independent binary in which 'system software' is accessed through traps, and the timing properties of all such system calls are defined and guaranteed by the system (as per the system's design specification).

pSOS^{+} is a real-time OS kernel component for embedded designs, and is a superset of pSOS. pSOS^{+} partitions system activity into any number of distinct tasks, which logically execute in parallel with each other. Tasks are scheduled using preemptive priority-based scheduling, with 255 priority levels. Priorities of tasks may be changed dynamically. Four inter-task communication primitives exist. A 'receiving task' may poll or block on a single *event flag* or a boolean combination of such flags. System wide *counting semaphores* are provided for mutual-exclusion. A timeout can be specified with a semaphore. Semaphores are dynamically created and deleted. Tasks waiting on a deleted semaphore are made ready, and an error condition is returned. pSOS^{+} also provides global, named *message queues*. Each packet can be up to four longwords large. Blocking and non-blocking versions of receive are provided, and a timeout can be specified with a blocking receive. Finally, tasks can interact using Unix-like signals.

pSOS^{+} supports both tightly-coupled and loosely-coupled multiprocessors in its design. The functionality of the multiprocessing version of pSOS^{+}, called pSOS^{+m}, is divided into two layers: the *Kernel Interface* (KI) layer handles all inter-processor communication generated by the pSOS^{+m} kernel, while the pSOS^{+m} layer handles task activities, intertask communication and interrupt service routines within a single processor. Two types of KIs are currently supported: one based on shared-memory communication between processors, and the other based on TCP/IP.

4.2.2 GEM: High Performance for Parallel Real-time Applications

The Generalized Executive for real-time Multiprocessor applications (GEM) addressed several requirements of real-time software[SBWT87, Sch88]. It was constructed for an embedded target architecture consisting of multiple, networked multiprocessor, and it was evaluated with the control software of a semi-autonomous robot walker, the ASV walking machine[MI79]. The primary contributions of the GEM system are: (1) when using GEM, programmers can select one of two different types of tasks differing in size, called processes and micro-processes, the latter being a precursor of modern notions of 'threads' of execution, (2) the scheduling calls offered by GEM permit the implementation of several models of task interaction, (3) GEM supports multiple models of communication with a parameterized communication mechanism, representing functionality routinely offered by modern real-time operating systems, and (4) GEM was closely coupled to a prototype real-time programming environment that provided programming and scheduling support for the models of computation offered by the operating system. In addition, GEM initiated research addressing highly dynamic real-time applications, by offering simple mechanisms for on-line program adaptation in response to variations in hardware configuration and application performance or reliability requirements. Such research is described in detail in section 4.4.1 below.

Processes and Scheduling. The co-existence of multiple, loosely interacting, larger activities in robotics applications (e.g., path planning) with tightly interacting, smaller activities (e.g., servo control) motivates the support of two different activity sizes in GEM: processes and micro-processes. A GEM *process* can be scheduled for execution at a moderate cost in time and, typically, interacts loosely with other processes. For representation of activities that are activated frequently and interact tightly with each other, a programmer may select a GEM *micro-process* (much like a *a thread* in later systems), which is a separable control activity within a single GEM process. It may interact with other micro-processes in the same or in a different process, and it can be activated at a low cost in time. Process scheduling constructs include a rich variety of explicit system calls for handling both periodic and sporadic processes and micro-processes running at different execution rates, including putting processes to sleep, waking them up or signalling them, controlling system timers used for process scheduling, etc. Micro-processes are scheduled using a lightweight communication construct, called a 'Poke' operation, which permits a micro-process waiting on an explicit input port to be activated at very low overhead.

Communication. GEM offers a mailbox-based communication mechanism coupled with explicitly reserved and managed communication buffers to support three different models of communication used in real-time applications. The first model supports asynchronous process execution with data loss. In this model, tasks generate outputs continuously based on their inputs, which are always assumed present. Communications between tasks can occasionally be lost, but tasks operate correctly as long as their inputs have not aged beyond statically defined tolerances known to the application programmer (e.g., processing inputs from a sensor). The second model support synchronous process execution without data loss, where tasks execute synchronously and their execution is driven by the acceptance of individual items of input from each other (e.g. a receipt of a service request and its parameters), which are not always assumed present. Inputs and outputs cannot be lost without jeopardizing the correctness of system operation. The third model supports the synchronous or asynchronous operation of processes with possible loss of aged data. It is a hybrid of models 1 and 2 which assumes that a fixed-size set of recent output items of one task is available as input to other tasks (e.g., useful for data logging).

Extensive system evaluation and performance measurements demonstrate the importance of both the varied functionality offered by GEM as well as the significant performance implications of design and implementation choices made by application and operating system implementors when building real-time software. The term *operating software* is coined to capture the tight coupling between real-time operating system and application code.

Multiprocessor Scheduling. In contrast to other research in real-time systems, higher level scheduling policies and mechanisms developed for GEM address the dynamic real-time applications exemplified by the ASV vehicle's operating software, where tasks can appear during system execution, with timing constraints not known at the time of program initiation. Scheduling, then, must address both the assignment or mapping of tasks to processors and their scheduling on individual processors. A novel offer-based mechanism implemented on GEM performed such runtime scheduling with sufficiently high performance to permit the dynamic scheduling of all but the lowest level control tasks on the ASV vehicle[BS91b].

4.2.3 The Spring Kernel: Exploring System Predictability

The goals of the Spring project[SR91] include: the development of dynamic, distributed, online real-time scheduling algorithms, the support of a network of multiprocessors, the development of multiprocessor nodes in order to directly support the kernel, and the development of real-time tools. The Spring system is physically composed of a network of multiprocessors, one of which was constructed as a testbed machine. Each multiprocessor contains at least one application processor, one or more system processors, and an I/O subsystem.

Periodic or non-periodic tasks are execution traces through programs, and are the dispatchable entities in the system. Non-periodic tasks have deadlines, and periodic tasks have recurring initializations and deadlines until they terminate. System tasks run on system processors, and application tasks can run on both application and system processors by explicitly reserving time on the system processors. In addition, the kernel contains task management primitives that utilize the notion of preallocation whenever possible to improve speed and to eliminate unpredictable delays.

The I/O subsystem is a separate entity from the Spring kernel. It handles non-critical I/O, slow I/O devices, and fast sensors. The I/O subsystem can be controlled by some other real-time kernel, if necessary.

The design of the Spring kernel is based on the principle of segmentation and reflection as applied to hard real time systems[SS87, SR94]. *Segmentation* is the process of dividing resources of the systems into units where the size of the unit is based on various criteria particular to the resource under consideration and to the application requirements. The basic idea of using segmentation in hard real-time systems is well defined units of each resource, to increase understandability, to permit on-line algorithms to explicitly combine well-defined resource units such that that predictability is achieved with respect to timing constraints[SR91]. *Reflection* refers to the concept of the system reasoning about its own state, and the state of the environment, in taking its actions. This is required for systems operating in highly dynamic environments, where hand-crafting all courses of action becomes infeasible.

Scheduling. Spring's schedulers are composed of four modules. As in most operating systems, the lowest level module is a processor-resident dispatcher, which simply removes the next task from a global system task table (STT) containing all guaranteed tasks. The tasks in this table are already arranged in proper order for the multiple application processors. The second module is a local scheduler, again one per processor. The local scheduler is responsible for locally guaranteeing that a new task can make its deadline, and for ordering the processor-specific tasks properly in the STT. The third module is the global scheduler, which attempts to find a site for execution for any task that cannot be locally guaranteed. The final module is a Meta Level Controller, which can adapt various parameters by noticing significant changes in the environment, and it can serve as the system's user interface. In [RS84, SRC85, ZRS87a, RSZ89], the authors present and analyze the details of their scheduling algorithms. Some other heuristic approaches are presented in [RSS90]. Section 4.6 discusses some custom hardware which is used to speed up real-time scheduling in Spring.

Memory Management. In the Spring kernel, the OS is core-resident. To eliminate large and unpredictable delays due to dynamic memory allocation (page faults and page replacements), the Spring kernel pre-allocates a fixed number of instances of some of the kernel data structures (task control blocks, stacks, buffers etc.), and tasks are accepted dynamically if the necessary data structures are available.

Inter-Process Communication. The Spring kernel supports synchronization and communication with five IPC primitives: SEND, RECV, SENDW (send and wait), RECVW (receive and wait), CREATMB (create mailbox). Mailboxes are memory objects. The Spring kernel avoids the need for semaphores by implementing mutual exclusion directly as part of the task schedule.

4.2.4 YARTOS: High Performance and Predictability for Multi-media Applications

YARTOS (Yet Another Real-Time Operating System) is an operating system kernel that supports the construction of efficient, predictable, real-time applications [JSP91, Jef89b, Jef93, Jef89a, Jef92]. The programming model supported by YARTOS is an extension of Wirth's discipline of real time programming [Wir77]. It is a message passing system with a semantics of inter-process communication that specifies the real-time response that an operating system must provide to a message receiver. These semantics provide a framework both for expressing processor-time dependent computations and for reasoning about the real-time behavior of programs. The YARTOS programming model is described in detail in [Jef89b].

YARTOS supports two basic abstractions: tasks and resources. A task is an independent thread of control that is invoked at sporadic intervals. The invocation intervals and deadlines for a task are derived from constructs in the higher level programming model. During execution, a task accesses a number of resources. A resource is a software object that encapsulates shared data and exports a set of procedures for accessing and manipulating data. Like a monitor, objects require mutually exclusive access to the data they encapsulate. A set of tasks is said to be feasible if all requests of all the tasks will complete execution before their deadlines and no shared resource is accessed simultaneously by more than one task.

The sequencing algorithm for tasks is a variation of the well-known earliest deadline first (EDF) scheduling algorithm. It is a preemptive priority driven scheduling algorithm with dynamic priority assignment[Jef92]. The novel feature of the algorithm is its dynamic manipulation of the deadlines of task invocations to ensure that the tasks maintain exclusive access to whatever shared resources they might be accessing. This manipulation of deadlines ensures that there will exist no contention for shared resources at run-time. Hence, YARTOS need not provide any special locking facilities for shared resources. Since tasks execute to completion in YARTOS, all tasks are executed on a single run-time stack. This improves memory utilization and reduces context switching overhead [Bak90]. YARTOS has been used to support a digital conferencing application [JSS92], a HiPPI data link controller and a virtual reality system.

4.2.5 Real-time Threads: Toward Portable Real-time Kernels

Portability is an important attribute of real-time operating systems[CMMS79] because their target hardware routinely varies from special purpose processors, to parallel machines, to distributed execution environments. Unfortunately, portability is difficult to attain due to common requirements of predictability and high efficiency for real-time kernels and application programs. As a result, implementors are often forced to repeat the implementation of low-level operating or runtime system (e.g., for Ada runtime system implementations) functionality for each target machine.

Recent work is addressing the issue of real-time system portability in two ways: (1) by definition of standards as in POSIX or TRON and (2) by layering higher-level real-time operating system functions on a common, lower-level set of runtime functions, called the 'Common Runtime System' for high performance languages (e.g., parallel Fortran), and in the case of real-time systems, inevitably called 'real-time threads'. Specifically, real-time threads underly several of the operating system described in this survey, including the later version of the ARTS system (see section 4.3.1), real-time Mach (see section 4.3.3), and the CHAOS^{arc} operating system (see section 4.4.3). Moreover, due to the importance of system configurability to different application domains, recent research on operating system microkernels is also resulting in configurability considerations for the threads layer itself[MS93b], at a minimum with respect to the threads schedulers included with such packages (e.g., the priority-based schedulers used in real-time Mach vs. the deadline-based, dynamic threads schedulers used in CHAOS^{arc})

The essential idea of all real-time threads packages is the provision of basic facilities for real-time support based on which higher-level real-time operating system facilities may be constructed. For example, concerning thread scheduling, the typical use of real-time threads in an actual system is to make static (initialization time or compile-time) guarantees for the application's statically defined task set, while runtime guarantees are made for tasks with unpredictable arrival and execution times. The role of real-time threads in this context is not to offer many different algorithms or mechanisms for threads scheduling. Instead, packages must offer the ability for inclusion of alternative runtime algorithms and associated data structures for threads schedulability analysis. Moreover, if a thread cannot be scheduled (i.e., schedulability analysis results in a negative decision), then it is up to application-level software or higher-level operating system modules[GS89a] to deal with such failures, by creation of alternative threads, by execution of exception handling code, etc.

The main functionality of real-time threads may be divided into calls regarding thread manipulation (e.g., creation), thread control (e.g., synchronization), and higher level utilities (e.g., condition variables). Moreover, for multiprocessor, real-time threads, package implementations distribute required functionality and data structures across the nodes of the target parallel machine. For example, the CHAOS^{arc} system's real-time threads layer described in [SZG91] and its on-line scheduling described in [SZ92] originally implemented on a BBN Butterfly multiprocessor maintains on each node (1) a pool of stacks for use by locally executing threads, (2) pools of thread descriptor and timing information blocks, (3) a local ready list in earliest deadline order – termed EL, for earliest deadline list, and (4) other structures used for maintaining scheduling information, and (5) a memory pool for future memory allocation/deallocation requests. Scheduling is performed as a result of a thread creation call, on the node on which the call is issued. Thread assignment to nodes is not automated. Instead, all thread creation calls explicitly specify a node on which the newly created thread is to be run. An extension of the package offers a global thread scheduling algorithm performing both automatic thread assignment and scheduling (see [BS91b, ZSA91]).

Sample calls to the real-time threads package concerning thread manipulation include (1) `RTthread_fork`, which creates, performs schedulability analysis for, and schedules a sporadic, real-time thread and (2) `RTthread_forkP`, which creates, analyzes, and schedules a periodic thread. Thread deadline, start time, and maximum execution time are assumed known when a thread is created. For example, a sporadic real-time thread may be created with:

```
RESULT
RTthread_fork(func, arg, node, start-time, runtime, deadline)
int             (*func)();
any_t           arg;
int             node, type;
int             starttime, runtime, deadline;
```

which creates a sporadic thread with the specified `starttime`, `runtime`, and `deadline`. In contrast to non-real-time threads packages, this call first creates a *timing information block* which is used for schedulability analysis, and then calls the dynamic scheduling algorithm to verify the schedulability of the thread being created on node `node`. Since schedulability analysis is performed for every real-time thread created, the time required for such analysis should be included in the thread's maximum execution time. The new thread's *thread control block* (TCB) and its stack are created only if schedulability analysis shows that its desired deadline can be met, resulting in return value of `T_SUCCEED`. Otherwise `T_CANNOT_MEET_DEADLINE` is returned.

Interesting issues concerning real-time threads include: (1) how can the schedulability information available at the threads level be shared efficiently with higher levels[ABLL92], (2) how can threads packages be structured so that both their interfaces (e.g., the required 'real-time' parameters) and the internal implementation of scheduling support is easily varied to permit the use of alternative scheduling algorithms and scheduler data structures, including those required for multiprocessor threads scheduling, (3) how can all threads-level calls be made 'safe' for use by real-time applications, including synchronization calls to condition variables or mutex locks (also see section 3), and (4) what specific support may be offered for package portability across the wide range of platforms used in real-time systems (see [MEG94] for a discussion of this topic for non-real-time threads). At this time, it is not clear whether real-time threads will evolve into a standard suitable for large-scale real-time system construction, but the utility of identifying lower vs. higher level support in real-time systems is broadly accepted, as apparent from the TRON standard and current research in operating systems. Real-time threads constitutes one flexible basis for real-time system construction.

4.3 Distributed Real-time Operating Systems

4.3.1 The ARTS Distributed Operating System

ARTS[TML90, TM89, MT90, TK88, TNR90a] is a distributed real-time operating system developed in the ART (Advanced Real-time Technology) project. The goal of ARTS is to provide users with a predictable, analyzable, and reliable distributed real-time computing environment, so that a system designer can analyze the system at the design stage and predict whether real-time tasks with various types of system and task interactions can meet their timing requirements. The novel aspects of ARTS are its initial focus on distributed real-time applications[TM89], its integrated support of monitoring tools used for timing evaluation and display[Tok88], and its later support of real-time communication protocols addressing live video transmission[TTCM92].

This survey focusses on later versions of the ARTS system, which was a precursor of real-time Mach as described in section 4.3.3. In these versions of ARTS, the original task-based representation of parallel and distributed programs is replaced by one using real-time threads, with an object model layered on top of threads. The basic object model is similar to the one first developed in the CHAOS operating systems[SGB87], which is described in section 4.4.1. Differences between both systems are due to the schedulability analysis in ARTS associated with object invocations, where each operation of an object has an associated worst case execution time, called a 'time fence' value and a time exception handling routine. When the operation is invoked from a real-time thread, the operation is executed if there is enough remaining computation time allocated to the calling thread to complete the operation. Otherwise, the invocation is aborted and an exception is raised. Objects are implemented using the C language or C++ with real-time extensions, called RTC++[ITM92].

Scheduling. The ARTS kernel implements an Integrated Time-Driven Scheduler (ITDS). The ITDS scheduler provides an interface between the scheduling policies and the rest of the operating system. The object oriented approach is also used to implement the scheduler, with the scheduling

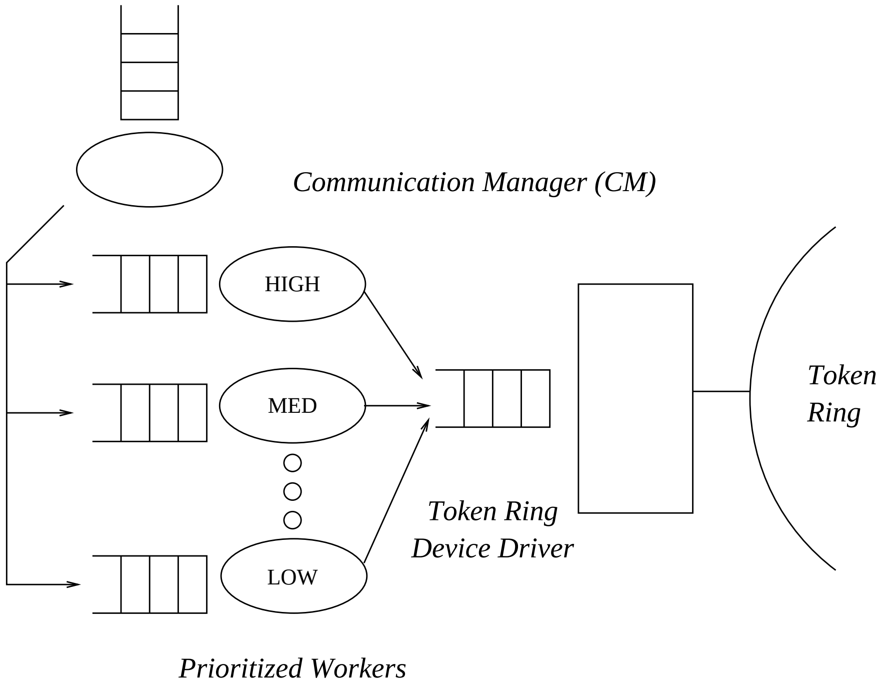

Figure 2: Structure of the ARTS Communication Subsystem.

policies embedded in the scheduler object. Each instantiation of the scheduler may have a different scheduling policy governing the behavior of the object, with only one instantiation being active at a given time.

Real-Time Threads. Underlying ARTS objects are real-time threads[TNR90b]. Each thread has an associated procedure name and a stack descriptor which specifies the size and address of the thread's stack. A real-time thread can be a hard real-time or a soft real-time thread. A hard real-time thread must complete its activities by its deadline time whereas the deadlines of soft real-time threads are less important. A real-time thread can be defined to be a periodic or an aperiodic thread based on the nature of their activities. Real-time threads in ARTS use priority based scheduling methods, in accordance with the rate monotonic scheduling work performed by others at CMU[SLR86, SLS88].

Synchronization. ARTS provides Lock and Unlock primitives to delimit critical regions. When a thread wants to lock an already locked variable, it is enqueued on a priority queue of threads. If the priority of the calling thread is higher than the priority of the thread which is in the critical region, the priority of the thread in the critical section is raised to that of the calling thread, thereby preventing priority inversion. When the thread leaves the critical region, its original priority is restored.

Communication Scheduling. In the ARTS project, an extended rate monotonic scheduling paradigm is used for communication scheduling[TMIM89]. This allows the system to integrate message and processor scheduling with a uniform priority management policy. In [Str], the author develops an algorithmic scheduling model for an IEEE 802.5 Token Ring Network and proposes a modification to the control algorithm of the token-ring adapter chip-set[MST89]. ARTS implements a communication structure which is intended to serve as a testbed for new communication algorithms and protocols as well as new real-time hardware[TML90]. Protocols such as VMTP have been successfully implemented on the ARTS kernel. Furthermore, a Real-time Transfer Protocol (RTP) is developed to explore real-time communication issues. RTP features prioritized messages and a time fence mechanism. The

RTP manager implements the RTP protocol and, thus, forms a single point through which remote communications must pass. In [TTCM92], the Capacity-Based Session Reservation Protocol (CBSRP) is extended in order to realize predictable real-time communications. The extension of CBSRP is evaluated on a Fiber Distributed Data Interface (FDDI) in ARTS.

As stated earlier, some of the interesting contributions of ARTS are their results regarding the display of scheduling information and research on multi-media communication protocols using FDDI links (presented in the context of the real-time Mach project, also being done at CMU). Other related research at CMU (not strictly connected to the ARTS project) includes research on real-time transactions by Sha et. al.[SLJ88] and work on rate-monotonic scheduling by Lehoczky et. al. [SSL89]. Research on on-line program monitoring suitable for parallel and distributed real-time systems is described elsewhere [OSS93, SR85].

4.3.2 The Maruti Distributed Real-Time Operating System

The main focus of the Maruti project [GMAT90, Agr90, AL87, MA90, YA89, LA90] is to examine the constructs of future distributed, hard real-time, fault tolerant, secure operating systems. Maruti is an object-based system, with encapsulation of services. Objects consist of two main parts: a control part (or joint) which is an auxiliary data structure associated with every object, and a set of service access points (SAPs) which are entry points for the services offered by an object. Each joint maintains the object's information (such as computation time, protection and security information) and requirements (such as service and resource requirements). Timing information, maintained in the object, is dynamic and includes temporal relations among objects. A calendar, a data structure ordered by time, contains the name of the services that will be executed and the timing information for each execution.

In Maruti, each application is described in terms of a computation graph, which is a rooted directed acyclic graph. The vertices represent services and the arcs depict timing and data precedence between two vertices [LA87]. Objects communicate with one another by *semantic links*. Such links perform range and type checking of the information. Objects that reside in different sites needs *agents* as representatives on remote sites.

Maruti is organized in three distinct levels: the kernel, the supervisor, and the application level. The kernel is the minimum set of servers needed at execution time. It consists of a set of core-resident objects including: (1) a dispatcher which is invoked upon the completion of a service or at the start of another service, (2) a loader which loads objects into memory, (3) a time server which provides the knowledge of time to executing objects, (4) a communication server responsible for sending and receiving messages and (5) a resource manipulator responsible for resource management.

Supervisor objects in Maruti prepare all future computations, ensuring their timely execution by pre-allocation of resources, whenever possible. The supervisor level objects in Maruti are: (1) the allocator which extracts the resource requirements from requests' resource graphs and allocates the required resources, (2) the verifiers which verify resource usage and reservation, (3) the binder responsible for connecting communication objects, as well as for verifying that their semantic relations ares properly established, (4) the login server providing a user interface to Maruti, and (5) a name server responsible for bridging different name spaces and keeping track of machine locations and status.

Scheduling algorithm research in conjunction with Maruti is described in [NP91], where the notion of partial evaluation of program constructs is applied to real-time systems. The initial implementation of Maruti on a network of SUN Unix workstations was followed by a partial native kernel implementation on Dec-Stations, and is now being replaced by an implementation based on Mach [BKLL93].

4.3.3 Real-Time Mach – Real-time Threads and Communications

Real-Time Mach (RT-Mach)[TNR90b] primarily addresses the real-time aspects of threads[TNR90b], thread synchronization [TN91], interprocess communication (IPC)[KNT92], and some other mechanisms to allow greater predictability, as mentioned at the beginning of section 4.2. In addition, the RT-Mach project has developed a tool-set for real-time program design and analysis. Work on RT-Mach elsewhere[NYM92] addresses extensions for multimedia applications, including the required extensions for real-time scheduling, the support of user-mode device drivers, and a temporal paging system.

Real-Time Threads. As in other operating systems (see section 4.2.5) RT-Mach augments the threads model with timing attributes. While using a standard definition of a periodic thread (a newly instantiated thread must be scheduled at intervals determined by its period), the definition of an aperiodic thread in RT-Mach includes a worst-case *inter-arrival time* after which the aperiodic thread recurs. Furthermore, both periodic and aperiodic threads can have soft or hard deadlines, where a soft-deadline thread has an *abort time* used by the scheduler to determine when to abort the thread. In addition, the semantic importance of a thread is specified via a *value function* associated with the thread. There are primitives to create, terminate, kill and suspend a thread in addition to those for getting and setting the attributes of a thread.

Threads Scheduling. RT-Mach uses the *Integrated Time-Driven Scheduler* (ITDS) originally developed for ARTS and extended for RT-Mach. Mach provides processor sets (which are collections of processors available to an application), with run queues specific to processor sets [Bla90]. The ITDS scheduler extends this approach by allowing five different policies (Rate Monotonic, Fixed Priority, Round Robin, Round Robin with Deferrable Server, and Round Robin with Sporadic Server) on each processor set in RT-Mach, with primitives to get and set the scheduling policy. The ITDS scheduler utilizes a *capacity preservation* scheme, whereby the available processor cycles are first assigned to the hard periodic and aperiodic jobs, and whatever remains is then handed to the soft real-time jobs according to the respective value functions.

Synchronization. RT-Mach's algorithms for real-time synchronization are described in [SRL90, RSL88]. These algorithms determine queue orderings for the mutex lock and condition primitives offered for synchronization in real-time threads, with the purpose of alleviating blocking problems known to exist in real-time synchronization. In addition, RT-Mach permits different synchronization policies to be used with different instances of locks and conditions. As a result, a thread can be either *non-preemptable* (preemption is not allowed while the thread is in a critical section), *preemptable* (a higher priority thread can preempt the current running thread, but the higher priority thread must block if it needs to access a critical section which is being used by the current running thread), or *restartable* (a higher priority thread can preempt the current running thread; if the higher priority thread needs to enter a critical section being used by the current thread, it does so; the preempted lower priority thread restarts later from the beginning of the critical section).

By choosing appropriate queue-ordering schemes with one of the permissible preemption schemes, five synchronization policies can be supported:

- Basic Priority: Operations of this policy are NULL functions.
- Kernelized Monitor: No preemption is allowed while a thread is in the critical section.

- Basic Priority Inheritance: The lower priority thread executing the critical section inherits the priority of the higher priority thread trying to access the critical section.
- Priority Ceiling Protocol: The execution of the thread is blocked if the priority ceiling of the thread is not higher than all locks owned by other threads.
- Restartable Critical Section: The critical section is aborted by the lower priority thread when a higher priority thread tries to access it. The lower priority thread restarts from the beginning of the critical section.

Real-time Communications. The IPC extensions in RT-Mach are derived from the synchronization results described above. Specifically, it is clear that queueing messages in FIFO order, while acceptable for non-real-time applications, may cause unbounded delays for receiving high-priority messages. This problem can be addressed by use of priority-based queueing in message buffers. In addition, there must exist primitives to propagate priorities from the sender of a message to the receiver, and there must be mechanisms to inherit priorities from the sender to the receiver of a message (otherwise, the processing of a message or the receiving of a message by a server might be interrupted or delayed [KNT92]). Furthermore, since there may be multiple receivers in a single server, a decision has to be made about which recipient thread should process an incoming message (the same problem is discussed in [SGB87] for multiple threads serving an object invocation).

To handle real-time message handling, four attributes must be defined for message ports: (1) the *message queueing* attribute specifies whether priority-based or FIFO ordering should be used, (2) the *priority hand-off* attribute modifies the receiving thread's priority when a message is received (if enabled, the priority of the receiver is set to that of the sender, or set according to the selected policy), (3) the *priority inheritance* attribute, if enabled, makes the receiver inherit the priority of the sender thread which sent the highest priority message to the port, and (4) the *message distribution* attribute selects a receiver thread from among those available in FIFO order if 'arbitrary' policy is specified, and according to a given priority if priority-based selection is specified.

General Research with RT-Mach. A second effort at enhancing Mach for real-time and multimedia applications has a concept of real-time threads similar to the one described in section 4.2.5, using use *deadline-driven* and *event-driven* threads for appropriate types of multimedia devices[NYM92]. Deadline-driven threads are used for devices where the media does not deteriorate if operations to the device complete before a deadline. Event-driven threads are used for those devices where the operation has to be initiated immediately after an event occurs. Such devices may also have explicit deadlines, where device response deteriorates as the software's response time increases. The characterization of real-time threads includes a start time, a deadline, a worst-case execution time, and a 'weight' which specifies the relative importance of the thread. Event notifications are similar to user-level device management: upon interrupt, a handler performs a small routine in kernel mode on the device, an asynchronous system trap is posted, the preemptive-deadline scheduler is invoked, and then user-level operations follow.

In [NYM92], the authors also develop a *temporal paging system* (TPS), which allows real-time access to large data segments, as is often required in multimedia applications. This is required because page faults introduce problems of predictability, unless regions of address space are locked into core-resident pages. Unix allows this in superuser mode, while Mach's `vm_wire` primitive allows such locking of pages. The TPS provides a more convenient set of mechanisms for use by devices that need large address spaces. In addition, while conventional memory is addressed using only a spatial coordinate (the address), temporal memory is addressed using spatial and temporal addresses[5]. The contents of temporal memory can change while the temporal coordinate is valid. After that duration, the original memory contents recover (they are either removed from the memory space, or become 'ordinary' memory, depending upon an attribute which specifies the temporal memory). Moreover, the same physical page can be shared by multiple temporal pages, if those pages are not used in the same temporal interval, thereby reducing total memory usage. The TPS implementation interacts with the operating system kernel by providing it with hints about what pages should be preloaded or swapped out at any time. TPS also provides a uniform interface for real-time access to memory mapped to specific devices, since it lets the kernel load/swap pages, and the application programmer need not know the hardware and/or latency-related idiosyncrasies of the specific device or storage system.

4.3.4 HARTOS – Fault-Tolerant Embedded Systems

The HARTOS real-time operating system is being constructed for a distributed-memory architecture consisting of nodes connected in a hexagonal mesh [KKS89]. The nodes consist of collections of application processors (APs), and a network processor (NP). The APs are VME-bus based 68040s: up to 4 in a single node. Different nodes communicate using the X-kernel[PHOA89]. The operating system focusses on support for on-line scheduling, in part targeting applications in autonomous robot control [KMTD86] and multi-media applications. The HARTS project has contributed several novel scheduling algorithms – especially on-line scheduling for distributed memory machines (e.g., sets of workstations) [HS91b, HS92] – though they do not appear to have been incorporated into HARTOS. The chief contribution of HARTS is architectural support for real-time communications (this is discussed in section 4.6); here, we simply describe the salient attributes of the HARTOS operating system.

HARTOS specifically addresses fault-tolerant communication, using some of the same approaches as those employed by the MARS system described in section 4.3.5. For predictable communications, the local memory on each node has a hardware mailbox interrupt, which raises a CPU interrupt on a write to the top 256 bytes of dual ported memory. The original design had support for process-level N-modular redundancy, using the concept of process groups. Local deadlines are imposed on each hop of a message (rather than deadlines for end-to-end delivery) [KKS89]. HARTOS is built on top of pSOS^+. While the pSOS^+ executive provides the low-level mechanisms for processor and memory management, HARTOS extends these for the multiple-node environment, and handles real-time communications. Timeouts values are specified in several of the communication calls; clock synchronization software and hardware is used to obtain close synchrony between the nodes, thus achieving a global time base.

4.3.5 MARS: Fault-Tolerance in Distributed Real-Time Systems

The MARS (MAintainable Real-time Systems) project focusses on the fault-tolerant aspects of hard real-time systems used in critical applications. The objectives of MARS [VK93] are to provide guaranteed timely response under peak load conditions, to support real-time testability by breaking up the system into encapsulated subsystems, to facilitate on-line maintainability of hardware and software, and to offer fault tolerance and architectural and operating system support for systematic software development through predictable temporal properties. The system is targeted at process control applications in industrial real-time systems.

[5] The temporal coordinate has to be within a known range.

Time Driven System. The designers of MARS hold that a time-driven system – where the system initiates all activities at pre-determined times – is better suited for predictable performance than an event-driven system – where significant events in the environment produce corresponding activity in the system[6]. This is because a lot of information about system activity is available *a priori* in a time-driven system, and can therefore, be taken into consideration in the various protocols used in the system. The advantages of the time-driven mechanism include the fact that 'control signals' are wholly dependent on physical time. Thus, in the presence of a global time base (as in the MARS, and in the HARTS system described above), there is little need for 'control signals' to cross subsystem interfaces [Kop93b, KG94]. Therefore, time-driven systems are more predictable and testable, though less flexible and dynamic.

System Architecture. A MARS application consists of a set of *clusters*, which are autonomous subsystems. The several components of a cluster are connected by a real-time bus, where each component is a self-contained computer, including the application software. Each component runs an identical copy of the operating system [DRSK89]. Different clusters are connected through an inter-cluster interface, forming a network (rather than a hierarchy). The clusters consist of *Fault Tolerant Unit*s (FTUs), which consist of replicated components providing redundancy. Shadow components update their own internal state and monitor the operation of the active components. A shadow component becomes active when an active component fails. Each message is also sent twice on two real-time busses. Each component contains a local clock, and all clocks are synchronized to an *a priori* known maximum deviation.

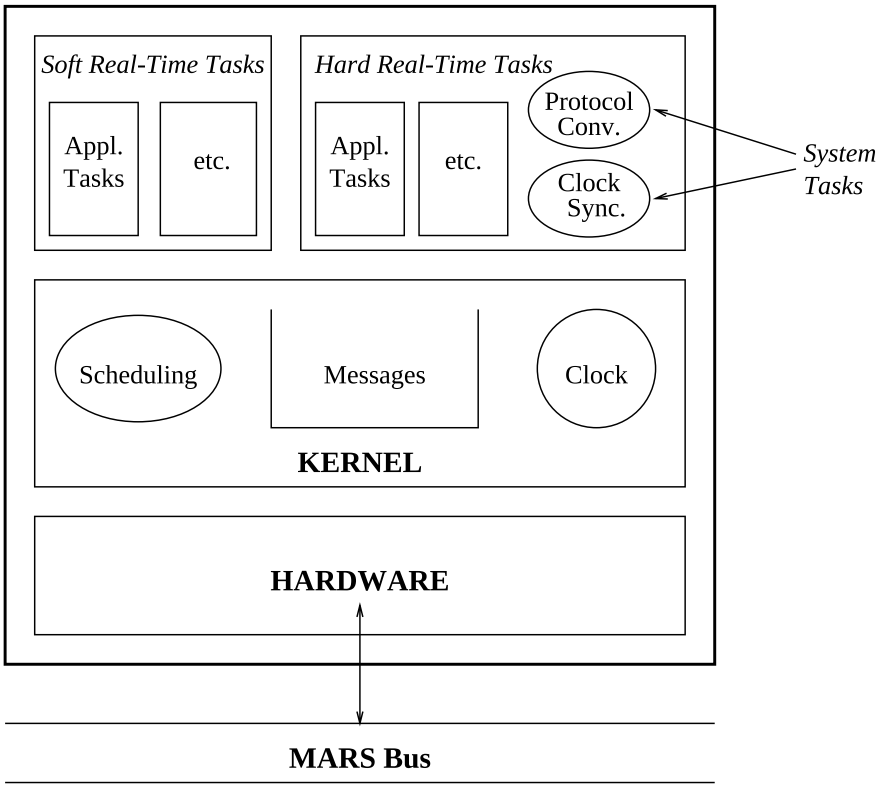

Figure 3: Structure of a MARS component

Fault Tolerance. MARS addresses both transient [KKG+90] and permanent faults. Messages have checksums, and hardware components are self-checking; based on experimental data, the hardware is

[6] Most of the real-time systems and real-time scheduling strategies discussed in this paper consider the system to consist of a combination of time-driven and event-driven activities.

assumed to have an error coverage exceeding 99%. The remaining 1% are detected by the operating system and application software. The operating system uses robust storage structures[7], assertions to check for consistency of function parameters and local variables, plausibility tests for results of system routines, and checks for the execution times of system routines. Application software detects errors by attempting to execute each task twice[8], thereby catching transient faults of duration less than the task execution time. Transients of longer duration are assumed to cause many subsequent errors, resulting in a high probability for their detection.

MARS is *fail silent*, which means that a component is turned off on detecting its first error. A fault is discovered using a membership service protocol, where every active component knows (by a bounded interval) which components have failed. Upon fault discovery, a shadow component takes up the task of the failed one[Be85]. The shadow has exactly the same 'internal state' as the failed component, since it performs all of the activities of the component it is shadowing, without using the TDMA slot (i.e., it does not communicate its results). Upon failure, the TDMA slot of the failed component is simply assigned to the shadow. Failure of a shadow component is handled by automatic restart in the case of a transient failure (which has no effect on the cluster, the shadow not affecting the active components), and 'bringing down' the shadow in case of a permanent failure.

Clock Synchronization. MARS uses custom hardware to obtain global time values [KO90]. Each component 'reads' the time of other components: this might contain a measurement error. The custom hardware ensures that this error is less than 4 μsec. Correction terms for the local clocks are calculated based on the values read. This results in clocks being synchronized with respect to each other. Synchronization with world-time is achieved by receiving time-values with long-wave radio from a standard source. Each MARS cluster contains one long-wave receiver; the error in obtaining global time is less than 100 μsec. The custom time transmission hardware across clusters ensures that changes in local time are gradual, and do not 'jump' when a correction is applied.

Tasks and Messages. Tasks in MARS can be periodic or aperiodic, and they are scheduled with a static scheduling mechanism. The OS kernel [DRSK89] runs entirely in supervisory mode. Hard real-time tasks (mostly periodics) are run at specific intervals that are known at the time of system initialization. Soft real-time tasks (usually aperiodics) are run at intervals not used by hard real-time tasks. Communication among tasks is performed by message passing, where all messages have cluster-wide unique names. MARS also uses *state messages*, which are produced periodically at predetermined times. They convey information about the state of the system or the environment at a particular time, and such state is assumed to be steady for a certain duration in the future. The time when a message is sent is pre-determined by a pre-runtime scheduler. State messages deliver 'read-only' state about the system which cannot be altered by tasks. Therefore, they can be read an arbitrary number of times by an arbitrary number of tasks. This eliminates the need for tight synchronization between producers and consumers of state messages (it is the responsibility of the receiver to assure that all state messages are available when it 'reads' them). Each time a new state message is received, the previous 'version' is deleted – receiving state messages being equivalent to reading a sensor. Therefore, flow control becomes unnecessary, buffer requirements are static, and only a pointer to a message is delivered (rather than copying) on receipt of a message.

To avoid unpredictable message delays associated with CSMA/CD protocols, MARS uses a TDMA protocol to provide collision-free access to the Ethernet. At most one hard real-time message is assigned to each TDMA slot, and a soft real-time message is assigned to a slot if it is empty. Further details of the MARS communication protocol – the Time Triggered Protocol – can be found in [KG94].

[7] 'A storage structure is called robust if some (specified) number of changes made to its structural information can be detected [KKG+90].'

[8] An analysis tool is used to determine the worst-case execution time of a task. The 'slack time' between this worst-case time and the actual time taken to run the task is used to run the same task a second time.

MARS allows only one kind of interrupt, a periodic clock interrupt. Interaction with peripherals is through polling. The handler is divided into a 'minor-handler', which is executed frequently (every millisecond) and written in assembler, and a major part executed less frequently (every eight milliseconds). The minor handler suspends system calls, while the major part blocks until a system call completes. This approach allows time-critical operations on devices (e.g., polling) to be performed with greater frequency.

Scheduling. MARS performs scheduling off-line. It is assumed that the running task will itself yield the CPU before the end of its quantum; else, an error condition is flagged. Task switching is performed by the major handler, every eight milliseconds. It is possible to combine different schedules in an application. The change can be triggered by invoking a system call or receiving an appropriate message.

The designers of MARS state that only time-triggered architectures can provide the predictable performance necessary in distributed real-time control systems [KG94]. The structure and operations of MARS is a good example of a static, predictable fault-tolerant, real-time system.

4.3.6 The CTRON Operating System Framework

The CTRON[OWK87, WOK+87, KOOH87, Sak89b] real-time operating system is a part of the TRON[9] [Sak87c] platform for industrial systems. The general TRON project is designed for network nodes consisting of different kinds of computers. The goal behind the design of CTRON is (1) to achieve software portability, while (2) providing a high level of performance.

To assure software portability, the operating system is subdivided into two functional sections. One section consists of functions that hide the processor architecture and provide common interfaces; these functions are not portable among nodes with different processor architectures. The other section offers portable functions that assume a common interface.

Network nodes can be classified into various groups based on the kind of services they provide. A few examples are: switching nodes (for circuit switching or packet switching, etc.), communication processing nodes (voice storage service, facsimile communication processing, etc.), information processing nodes (files and data bases, data processing service etc.), and workstation nodes (to provide user-friendly interface to the end users). These groups, normally, have different operating system requirements. Switching nodes accommodate a number of network nodes and have to be capable of processing a multiplicity of nodes and information simultaneously. For example, workstation nodes process less than a hundred operations per time unit, while information processing nodes perform approximately a thousand operations. To accommodate such varied requirements, the operating system interface is divided into two classes: (1) interfaces that can be used for all applications and (2) interfaces that are used selectively for specific applications. Accordingly, the CTRON operating system model is divided into two groups: a common model group and a selectable model group. The various functions in the latter group are further classified into several subgroups, based on the requirements of application domains. The kernel interface is divided into four parts: a group for the common model, a group for the advanced real-time model, a group for the advanced complex function model, and a group for the advanced virtual memory model. It is possible to provide a combination of these groups (there are six subsets available in total) for various service systems.

CTRON uses a virtual processor model for the purpose of hiding underlying processor architectures.

[9] This project studies the operating system interfaces/requirements for real-time processing. This project consists of several sub-projects, including ITRON[Mon87] for industrial embedded systems, BTRON[Sak87b, KTKS87] for business workstations, TRON CHIP[Sak87a] for a microprocessor used in the ITRON and BTRON, etc.; see section 5.1.

This model is characterized by a set of objects that indicate the abstraction of processor functions. The objects supported by the CTRON kernel are: tasks, synchronization, exceptions, timers, memory, interrupts, and black box.

Tasks. Tasks are the parallel processing units of a program. To attain real-time performance and high degrees of multitasking, the CTRON kernel implements a two-level scheduling model. The task model consists of scheduling functions required by high-level operating system utilities and by application programs. The pseudo-task model support a cheaper 'task' abstraction, which is used to implement a batch polling form of processing. For example, pseudo-tasks are used for high performance polling of circuit equipment. To reduce overheads in task creation, CTRON supports a dormant state, which is a pre-ready state in which all required system resources have been provided (except for CPU scheduling rights). To further improve real-time processing abilities, CTRON also defines a cyclic task schedule from the dormant waiting states, using fixed time periods.

Synchronization/Communication. The CTRON kernel offers optional functions for synchronization and communication between tasks. CTRON provides event flags, semaphores, and message boxes for simple synchronization, mutual exclusion, and message communication respectively. The CTRON kernel provides a logical lock function without queueing of serially reusable resources to improve performance and predictability. It also provides an Ada-like rendezvous function as an operating system interface for synchronous communication between tasks.

Exceptions, Timers, and Interrupts. An exception is defined as an asynchronous interrupt signal to a task, as distinct from an asynchronous interrupt signal to a real processor. Exception management functions include the registration of exception-processing handlers corresponding to tasks, management of exception masks, generation of asynchronous exception from software etc.

Two kinds of timers are supported: a system timer (one per system), and private timers (one or more per task). They are useful for real-time communication protocol processing.

Interrupt related operations include registering interrupt handlers and setting and releasing interrupt masks. A pseudo interrupt generation operation from software is also possible. With the help of a mapping table, physical interrupt (intended for software operations) are changed to logical interrupts to increase portability of interrupt processing handlers created by users and intrinsic to the kernel.

Memory Management. Two memory models are supported: (1) a common memory model (which applies regardless of the processor architecture), and (2) a selectable memory model (which recognizes a virtual memory architecture). The selectable model makes efficient use of virtual memory. The memory management interface hides all hardware architectures from the users.

Black Boxes. Some objects are strongly dependent on processor architecture or system configuration. In such cases, it is difficult to prescribe a common-use model. Instead, individual interfaces for individual systems are needed. However, this presents an obstacle to software portability. CTRON defines a black box model for this purpose. It defines only system call names in the black box model; it does not define input and output conditions, error conditions, or side effects. In [OWK87], the authors discuss this model in detail.

Brief Evaluation. CTRON is comparable to other operating systems standards for real-time control offered in the U.S. in their attempt to construct a portable platform for use in different applications on different target machines. However, there remain issues regarding performance and flexibility due to the rigid definition of low-level CTRON functions (e.g., cyclic task scheduling, or the single RPC semantics part of CTRON). It would be more appropriate if interfaces were defined such that alternative mechanisms and policies are easily added to the systems being constructed on the CTRON basis. CTRON implementations in the U.S. are being performed by Tandem Computers.

4.4 Object-Oriented Real-time Operating Systems

4.4.1 The CHAOS Operating System for Embedded Applications

The CHAOS operating system[SGB87, SB87, Gop88] is an object-based layer on top of the GEM executive described in section 4.2.2. Its goals are to support three basic principles in real-time operating system construction for multiprocessor platforms: (1) minimal hardwired or 'basic' operating system functionality, (2) increased sharing of operating system functions by application-level code by provision of explicit constructs permitting the application to (a) select from and (b) parameterize the operating system functionality it desires, and (3) explicit support for runtime configuration of operating system or application functions to facilitate software adaptation to variations in the embedded systems's underlying hardware or external execution environment.

CHAOS is implemented on a shared memory, embedded multiprocessor identical to the one running the GEM system's ASV vehicle's application software[SBWT87]. However, the programming model offered by CHAOS is quite different: it describes a parallel program as a set of abstract *objects* that interact by invocation of each others' operations. Each object has a *type*, unique *name*, and a set of valid *operations*. Object types are user-defined and are not dynamically checked or known to the operating system. To invoke an operation of an object, only the identifier for the object and its operation name need be known, so that the parallel operating software generated for a set of objects specified by the programmer may be adapted considerably without changing its object description. For example, an object can be a *passive* object – its operations are implemented as code modules which when invoked execute within the thread of execution of the invoking object – or it can be an *active* object – its operations are realized either as a single process or as a set of executable GEM processes.

The actual number of processes associated with each object is determined statically by performance or reliability considerations (e.g., by the frequency of invocations on the object). Multi-process objects are controlled by a single coordinator process accepting object invocations and scheduling them for processing by one of several server processes.

CHAOS demonstrates that the object model of software can be specialized and implemented efficiently so that it may be used in the real-time domain. The following specializations and implementation attributes exist for objects:

- Objects of different weights may be created, ranging from light-weight, passive objects that have no internal processes to heavy-weight objects that may have multiple internal processes. Therefore, in contrast to the micro-processes within a GEM process that cannot execute concurrently, an object may exhibit internal parallelism.
- CHAOS objects interact by means of invocations. In order to implement efficient object interactions, multiple primitives exist for the invocation of an object's operations. These primitives differ in their semantics, performance, lifetimes, and reliabilities; and their diversity emphasizes the fact that existing implementations of objects or of RPC semantics for computer networks cannot be trivially applied to the real-time domain.

- Explicit scheduling parameters and real-time constraints can be attached to object invocations, and the queueing policy for invocations in the target object can be controlled and changed, as well.
- Since objects can reside anywhere in the multiprocessor hardware, the invocation code can select the communication link to reach the target object that best fits the invocation's required performance or reliability. These links include the system bus, and serial and parallel links. In addition, CHAOS allows the programmer to explicitly control the visibility of objects by locating them in 'low latency of access' local memory or in 'higher latency of access' global (shared) memory.

The basic contributions of CHAOS include the manner in which the object model is specialized for real-time applications and the detailed investigation of implementation details of object invocation, a precursor to later work in non-real-time systems concerning efficient implementations of RPC primitives[SB89]. Regarding the latter, the kernel modularizes the different steps to be performed for any object invocation so that different types of invocations are easily assembled while still offering high performance. Specifically, CHAOS invocation primitives range from (1) *ObjFastInvoke* – fast control invocations that may be used to toggle actions through (2) *ObjInvoke* – invocations that entail the transfer of control and parameter-passing (much like RPC implementations) to (3) *ObjStreamInvoke* – streaming invocations with low incremental cost of data transfer (similar to streaming sockets in Berkeley 4.2 Unix). In addition, alternative implementations may be selected based on (a) whether or not the invoker intends to block on the status of the invocation (synchronous vs. asynchronous invocations) and on (b) whether or not all resources acquired for the invocation (e.g., parameter blocks) must explicitly be released after each invocation or whether the cost of releasing resources can be amortized over several invocations (persistent invocations). Since invocations may consume memory resources, CHAOS also provides primitives for memory management and garbage collection.

As with other object-oriented operating systems, no runtime support for inheritance is provided, thereby eliminating both inefficiencies in operation access as well as uncertainties in such access times. As a result, CHAOS is able to demonstrate that objects can be implemented efficiently enough to satisfy the timing requirements of even low-level control loops in real-time systems. Last, while most memory management in CHAOS maps to the lower level facilities offered by GEM, interesting issues concerning main memory arise when organizing and handling the buffer pools (i.e., parameter and invocation blocks) necessary for implementation of operation invocations, including ensuring system predictability. Research in real-time systems using the CHAOS kernel focusses on the runtime adaptation of object-based parallel programs, including the work of Bihari[BS91a] and Gopinath[Gop88, GS89b, GBSG89]. Additional results attained with a next generation object-based real-time kernel – the CHAOS^{arc} kernel – on commercial multiprocessor platforms are discussed in section 4.4.3.

4.4.2 The Alpha Operating System

Alpha[JN90a, JN90b, NCS+90] is a non-proprietary operating system for large, complex, distributed, real-time systems. Alpha arose from the Archons Project at Carnegie Mellon University, which offered a partially implemented prototype operational in 1987. Versions now run on Sun, Concurrent, and SGI hardware.

Alpha's kernel provides its clients with a coherent computer system on an underlying platform that may be composed of an indeterminate number of networked physical nodes. Its principle abstractions are objects, operation invocations, and threads. The Alpha object model resembles the Clouds distributed operating system[DLJA88] in that objects are passive abstract data types consisting of code and data in which there may be any number of concurrently executing activities. Each instance of a client level object has a private address space, and exists entirely on a single node. Objects can be dynamically migrated among nodes. Initial object placement is specified by the user. Objects may be transparently replicated, with members of the replicated set residing on different nodes. The kernel defines a suite of standard operations that are inherited by all client objects, and these standard operations can be overloaded. Objects are named by capabilities that are protected by the kernel and not directly accessible by applications. Alpha's kernel offers atomic transaction-controlled updates to an object's permanent representation.

Real-time Threads. Alpha threads are the units of schedulability, and they are fully preemptable. A thread is the executing entity, which moves its locus of control[NCS+90] among objects via operation invocations. It is a distributed computation which transparently and reliably spans physical nodes. A thread carries parameters and other attributes related to the nature, state, and service requirements of the computation it represents.

The invocation of an object's operation is the vehicle for all interactions in the system, including operating system calls. Threads move from object to object via invocations. Operation invocation has synchronous request/reply semantics.

Exception Handling and Transactions. An Alpha exception block is a kernel level mechanism for the specification of application specific consistency and correctness. If an exception occurs, control is returned to the appropriate exception handler where the operation can be retired or some other compensating action can be taken.

Alpha utilizes a transactional distributed computing model for trans-node concurrency control and integrity because it can be well integrated with Alpha's block structured management of real-time constraints and exceptions, and can be tailored to meet application-specific needs. Alpha's kernel provides transaction mechanism for atomicity, permanence, and application specific concurrency control individually.

4.4.3 CHAOS^{arc} : Atomic Real-time Computations in a Configurable Kernel

Configurability with Objects, Attributes, and Policies. The CHAOS^{arc} kernel[10][GGSW88, GS89a, SGZ90, GS93] is actually a family of object-based real-time operating system kernels that address portability, extensibility, and customizability for low-level and subsystem-level operations. The family is *extensible* in that new abstractions and functionalities can be added easily and efficiently such that uniform kernel interfaces are maintained. This is useful because it permits the implementation of domain or target machine specific features while preserving some given kernel interface for existing programs. It also provides an environment for experimenting with and prototyping of new operating system constructs and policies.

The family is *customizable* in that existing kernel abstractions and functions can be modified easily. This is useful because it facilitates changes to an operating system for uses with different target architectures or application domains. The family is *portable* in that its implementation is based on the Mach Cthreads standard[SFG+91] as a base layer for uniprocessors and parallel architectures – called the CHAOS^{base} member of the kernel family. However, upwardly compatible modifications have been

[10] A **C**oncurrent, **H**ierarchical, **A**daptable **O**perating **S**ystem supporting **a**tomic, **r**eal-time **c**omputations.

made to the Cthread interface in order to accommodate real-time applications, called real-time threads (see section 4.2.5 and [ZS91]). Extensibility and customizability of the family are attained by use of the object model for description of the operating system's interface[SGB87, HFC76] and for operating system and user program implementation.

The object model of CHAOS^{arc} offers explicit support for system extension, configuration, and customization. Each application program is composed of a number of user objects, which use system-defined objects to access operating system services. However, as opposed to other object-based real-time kernels[SGB87] and in order to attain extensibility and customizability, system interfaces are not simply described by exporting some system objects (i.e., their classes[HFC76]). Instead, the exported object classes are refined by a novel abstraction supported by the CHAOS^{min} layer of CHAOS^{arc} , called *attributes*[11]. CHAOS^{min} is the lowest-level object-based layer in the CHAOS^{arc} kernel. It defines *attributes* to be abstract properties that can be associated with classes, objects, object states, operations, and invocations. However, CHAOS^{min} neither defines nor interprets attributes; it merely passes them to a *policy* object that may be associated with any CHAOS^{min} objects. Exactly one policy object is associated with each object (user or system). Such a policy has complete control over the object's execution and can therefore, interpret and enforce its attributes. Policy objects are invoked implicitly as a result of operations (e.g., creation, invocation, ...) on the objects they manage. Each policy is itself implemented as an object and may use a limited form of inheritance for implementation of new functionality.

Objects are extended and customized, then, by changing their attributes instead of changing their operations or types. As a result, the interfaces of user programs to system objects need not be altered when policies are changed to support additional attributes or when the implementations of policies are varied. The resulting structure of the kernel family consists of three components: (1) the *nugget* implementing CHAOS^{base} , which is a real-time cthreads package[12][ZS91], (2) the *vanilla layer* implementing CHAOS^{min} , and (3) the *policies* or application implementing certain CHAOS^{arc} flavors and attributes. CHAOS^{base} is the machine dependent component that implements the basic abstractions used by the remainder of CHAOS^{min} : real-time *execution threads*, *virtual memory regions*, and *synchronization primitives*.

The *vanilla layer* is the fixed, machine independent component that implements the functionality of CHAOS^{min} : classes, objects and invocations. It supports the following built-in object flavors: *ADT*, *Monitor*, *TADT*, and *Task*. A primitive object of flavor *ADT* (abstract data type) is passive[SGB87] and has well-defined internal state. Its operations are executed in the address space and by the execution thread of the invoker (caller). An object of type *monitor* is a passive object that allows exactly one execution thread at a time to execute its operations. Monitor objects behave like Hoare monitors, with the exception that their explicitly specified scheduling policies (for the selection of invocations to be executed) may differ among instances. An object of flavor *TADT* (threaded abstract data type) is active and is used for the representation of parallelism in CHAOS^{arc} applications. A TADT object creates and starts a new execution thread for the execution of each operation invocation. A CHAOS^{min} *task* object is like an Ada task in that it consists of a single active thread of control and has multiple entry points selected by this thread. The vanilla layer does not implement any invocation attributes or special invocation semantics. All object operations (creation, deletion and invocation) go through the vanilla layer. If such operations involve the use of non built-in flavors or attributes, the vanilla layer redirects the operation to the appropriate policy operation using well-defined rules.

The CHAOS^{arc} Layer: Atomic Computations. The most interesting set of policies con-

[11] Attributes and policies predate and are more flexible than similar notions developed for commercial object-oriented systems, such as the Spring operating system's *subcontract* abstraction[PM93].

[12] All threads created by a single user process share the process' address space yet have their own execution states.

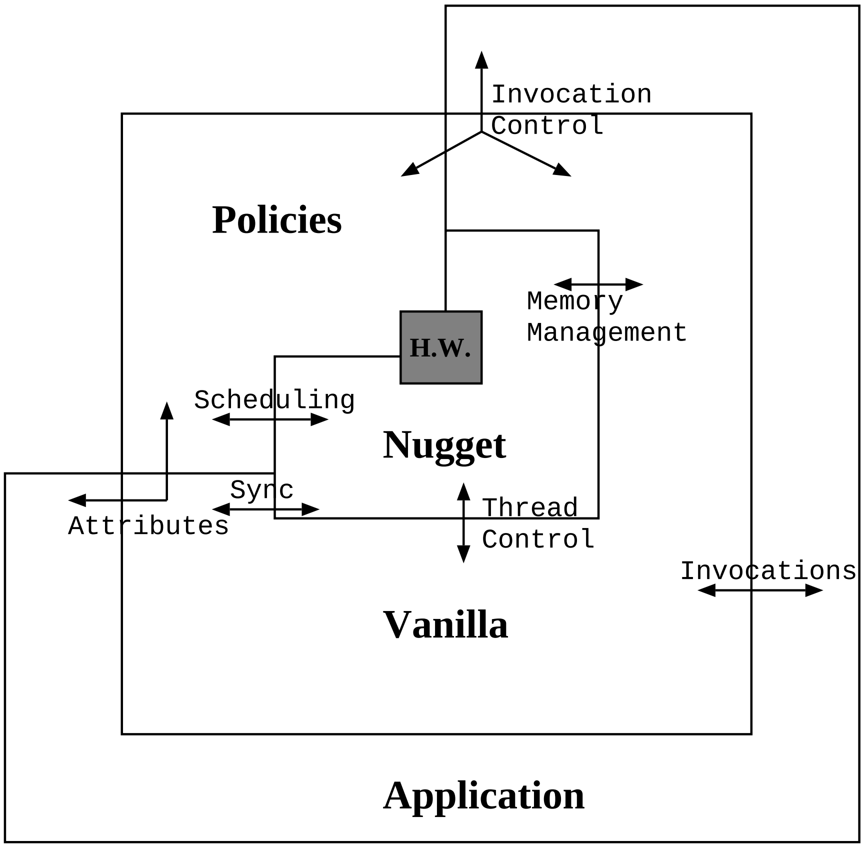

Figure 4: Structure of the CHAOS^{arc} kernel.

structed with CHAOS^{min} addresses the predictable execution of highly dynamic real-time programs. They are called as the CHAOS^{arc} kernel. CHAOS^{arc} permits the reliable execution of real-time software, where (1) computations must complete within well-defined timing constraints typically captured by execution *deadlines*, and (2) programs must exhibit predictable behavior in the presence of uncertain operating environments. (2) is achieved by provision of operating system constructs that may be used to *guarantee* desired performance and functionality levels of selected computations in real-time applications[GS89a] – termed *atomic, real-time computations*. These constructs implemented by the CHAOS^{arc} object-based operating system kernel provides constructs that deal with uncertainty by allowing programs to be *adaptable* (i.e., changeable at run-time) in performance and functionality to varying operating conditions. The adaptations specifically supported by CHAOS^{arc} constructs are those that may be implemented as reactions to external events – termed *reactive adaptations*, as opposed to adaptations that anticipate changes in the operating environment – termed *preventive adaptations* [BS88, SBWT87, SGB87, GS89b]. However, programming and monitoring system support is implemented for CHAOS^{arc} so that preventive adaptations may be performed as well.

On-line monitoring. Another issue addressed by the CHAOS^{arc} researchers is the application-specific, on-line monitoring of running real-time programs. The purpose of such monitoring is to use monitor data to adapt running programs in performance and functionality to changing external execution environments. The ideas presented in [KSO90, KS91, OSS90] are now being integrated into the lowest layers of CHAOS^{arc} , thereby permitting even the implementors of specific abstractions at the threads or object levels[Muk91] to configure object implementations during program execution.

The CHAOS^{arc} system runs on multiple platforms due to its use of real-time threads (see section 4.2.5) as a lower layer. While the systems are not intended for commercial use, offshoots are being used in commercial robotics applications[BG92].

Scheduling. Scheduling in CHAOS^{arc} may be performed both at the object level, by use of attributes and policies, and at the threads level. The real-time threads underlying the system are described in section 4.2.5. The dynamic multiprocessor scheduling policies and implementations are described in [Zho92].

4.4.4 Ongoing Efforts

Several current efforts in industry concern the development of object-oriented operating system kernels. At Microsoft Corporation, researchers are planning to use notions similar to CHAOS^{arc} attributes to capture timing information and make it available to real-time scheduling policies[Jon93a, Jon93b]. Furthermore, reflective programming results are being applied to operating system kernels to result in the highly configurable system kernels required for future real-time hardware [SR94]. Last, object-oriented layers are being constructed for task-based real-time systems, such as the object support offered for real-time Mach.

4.5 Real-Time Network Communications

A relatively recent topic in real-time systems research concerns real-time communication protocols. As distributed real-time systems become increasingly popular due to inexpensive processing capabilities, and ease of extensibility, the real-time requirements on the communication media must be analyzed to ensure that end-to-end deadlines are satisfied. Several hardware properties have been identified as useful for implementation of real-time communication protocols on a LAN[Ver93]. They include bounded transmission delays for uncorrupted messages[13], bounded omission degree and bounded inaccessibility of communication medium.

While traditional communication protocols deal with failures, real-time communication protocols have to take into account end-to-end communication times and the support of high-priority, 'out of band' messages. Worst-case latencies (for messages destined for hard-deadline tasks) have to be analyzed for traffic patterns in different priority classes. This is usually done in an iterative process, the desired latency leading to modifications of parameters of the protocol/communication medium, which in turn modify latencies, which may necessitate modification of the application's communication demands.

A running continuous media application, such as full motion video, can occupy significant bandwidth of the computer resource. Although some compression schemes such as JPEG[Wal91], MPEG[Gal91], and px64[Lio91] have been suggested to reduce data size, high quality video frames are usually too large for a conventional local area network[TTCM92]. For asynchronous applications like interactive video/teleconferencing, end-to-end delay has to be bounded and observable jitter should be avoided [TTCM92]. Because of these temporal and spatial constraints, continuous media communication requires special resource management[ATW+89]. In order to overcome such spatial and temporal constraints of continuous communication media, a few transport protocols such as ST-II (Stream Protocol II)[ea90], SRP (Session Reservation Protocol)[ATW+89], XTP (Express Transport Protocol)[CA90],

[13] *Tightness* – that destinations which actually receive an uncorrupted message, receive it within a bounded delay of each other – follows from this property.

VMTP (Versatile Message Transport Protocol)[Che87], and fast lightweight transport protocols have been proposed. These protocols can be divided into two classes: reservation and non-reservation based protocol[TTCM92]. The ST-II, and SRP protocols reserve system resources such as processor execution time, buffers, and network bandwidth before transmitting any data. A similar resource reservation model, a real-time channel, has been proposed for a wide area network environment[FV90]. Such reservation of resources requires significant operating system support. On the other hand, VMTP and XTP transfer data on a best-effort basis and without any resource reservation. However, they do not guarantee on the end-to-end delay or jitter bound for a session[TTCM92].

Recent research in this area has resulted in the construction of a real-time IP protocol, called RTIP[Zha91]. In [TTCM92], Tokuda et. al. present a Capacity-Based Session Reservation protocol(CBSRP), which provides guaranteed end-to-end delivery of data through resource reservation in a local area network environment. CBSRP differs from ST-II and SRP in its capability of changing quality of service (QOS) parameters of a session dynamically. Other approaches investigate the use of TDMA (see section 4.3.5) and related protocols (e.g., the Timed-Token protocols [MZ94]) which use a token-ring based approach, instead of the ethernet-style 'exponential backup based' approaches, which are not useful from a predictability standpoint. In the timed-token protocol, a circulating token ensures that each node is guaranteed to receive a certain fraction of the network bandwidth. A protocol parameter called the *target token rotation time* (TTRT[14]) is the key parameter which is adjusted, along with the amount of hard real-time message-bandwidth, and buffer sizes, in the timed-token protocol [MZ94]. A node transmits its hard real-time messages when its share of the network bandwidth comes up. If any time is left unused after a node completes transmitting its hard real-time messages, the rest of its share is used to transmit the soft real-time messages. This approach is much like the one used in MARS [DRSK89].

Malcolm and Zhao have recently performed a thorough survey of hard real-time communications on multiple-access networks [MZ93].

4.6 Real-time Implications on Computer Architectures

As discussed in section 4.2, the predictable execution of real-time software often precludes the use of mechanisms that enhance average-case program performance. However, OS and architecture modifications similar to those performed in non-real-time systems can achieve comparable performance improvements. Here, we discuss some examples of the synergism between the architecture and the OS as found in real-time systems: how scheduling approaches can cause the development of new architectures (e.g., co-processors for scheduling, partitioning caches to hide the effects of frequent task preemption), how architectural idiosyncrasies can result in a different approach to scheduling (e.g., ascertaining close upper-bounds for worst-case task execution times in the face of caching), and how integrated architectures are being developed to support specific real-time applications (e.g., to provide architectural support for predictable, time-constrained communications).

Caching. While caches are known to enhance performance in computer systems, designers of real-time systems have seldom used all the benefits of caching. This is because worst-case task execution time (WCET, henceforth) has to be taken into account in real-time schedulability analyses, and caching, with cache-misses, adds an unwelcome degree of unpredictability to the system. This problem arises chiefly from cache-reload transients, since real-time systems typically have frequent task preemptions and context switches.

[14] The TTRT is the expected time the token takes to rotate through the system.

The use of caching in real-time applications is examined, among others, in [Kir89, KS90], where the strategy for dealing with cache effects is the use of *a-priori* knowledge of program behavior (the periodic tasks, at least) to partition an instruction cache into static and dynamic parts [Kir88]. The dynamic cache part uses an LRU replacement policy. While this approach usually produces a sufficiently high predictable hit-rate (see the simulation results in [Kir88]), the author admits that it will not produce much improvement in the hit rate in two situations: (1) when the application has a high number of misses due to instructions being executed very infrequently, and (2) when the application has a large number of hits due to instructions that have a distance[15] just smaller than the size of the cache.

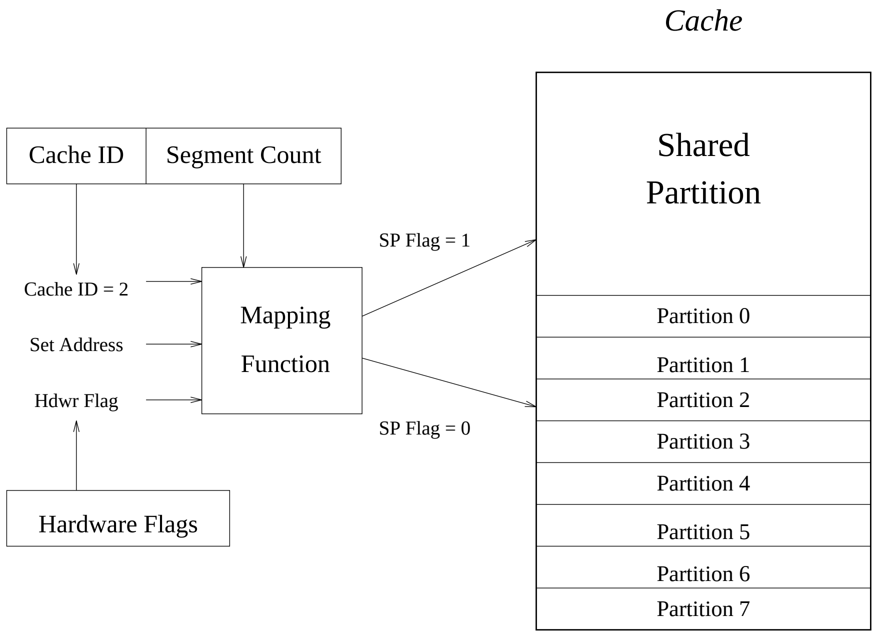

Figure 5: SMART Cache Partitioning

This two-way partitioning scheme is generalized in [Kir89] and [KS90]. In the first of these papers, a partitioning scheme is presented which combines several beneficial features of previously known schemes (N-way partitioning, static and dynamic locked partitioning), called the Strategic Memory Allocation for Real-Time (SMART) cache design scheme. Here, the cache is divided into M+1 segments: one segment – a shared-pool – is the largest; the other M segments are roughly of equal sizes, and are divided among the most critical real-time tasks. Each non-shared segment can be accessed only by the task (or preemption group) to which it is allotted: hence, cache footprints are protected across preemptions. Shared data is allocated in the shared segment. Private partitions are suggested for different interrupt levels, and for groups of aperiodics. In [KS90], the authors discuss the implementation of these techniques on a MIPS R3000 RISC processor.

Designers of the Spring system have devised another way to attain somewhat predictable behavior when using instruction caches[NNS91]. First, it is argued that a task in Spring never blocks, because Spring's design isolates application code from external interrupts[16], and because tasks are scheduled taking their resource requirements into account. Second, Spring uses virtual address instruction caches. Fetching instruction blocks into the instruction cache (on a miss) reduces several later instruction-fetch penalties. The WCET for an instruction block – in the presence of caching – is ascertained from the time graph[17] by noting the number of bytes in the basic block, and by identifying the speedup due to

[15] The distance of an instruction is the number of distinct instructions accessed between two successive accesses of that instruction.

[16] In Spring, external interrupts can arrive only at the I/O subsystem, and they are handled using a special scheduler on a processor in the I/O subsystem [SR89]. The application runs on other processors, using another scheduler in the kernel, while a third scheduler schedules OS tasks on the 'system processor'. The system processor and scheduler, thus, protect the application from frequent interrupts from the front-end.

[17] The time graph for a basic block is generated from the corresponding basic block when Spring performs the program translations necessary to transform the process-oriented programming model to one that can be handled by the real-time schedulers in Spring [Nie89].

instruction prefetches in the linear portions of the basic blocks. A similar method is employed when instruction loops[18] without branching fit completely into the cache. Subroutine calls are inlined and virtual addresses are assigned to the loop and the 'contained' subroutine code.

Measuring Task Execution Times. In general, the determination of task execution times is difficult due to data-dependent branches, resource-sharing, etc. Some efforts have been made to model the structure of such applications as task graphs, and to use real-time logic [JM86] and distribution functions for specific types of fork-join structures [Woo86] to analyze their execution times. We discuss some work in this area which uses dedicated hardware to acquire monitoring data in real-time, and then uses such data to obtain close approximations of WCET.

In [HS91a], Haban and Shin describe the use of dedicated hardware – which they call *Test and Measurement Processors* (TMP) – to measure execution time with minimum interference with the host system[19]. Considering a model where delays can occur through sharing of resources (unlike the Spring model), the authors measure the 'pure execution time' and resource-sharing delay of a task, and use both to reduce the difference between actual and estimated WCET. A task is divided into several disjoint parts. Initially, the worst-case execution time of each part is known. The WCET for the whole application is the sum of these individual worst-case execution times. As the application progresses, the TMP measures the resource sharing delays, and the actual elapsed execution times of the individual parts (as the processing of these parts completes). The worst-case execution times of the individual parts are replaced by their actual elapsed execution times. The sum of the actual execution times of the finished parts and of the worst-case execution times of the unfinished parts produces the *Anticipated Execution Time*. This is a much closer approximation of actual task execution time than the WCET. If at any time during application execution, it can be determined that the sum of the resource sharing delays and actual execution times of the finished parts has become too large to allow the whole task to complete in time, the task can be aborted immediately. Of course, once the WCET is updated, or a task is aborted, a reschedule has to be computed for the remaining tasks. Resource-reclaiming strategies have been suggested [SRS93] to obtain such reschedules without redoing the entire schedule for all remaining tasks (the approach in [SRS93] has a bounded overhead). The research described in [HS91a] and discussed in this paragraph builds in part on earlier work at Honeywell which concerned the construction of monitored network hardware used for support of distributed real-time applications[MS82].

Supporting distributed execution platforms. The HARTS architecture [Shi91] was designed to support the special requirements of a distributed real-time system (see section 4.3.4) – among them, the communication requirements. Clock synchronization algorithms and hardware assure the concept of a global time base across the nodes, which is needed for checkpointing, IPC and allocation of (shared) resources. Among the components of a HARTS node is a *network processor* (NP) – a custom-designed interface to the interconnection network, the nodes in HARTS being connected in a hexagonal topology[CKD90]. The NP contains hardware for clock synchronization and timestamping. The NP also has support for multiple interrupt levels (for messages of varying priority), and for resource preemption, and monitoring traffic and hosts' load for real-time load-balancing, and fault-tolerance. Most of these functions are provided for by the *Interface Management Unit* of the NP. HARTS uses a clock synchronization algorithm based on [LPS85]. However, three problems have to be solved before this approach can be used: (1) remote clock values have to be obtained, (2) clock values are 'transmitted' in the original algorithm: in a faulty set-up, a the clock value of a good node may be corrupted by a transmitting node, and (3) queueing delays in processing the clock values obtained are random; hence, a plain subtraction of clock values (see the original algorithm by Lamport and Melliar-Smith mentioned

[18] An upper bound on the number of times a loop is executed must be stated in the Spring programming model.

[19] See [HS91a] for details about the monitoring hardware and the instrumentation and monitoring principles.

above) will not suffice.

HARTS addresses (1)-(3) [RKS90] by broadcast of clock values at a specified time, as needed, thereby solving (1). This broadcast delivers multiple copies through node-disjoint paths, thereby solving (2). In order to solve (3), each intermediate process in HARTS is required to append the message delay at that process when it transmits a message[20].

The use of special-purpose hardware to for time-adjustment, clock synchronization and timestamping in MARS has already been mentioned in section 4.3.5.

Specialized Co-processors. The use of a co-processor for scheduling real-time tasks in the Spring operating system[RSS90] is described in [NRS+93]. The Spring scheduling co-processor (SSCoP) can be used for static scheduling, as well as for dynamic, on-line scheduling. It also permits the use of different scheduling algorithms within each class. The SSCoP contains registers that are accessible to the host via memory mapping. The host writes task attributes (deadlines, execution times and arrival times) into the SSCoP before initiating the scheduling operation. There is a pre-processing phase, during which SSCoP task slots are allocated to the tasks that need scheduling, and the resources that are needed by these tasks. Precedence relations between tasks are set during this time; they are stored as bit vectors in the SSCoP slots for the particular tasks. If the task(s) can be guaranteed, the host can read back a feasible schedule from the SSCoP. This involves some post-processing, during which (1) the starting times of the scheduled tasks (the indices of which can be obtained from the output queue of the SSCoP) have to be read from certain registers in the SSCoP, (2) the *absolute* start and finish times are calculated from the *relative* start times obtained from the co-processor, and (3) the entry for the scheduled task is added to the end of the appropriate dispatch queue. Preliminary measurements reported in [NRS+93] indicate that use of the SSCoP "speeds up the overall scheduling operation 30 fold."

5 Standards and Performance Evaluation Techniques for Real-Time Kernels

This section concerns standardization efforts and techniques for the evaluation of real-time kernels and executives.

5.1 Standards for Real-Time Operating Systems Support

The notions of kernel 'families', or attributes and policies, of subcontracts, and of 'reflective programming' are being advanced by the object-oriented community in order to achieve truly configurable real-time and non-real-time operating systems. At the same time, industry efforts concern the development of standards for large-scale real-time systems. Notable among these standards are the TRON effort in Japan and POSIX Unix in the U.S. In addition, manufacturers are developing standards for real-time communications for use in multi-media applications. This section reviews the TRON 'framework' effort in Japan and the POSIX Unix-based real-time standards effort now proceeding in the U.S.

[20] Hardware support is provided to compute this delay accurately: on receiving a clock message, a receive time stamp is appended to the message; on being transmitted, a transmit timestamp is appended. At intermediate nodes, the receive and transmit timestamps use the same local clock. Thus, their difference gives an accurate estimate of message time in that node.

5.1.1 POSIX

Derived from Unix, the IEEE POSIX (Portable Operating System Interface for Computer Environments) standards (also see section 1) are an effort to control the multitude of interfaces available in OS products. POSIX 1003.1 deals with the base operating system functionality/interface, 1003.2 with the commands set, 1003.3 with testing of the standards, 1003.4 with source code portability for real-time, and 1003.4a with threads extensions. In the next few paragraphs, we briefly discuss some of the features for real-time support in 1003.4.

The interface allows a process to change its scheduler. POSIX allows applications to be multi-threaded, and different threads of an application (even different threads in the same process) may use distinct schedulers. There are calls to set the scheduler to one of a set, and to incorporate new scheduling policies. The system must support at least the FIFO and round-robin policies. Threads can have priorities, and are fully preemptive. There is support in the 1003.4 framework for reliable delivery of signals, which are queued. Timers are of nanosecond resolution, with support for absolute and relative timers. The standard allows for asynchronous I/O, with a reliable signal (mentioned earlier) being delivered when the I/O completes. It also allows for synchronized I/O: unlike in non-real-time systems where writes might be placed in buffer caches to 'complete' at a later time, synchronized I/O calls do not return until the output completes physically. There is a rich message-passing facility under the auspices of 1003.4, and support for binary semaphores for process synchronization.

In [GL91], Gallmeister and Lanier state their experience with implementing POSIX 1003.4 and 1003.4a over LynxOS – a Unix-compatible real-time operating system. While their experience was that even with a real-time version of Unix, they had to rewrite some of the facilities, the authors largely opine that the POSIX standards are attractive. Not only did the use of the standards help in determinism of the system, but the performance was at least as good as that using equivalent Unix or proprietary interfaces.

5.1.2 TRON

TRON [Sak89a, Ass92] (The Real-time Operating-system Nucleus) is under development as a specification of operating system interfaces. The guiding principles of TRON are to have a highly functionally distributed system with outstanding real-time response, a uniform basic operating method[21], specification for a high-performance single-chip processor, and having the TRON specification freely available and encouraging free competition in product implementation. TRON consists of several subprojects: ITRON being the real-time multitasking OS specification for industrial and other embedded systems, TRON-CHIP being the specification for a processor, BTRON being the OS specifications for workstations and personal computers used in business: focussing on smooth human-computer interactions, CTRON the OS interface specifications for communication and information processing, and MTRON the 'attached operating system architecture' to link systems based on the TRON architecture.

A series of ITRON specifications have been published and implemented. They include μITRON [FYTY92], which are targeted to low-cost microcontrollers and microprocessors, ITRON1 and ITRON2. One of the key features of ITRON is system configurability [Sai92], in order to optimize the OS for each target system. Standardization increases from μITRON to ITRON1 to ITRON2, while flexibility (configurability) decreases. The specification of the functionality in TRON is layered. *Basic Functions* comprise synchronization and communication (events, semaphores and mailboxes), task, time, memory,

[21] This means that operators trained under one TRON system need not undergo retraining to use a different model which adheres to TRON.

exception and interrupt management. *Extended Functions* are optional and consist of primitives for message buffering and communication through ports, resource management support, time handler and local memory pool management facilities. *System Control Functions* and *Utility Module* are layered over the aforementioned functions, and comprise the debugging support, some more exception management, device drivers and file management routines. ITRON was later extended to support distributed and multiprocessor systems [TS92].

CTRON has certain 'optional parts' to support real-time processing. Thus, cyclic procedures can be defined, activated and cancelled. There is support for a rendezvous mechanism, selective message receiving, and creation, deletion and query of timers which are under the control of specific tasks. In addition, there is support for locking memory objects. Section 4.3.6 discusses CTRON in more detail.

5.2 Performance Evaluation Techniques for Real-Time Kernels

Most thorough performance evaluations of real-time operating systems have relied on system evaluation with specific target applications. Unfortunately, it can be difficult to generalize such evaluations to other applications or target architectures. As a result, researchers have attempted to formulate more general methods for system performance evaluation. Three general methods have been devised, differing in the degree of detail at which they evaluate the target system. This section reviews some common methods of performance evaluation for real-time operating systems.

Fine-grained benchmarks or measurements investigate a real-time executive at a very low level, evaluating the efficiency of the hardware and software for the most frequently used primitives of the real-time kernel. Based on the performance of these primitives on the particular hardware, a figure of merit is assigned to the hardware-software combination. One problem with such benchmarks is that for very accurate measurements, one may have to resort to special-purpose hardware; otherwise, the experiments may not be repeatable – rendering the benchmarking process useless. Rhealstone (section 5.2.1) is the best known fine-grain real-time benchmark. Various other fine-grain benchmarks have also been suggested. In [FGG+91], e.g., the authors describe a 'tri-dimensional measure' of real-time performance, based on CPU speed, interrupt handling capability and I/O throughput. The 'equivalent MIPS' of a real-time system is then calculated as the cube-root of the product of the measures in each of the individual 'dimensions'.

Application-oriented benchmarks take a much higher-level look at a real-time executive, in terms of the number of deadlines kept or missed, and the utilization point at which the system begins to break down. They are often implemented as synthetic applications running on the real-time executive – synthetic in the sense that their deadline requirements are drawn from certain distributions, rather than being synthetic in terms of 'instruction mix' as in [Wei84]. Based on the resulting performance with respect to the number of deadlines the hardware-software combination can keep, a figure of merit is assigned. Hartstone (section 5.2.2) is the best known application-oriented real-time benchmark suite.

Simulation-based evaluations model a real-time system at some appropriate level of detail, and then implement and run the model. The advantage of this approach is that the hardware itself can be modelled in conjunction with the software. This permits developers to understand performance properties of a real-time kernel on hardware that may not yet be fully developed. It is important, of course, that the simulation faithfully models the simulated system, which is also the case for the application-oriented benchmarks modeling the behavior of certain real-time applications[22]. The disadvantage of this method is that it is difficult, indeed impossible, to account for all of the idiosyncrasies

[22] See [Jai91] for an excellent treatment of various aspects of modelling and simulation for performance-evaluation.

of an actual real-time system. Even if it were possible to develop such models, programming them would be difficult, and executing the simulation time-consuming.

Combinations of these techniques are also used. In [FGG+91], e.g., the authors describe a performance measure which they call the 'Real/Stone' benchmark. This essentially combines the 'tri-dimensional measure' with a simulation-based approach.

5.2.1 Rhealstone

The Rhealstone benchmarks [KP89, Kar90] are used to obtain a figure of merit for low-level activities of a real-time kernel on some target hardware, thereby evaluating some hardware-software combination. A 'Rhealstone figure' is a weighted sum of six categories of activity: task switching time, task preemption time, interrupt latency time, semaphore shuffling time, deadlock breaking time and datagram throughput time[23]. The designers of these benchmarks posit that every real-time application is unique and that a 'statistical-mix' of operations like Dhrystone [Wei84] is not relevant in real-time systems. Instead, more basic numbers must be measured.

The Rhealstone figure – calculated as a figure of merit – is the sum of the reciprocals of the times for the five categories (all but the datagram throughput, which is added as such) obtained in seconds. Depending upon the expected relative frequency/importance of the categories, the individual numbers may be 'weighted' to obtain a weighted sum as the Rhealstone figure[24].

The Rhealstone benchmark's figure of merit characterizes the kernel-hardware combination and is almost independent of the benchmark (unless a weighted sum is used)! However, Rhealstone has some serious drawbacks. First, deadlines, which are of primary importance in real-time systems, are not accounted for. Essentially, the benchmark evaluates operation speed instead of taking into account *real-time*. Second, a single figure may not be obtainable for 'preemption time', since preemption can be quite complicated (due to priority inversion, etc.) in a multitasking real-time system. For example, the authors state that 'prioritized semaphore queues' (basically, signalling the highest priority task waiting on a semaphore) are important [KP89], but do not take them into consideration in the benchmarks. Third, the 'deadlock-breaking' criterion loses importance in the face of priority inheritance protocols used by well-designed schedulers. Finally, the six measurement categories are somewhat ad-hoc[25].

5.2.2 Hartstone

Originally designed for uniprocessor real-time systems, the Hartstone benchmarks [WK92] consist of periodic and aperiodic tasks, and of additional tasks to represent synchronization points in the application. Periodics have hard deadlines, and aperiodics can have hard or soft deadlines. Aperiodics with soft deadlines run as background tasks, and an optimal schedule tries to minimize their turnaround time while also maximizing the number of hard deadlines kept. The benchmarks are useful for evaluating a system as a whole; components of a system can be compared using these benchmarks only if the component can be changed while keeping all other components identical.

Initially, there is a *baseline task set*, which is characterized by the number of tasks in it, their deadlines, periods and interarrival times. The experiments in the benchmark proceed in steps, one parameter being changed at each step while the other parameters are held constant. The various experiments – derived from the Rate Monotonic and Earliest Deadline First algorithms (see section

[23] While the original proposal of the Rhealstone benchmarks [KP89] used datagram throughput time, a later implementation by one of the authors [Kar90] uses inter-task message latency instead.

[24] While the initial proposal [KP89] suggested a 'sum of reciprocals' as the Rhealstone figures, a later implementation by one of the authors [Kar90] uses the reciprocal of the mean of the component categories as the final Rhealstone performance number.

[25] Hartstone, e.g., gives a motivation and a derivation for the 'categories' in the uniprocessor benchmarks in Hartstone [WK92].

2)[26] – use a mix of tasks consisting of (1) all periodics with frequencies that are multiples of the smallest frequency (harmonic periods), (2) periodics with non-harmonic periods, (3) combination of periodics and aperiodics, (4) periodics with harmonic periods and a synchronization task (with which the other periodics must synchronize at certain intervals), and (5) a combination of periodic, aperiodic and synchronization tasks. The execution times[27], deadlines, blocking times, and number of tasks are varied appropriately in step in each experiment, until deadlines start to be missed, or until other stopping conditions become true – like response time becoming too large. The point at which this happens is a figure of merit for the hardware-software of the real-time operating system under consideration (the total *Kilo-Whetstone instructions per second*, or KWIPS). Apart from the figure of merit, a careful analysis of this 'breakdown point' may result in improved performance in the real-time executive.

These uniprocessor benchmarks were later extended to take into account distributed real-time systems [KW91][28]. The figure of merit considers the system software and hardware of the local host, communication software, the communication hardware of the node, and the physical channels and protocols. The distributed system may be homogeneous or heterogeneous. The benchmark assumes that each application consists of a set of tasks on a host processor, which might have to communicate with a remote node through a communication server (CS). One CS runs on each host and is implemented as an aperiodic task. Each message has a deadline which is identical to the deadline of the receiving task.

In addition to evaluating the processor-scheduling ability of the real-time system, the benchmark tests the performance of the CS. The baseline task set consists of three periodic tasks on each of the sender and recipient nodes. Periods of the harmonic periodics are in the ratio 2:4:8; those of the non-harmonics in the ratio 3:5:7. The smallest of the periods is 1 second. The execution time of each task consists of the time to send and receive all its messages as a minimum, and may have an extra synthetic workload factor. Sets of experiments are designed to test the datagram/virtual circuit/integrated protocol services, aperiodic communication (activated by internal/external events), and channel access protocols.

Three experiments are defined for each set. The first one increases the length of the messages sent or received. The second experiment increases the total number of messages by increasing the number of messages sent/received per task per period. The third experiment increases the total number of messages, by increasing the number of tasks sending/receiving messages (while each task sends/receives one message per period). These experiments produce figures of merit for maximum message lengths that can be sustained before communication deadlines are missed and for the maximum number of messages that can be scheduled before the scheduling subsystem fails to operate in a timely fashion.

5.2.3 Simulation

Simulation has been used effectively for performance evaluation of both implemented and unimplemented computer systems. The system simulation process consists of model formulation, model implementation, and performance evaluation based on collected statistics. Such system simulations tend to be extremely time-consuming, in part because initial models often have to be refined and then re-run and in part because simulators are slower than the target system being modeled. For example, simulating the effects of a single instruction fetch-execute on some proposed architecture might involve several instructions on another 'real' computer. In this regard, discrete-event simulations (DESs) often perform better than timestepped simulations, since DESs model the *changes* in the state of the system,

[26] [WK92] provides a 'derivation' of the experiments.

[27] A Hartstone task executes a specified number of Kilo-Whetstones for each instantiation, a Whetstone instruction being close to one floating-point operation

[28] [MIT90] presents a distributed version of Hartstone, and reports the performance of the ARTS kernel using that benchmark, but that design, at least as reported in [MIT90], appears to be somewhat less thorough than the one in [KW91]. Therefore, we consider only the latter work in this survey.

while timestepped simulations just model the system – irrespective of change – at suitable intervals.

One way to speed up such simulations is through parallel execution [Fuj90]. However, synchronization constraints, and constraints due to orderly access of state information at runtime, play a very crucial role in these simulations. It is in general impossible to predict the dependencies between events. Therefore, if one takes a conservative (worst-case) approach to parallelizing these simulations, performance may not be much better than that afforded by serial execution.

An optimistic approach has been proposed for parallelizing these simulations [Jef85]. While the conservative approach blocks the execution of a simulation event until it becomes certain that data-dependence constraints will not be violated, the optimistic approach aggressively performs the event computations, detecting and recovering from incorrect computations[29] using a 'rollback' mechanism as such errors are discovered. Several successes have been reported in speeding up simulations using the optimistic paradigm; however, they have generally been in non-real-time domains.

The essential problem with optimistic *real-time* simulation is that the simulator may have to 'keep up' (in terms of real-time) with an external device or human person (as in hardware-in-the-loop or man-in-the-loop simulations). This can be the case while evaluating the performance of a real-time system too: certain components might be 'real', while the performance of the other 'incomplete' components might have to be simulated partly or fully, and the performance of the system evaluated as a whole. Recently, researchers [GFS93a, GFS93b, GPFS93] have derived the conditions under which optimistic real-time simulation is possible, and have reported the preliminary performance of such a simulator.

5.3 Evaluating Large-scale Real-time Systems

Many issues remain concerning the evaluation of real-time systems, in part because such systems range in complexity from single-CPU embedded architectures in simple robots or manufacturing machines to large-scale distributed systems spanning entire cities (e.g., traffic control, video on demand, etc.), manufacturing plants, or even entire countries (e.g., air traffic control, satellite data acquisition and distribution, etc.). Therefore, we conclude this section by listing some of the attributes of next generation real-time systems that should be addressed by future real-time benchmarks. Benchmarks should:

- capture of real-time vs. basic performance properties of large-scale distributed or parallel applications, such as system predictability and timeliness,
- evaluate the mechanisms offered by systems to maintain high levels of predictability and/or performance for specific application programs, such as mechanisms for system monitoring and configuration, support for multiple timing semantics, task and communication models, system adaptability, and fault tolerance,
- address system support for dealing with external devices, such as networks, high performance I/O devices and sensors, and
- evaluate a system's ability to evolve, which may be tested in part by system configurability not only to improve performance but also to deal with issues of system interoperability.

[29] An event computation may be incorrect if that event was executed prematurely.

6 Conclusions and Future Work

This review has summarized current and recent research in real-time operating systems. We have selected certain key research systems for detailed review, rather than simply listing the large number of existing or past, research and industrial real-time systems. As a result, this survey is necessarily incomplete.

It should be apparent from this survey, however, that much research still remains to be done in real-time operating systems. Specifically, we believe that future work must address the topics of dynamic, physically distributed, and large-scale execution environments, often employing highly parallel and heterogeneous computational nodes. Clearly, such systems are already being addressed by research in many areas of Computing. The real-time community must focus on the 'real-time' nature of such systems, along with investigating the new classes of real-time applications and their timing constraints being constructed for these domains. Such shifts in focus from single embedded and relatively static to dynamic and distributed systems is already apparent in current research on multimedia applications and communication protocols, but the investigation of appropriate semantics for real-time constraints in multimedia applications is still in its infancy. Similarly, while research has resulted in many useful algorithms for off-line scheduling, the dynamic scheduling in large-scale complex systems must be investigated further, in conjunction with user support for such scheduling, mechanisms for on-line monitoring, and means of system configuration in response to exceptions or to improve reliability and performance.

Several bases for real-time computing are currently being developed, including Real-Time Mach in academia and by the U.S. Department of Defense, real-time POSIX Unix, and many commercial systems. Unfortunately, wide differences exist among these systems and convergence to a single system appears unlikely. A joint system may arise, however, from current and future work on object-oriented operating systems, which are now being developed by several industrial concerns, including Taligent (the cooperation between IBM and Apple Computer Corps.) and Microsoft. However, these commercial efforts are only now beginning to address real-time computing and applications.

Some opportunities for improvement and an agenda for research in core Computer Science have been chalked out in [HL92]. It is interesting to note that almost all of the items for research mentioned therein can also be applied to the realm of real-time systems. This is because real-time computing tends to incorporate several disparate disciplines of Computer Science. This implies that research in real-time computing often mirrors general research in Computer Science [HK89], including increasing systems speeds (10^3 MIPS and beyond) and parallelism (over 1000-fold for certain applications), improving system reliability, and dealing with physical system distribution. For acceptable performance, parallelism has to be expressed explicitly, and architecture-specific optimizations need to be performed in today's parallel programs. Furthermore, in large-scale distributed systems, besides addressing issues in real-time communication, researchers have to be concerned with the integration of a heterogeneous computing environment into single real-time systems able to function such that some global guarantees of performance or reliability can be made. As an example, consider applications from the entertainment industry utilizing large-scale telecommunication networks. This implies that real-time systems must be scaled from current, local-area networked systems, to addressing the size of the 'Internet-community' (under ten million at present), to (potentially) addressing 'every household' (billions!). It can be safely predicted that such exciting real-time applications will require the solution of many future research problems. Moreover, as real-time systems become more complex, it will not be practical to develop new systems from first principles. Instead, techniques must be found to re-map existing software into the newer systems, such that new systems can deal with new execution environments in a predictable, efficient, and reliable fashion. This is especially important in environments like medical applications where human lives may be involved, or in military scenarios where system malfunctions can endanger millions. The development and maintenance of complex systems require that researchers develop tools for automatic system integration, monitoring, scheduling, verification and testing, and reconfiguration.

Acknowledgements. The authors gratefully acknowledge the comments of Hermann Haertig, Kevin Jeffay, Ragunathan Rajkumar, Krithi Ramamritham, and Jennifer Rexford on specific sections of the paper. Prabha Gopinath provided feedback on an earlier version of the paper.

References

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Transactions on Computer Systems, ACM*, 10(1):53–79, February 1992.

[AC86] Steven Anderson and Marina C. Chen. Parallel branch-and-bound algorithms on the hypercube. In *Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee*, pages 309–317. Oakridge National Laboratories, Sept. 1986.

[Agr90] A. Agrawala. Systems engineering approach to time-driven systems. In *CompEuro 90*, 1990.

[AL87] A. Agrawala and S. Levi. Objects architecture for real-time, distributed, fault tolerant operating systems. In *IEEE Workshop on Real-Time Operating Systems*, July 1987.

[And90] T. E. Anderson. The perfomance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[Ass92] TRON Association. Bibliography of the tron project (1984 - 1992). In *Proceedings of The Ninth TRON Project Symposium (International), 1992*, pages 210–235. IEEE Computer Society Press, December 1992.

[AT87] Jr. Avadis Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Systems: The Mach Approach.* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., December 1987.

[ATW+89] D. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for continuous media in the dash system. Technical report, University of California, Berkeley, October 1989.

[Bak87] T. Baker. A corset for ada. Technical Report 86-09-07, Department of Computer Science, University of Washington, 1987.

[Bak90] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1990.

[BBH84] F. Baccelli, P. Boyer, and G. Hebuterne. Single server queues with impatient customers. *Advances in Applied Probability*, 16:887–905, 1984.

[Be85] Kenneth P. Birman and et.al. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, pages 502–508, June 1985.

[BG92] T. E. Bihari and P. Gopinath. Object-oriented real-time systems: Concepts and examples. *IEEE Computer*, 25(12):25–32, December 1992.

[BHI93] Sara A. Bly, Steve R. Harrison, and Susan Irwin. Media spaces: Bringing people together in a video, audio, and computing environment. In *Communications of the ACM*, volume 36 (1), pages 28–45. ACM Press, January 1993.

[BJ86] T. Baker and K. Jeffay. A lace for ada's corset. Technical Report 86-09-06, Department of Computer Science, University of Washington, 1986.

[BJ87] T. Baker and K. Jeffay. Corset and lace: Adapting ada runtime support to real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1987.

[BKLL93] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso. *Programming Under Mach*. Addison Wesley, 1993.

[BKM+92] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. In *The Journal of Real-Time Systems*, volume 4(2), pages 125 – 144. Kluwer Academic Press, June 1992.

[Bla89] Ben Blake. *A Fast, Efficient Scheduling Framework for Parallel Computing Systems*. PhD thesis, Department of Computer and Information Science, The Ohio State University, Dec. 1989.

[Bla90] David L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[BP91] T. Baker and O. Pazy. Real-time features for ada 9x. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

[Bri92] L. Briand. Time management for ada real-time systems. *Ada Letters*, 12(5), September 1992.

[BS88] T. Bihari and K. Schwan. A comparison of four adaptation algorithms for increasing the reliability of real-time software. In *Ninth Real-Time Systems Symposium, Huntsville, AL*, pages 232–241. IEEE, Dec. 1988.

[BS91a] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991. Older version available from the Department of Computer and Information Science, The Ohio State University, OSU-CISRC-5/88-TR, newer version available from College of Computing, Georgia Institute of Technology, Atlanta GA, GTRC-TR-90/67.

[BS91b] Ben Blake and Karsten Schwan. Experimental evaluation of a real-time scheduler for a multiprocessor system. *IEEE Transactions on Software Engineering*, 17(1):34–44, Jan. 1991.

[Bum93] Carl Bumeister. Real-time modelling in multimedia applications. In *Proceedings of Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems*, November 1993.

[Bur78] J. E. Burns. Mutual exclusion with linear waiting using binary shared variables. In *SIGACT News, Volume 10, Number 2*, Summer 1978.

[CA90] Protocol Engines Inc. CA. Xtp protocol definitions, revision 3.5. Technical Report PEI-90-120, September 1990.

[Can91] Ben J. Cantanzaro, editor. *The SPARC Technical Papers*. Springer-Verlag, 1991.

[Car84] Gene D. Carlow. Architecture of the space shuttle primary avionics software system. *Communications of the ACM*, 27(9):926–936, Sept. 1984.

[CC89] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[Che87] D. R. Cheriton. Vmtp: Versatile message transaction protocol. Technical report, Computer Science Department, Stanford University, January 1987.

[Chr93] Flaviu Chriatian. Talk at the first workshop on parallel and distributed real-time systems (does not appear in proceedings), newport beach, california. April 1993.

[CK88] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. In *IEEE Transactions in Software Engineering*, pages 141–154. IEEE Computer Society Press, February 1988.

[CKD90] M.-S. Chen, K.G.Shin, and D.D.Kandlur. Addressing, routing and broadcasting in hexagonal mesh multiprocessors. In *IEEE Transactions on Computers*, pages 10–18. IEEE Computer Society Press, Jan 1990.

[CL86] H. Y. Chang and M. Livny. Scheduling under deadline costraints: A comparison of sender-initiated and receiver-initiated approaches. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1986.

[CL90] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *The Journal of Real-Time Systems*, 2:325–346, 1990.

[CL91] M. Chen and K. Lin. A priority ceiling protocol for multiple-instance resources. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

[CMMS79] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. Thoth, a portable real-time operating system. *Comm. of the Assoc. Comput. Mach.*, 22(2):105–115, Feb. 1979.

[Cor90] Intel Corporation. *iRMK^{TM} Version I.1 Real-Time Kernel and iRMX^{TM} 286 Release 2.00 Operating System.* Intel Corporation, Santa Clara, CA, June 1990.

[CSR88] Sheng-Chang Cheng, John A. Stankovic, and Krithi Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. In *Tutorial Hard Real-Time Systems*, pages 150–173. IEEE, 1988.

[CT93] C. L. Compton and D. L. Tennenhouse. Collaborative load shedding. In *Proceedings of the IEEE Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems, Raleigh-Durham, NC*, November 1993.

[CW94] Soumen Chakrabarti and Randolph Wang. Adaptive control for packet video. In *Proceedings of the IEEE 1994 International Conference on Multimedia Computing and Systems, May 14–19, 1994, Boston, Mass.*, January 1994.

[DB88] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the 1988 National Conference on Artificial Intelligence(AAAI-88)*, pages 49–54. AAAI, AAAI, 1988.

[DJW93] G Dudek, M. Jenkins, and D. Wilkes. A taxonomy for swarm robots. In *Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Yokohama, Japan*, pages 441–447, 1993.

[DLJA88] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The clouds distributed operating system: Functional description, implementation details and related work. In *The 9th International Conference on Distributed Computing Systems, San Jose, CA.* IEEE, June 1988.

[DM89] Michael L. Dertouzos and Aloysius K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989.

[DRSK89] A. Damn, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. In *Operating Systems Review of the ACM Special Interest Group on Operating Systems*, pages 141–157. ACM Press, July 1989.

[ea90] Topolcic et al. Experimental internet stream protocol, version 2 (st-ii). 1990.

[Fer93] Edward E. Ferguson. Resource scheduling for adaptive systems. In *Proceedings of the IEEE Workshop on Real-Time Applications*, pages 102 – 103. IEEE Computer Society Press, May 1993.

[FGG+91] Borko Furht, Dan Grostick, David Gluch, Guy Rabbat, John Parker, and Meg Roberts. *Real-Time Unix Systems*. Kluwer Academic Publishers, 1991.

[FKRR93] Robert S. Fish, Robert E. Kraut, Robert W. Root, and Ronald E. Rice. Video as a technology for informal communication. In *Communications of the ACM*, volume 36 (1), pages 48–61. ACM Press, January 1993.

[FSW90] Jerry Fiddler, Eric Stromberg, and David N. Wilner. Software considerations for real-time risc. *COMPCON Spring 1990: Thirty-Fifth IEEE Computer Society International Conference*, pages 274–277, February 1990.

[FSW91] Jerry Fiddler, Eric Stromberg, and David N. Wilner. Software considerations for real-time risc. *The SPARC Technical Papers*, pages 305–313, 1991.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. In *Communications of the ACM*, volume 33 (10), pages 30–53. ACM Press, October 1990.

[FV90] D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communication*, 8(3), April 1990.

[FYTY92] Katsuhito Fukuoka, Akira Yokozawa, Kiichiro Tamaru, and Katsuyasu Yamada. A universal real-time kernel based on the μITRON specifications. In *Proceedings of The Ninth TRON Project Symposium (International), 1992*, pages 96–106. IEEE Computer Society Press, December 1992.

[Gal91] Didier Le Gall. MPEG: A video compression standard for multimedia applications. In *Communications of the ACM*, volume 34 (4), pages 46–58. ACM Press, April 1991.

[GBSG89] Prabha Gopinath, Tom Bihari, Karsten Schwan, and Ahmed Gheith. Operating system constructs for managing real-time software complexity. In *Proceedings of 1989 Workshop on Operating Systems for Mission Critical Computing, ONR, Maryland*, pages U1–U9. ONR et al, Sept. 1989. Also available as Philips Technical Note TN-89-110 and published by IOS Press, Netherlands, as 'Mission Critical Operating Systems, Studies in Computer and Communication Systems, Vol.1'.

[GFS93a] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. A testbed for optimistic execution of real-time simulations. *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1993. Also available as technical report GIT-CC-92/22.

[GFS93b] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. Time warp simulation in time constrained systems. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS)*, May 1993. Expanded version available as technical report GIT-CC-92/46.

[GGSW88] Ahmed Gheith, Prabha Gopinath, Karsten Schwan, and Peter Wiley. Chaos and chaos-art: Extensions to an object-based kernel. In *IEEE Computer Society Fifth Workshop on Real-Time Operating Systems, Washington, D.C.* IEEE, April 1988.

[GJ77] M. R. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM Journal of Computing*, 6(3), 1977.

[GK68] B. V. Gnedeko and L. N. Kovalenko. Elements of queueing theory. In *Israel Program for Scientific Research*, 1968.

[GL91] Bill O. Gallmeister and Chris Lanier. Early experience with posix 1003.4 and posix 1003.4a. In *Proceedings of the Real-Time Systems Symposium*, pages 190–198. IEEE Computer Society Press, December 1991.

[GMAT90] O. Gudmundsson, D. Mosse, A. Agrawala, and S. Tripathi. Maruti: A hard real-time operating system. In *Second IEEE Workshop on Experimental Distributed Systems*, pages 29–34, October 1990.

[Gop88] Prabha Gopinath. *Programming and Execution of Object-Based, Parallel, Hard Real-Time Applications.* PhD thesis, Department of Computer and Information Sciences, The Ohio State University, June 1988.

[GPFS93] Kaushik Ghosh, Kiran Panesar, Richard M. Fujimoto, and Karsten Schwan. PORTS: A parallel, optimistic, real-time simulator. Technical Report GIT-CC-93/71, College of Computing, Georgia Institute of Technology, Atanta, Georgia, December 1993. Submitted for publication.

[Gre88] I. Greif. *Computer-Supported Collaborative Work: A Book of Readings.* Morgan-Kauffman, 1988.

[GS89a] Ahmed Gheith and Karsten Schwan. Chaos-art: Kernel support for atomic transactions in real-time applications. In *Nineteenth International Symposium on Fault-Tolerant Computing, Chicago, IL*, pages 462–469, June 1989. Also see GIT-ICS-90/06, College of Computing, Georgia Tech, Atlanta, GA 30332.

[GS89b] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *SIGOPS Notices*, pages 106–125, July 1989. Also available as Philips Technical Note TN-89-006.

[GS93] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[HFC76] A. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19:266–72, 1976.

[HK89] John E. Hopcroft and Kenneth W. Kennedy, editors. *Computer Science: Achievments and Opportunities.* Society for Industrial and Applied Mathematics, 1989.

[HKL91] M. Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

[HL92] Juris Hartmanis and Herbert Lin, editors. *Computing the Future: A Broader Agenda for Computer Science and Engineering.* National Academy Press, 1992.

[HS91a] Dieter Haban and Kang G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. In *Proceedings of the Real-Time Systems Symposium*, pages 172–181. IEEE Computer Society Press, December 1991.

[HS91b] C. Hou and K. Shin. Load sharing with consideration of future task arrivals in heterogenous distributed real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

[HS92] Chao-Ju Hou and Kang G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 1992.

[Inc92] Tartan Inc. Evaluation of the implementation of multi-way select and the asynchronous transfer of control constructs. Technical Report LSN-039-UI, Ada 9X Language Study Note, March 1992.

[Ing91] Kim Ingram. Sparc for real-time applications. *The SPARC Technical Papers*, pages 313–321, 1991.

[ITM92] Y. Ishikawa, H. Tokuda, and C. Mercer. An object-oriented real-time programming language. *IEEE Computer*, 25(10):66–73, October 1992.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons Inc., 1991.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Jef89a] K. Jeffay. Analysis of a synchronization and scheduling discipline for real-time tasks with preemption constraints. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 295–305, December 1989.

[Jef89b] K. Jeffay. *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*. PhD thesis, University of Washington, Department of Computer Science, 1989.

[Jef92] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, December 1992.

[Jef93] K. Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing, Indianapolis, IN, ACM Press*, pages 796–804, February 1993.

[JLT85] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1985.

[JM86] Farnam Jahanian and Aloysius K. Mok. A graph-theoretic approach for timing analysis in real-time logic. In *Proceedings of the Real-Time Systems Symposium*, pages 98–108. IEEE Computer Society Press, December 1986.

[JN90a] E. Jensen and J. Northcutt. Alpha: A non-proprietary os for large, complex, distributed real-time systems. In *Second IEEE Workshop on Experimental Distributed Systems*, October 1990.

[JN90b] E. Jensen and J. Northcutt. Alpha: An open operating system for mission-critical real-time distributed systems - an overview. In *Proceedings of the 1989 Workshop on Operating Systems for Mission-Critical Computing*, 1990.

[Jon93a] Michael B. Jones. Modular real-time services: A key to flexible multimedia computing systems. In *Proceedings of the IEEE Workshop on the role of Real-Time Multimedia/Interactive Computing Systems*, Nov 1993.

[Jon93b] Michael B. Jones. *Personal communication*. 1993.

[JSP91] K. Jeffay, D. Stone, and D. Poirier. Yartos: Kernel support for efficient, predictable real-time systems. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 8–12, May 1991.

[JSS92] Kevin Jeffay, Donald Stone, and F. Donelson Smith. Kernel support for live digital audio and video. In *Computer Communications*, volume 15 (6), pages 388–395, July/August 1992.

[Kar90] Rabindra P. Kar. Implementing the rhealstone real-time benchmark. In *Dr. Dobb's Journal*, page 46, April 1990.

[KG94] Hermann Kopetz and Gunter Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. In *IEEE Computer*, pages 14–21. IEEE Computer Society Press, January 1994.

[Kir88] David B. Kirk. Process dependent static cache partitioning for real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 181–190. IEEE Computer Society Press, December 1988.

[Kir89] David B. Kirk. SMART (strategic memory allocation for real-time cache design. In *Proceedings of the Real-Time Systems Symposium*, pages 229–237. IEEE Computer Society Press, December 1989.

[KKG+90] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger. Tolerating transient faults in MARS. In *Proceedings of the Twentieth International Symposium on Fault-Tolerant Computing*, pages 466–473. IEEE Computer Society Press, 1990.

[KKS89] Dilip D. Kandlur, Daniel L. Kiskis, and Kang G. Shin. HARTOS: A distributed real-time operating system. In *Operating Systems Review of ACM SIGOPS*, pages 72–89. ACM Press, July 1989.

[KM85] Jeff Kramer and Jeff MaGee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[KM92] Tei-Wei Kuo and Aloysius K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of IEEE Real-Time Systems Symposium*, 1992.

[KMTD86] Korein, Maier, Taylor, and Durfee. A configurable system for automation programming and control. In *IEEE International Conference on Robotics and Automation, San Francisco, CA*, pages 1871–1877. IEEE, April 1986.

[KNT92] Takuro Kitayama, , Tatsuo Nakajima, and Hideyuki Tokuda. Rt-ipc: An ipc extension for real-time mach. *Proceedings of the USENIX Symposium on Micro-Kernel and Other Kernel Architectures*, pages 91–104, 1992.

[KO90] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. In *IEEE Transactions on Computers*, pages 933–940. IEEE Computer Society Press, August 1990.

[KOOH87] I. Kogiku, T. Ohrui, T. Ohkuba, and Y. Hamada. A real-time portable operating system common to switching and information processing applications. In *Proc. International Switching Symposium*, pages 308–312, March 1987.

[Kop93a] Hermann Kopetz. Scheduling. In Sape Mullender, editor, *Distributed Systems*, pages 491–509. ACM Press and Addison-Wesley, 1993.

[Kop93b] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered. In *IEICE Transactions on Electronics*, pages 196–207. Institute of Electronics, Information and Communication Engineers, Japan, November 1993. Also see Chapter 16: 'Real-Time and Dependability Concepts,' in 'Distributed Systems,' pages 425-433, ACM Press and Addison-Wesley, 1993.

[KP89] Rabindra P. Kar and Kent Porter. Rhealstone–a real-time benchmarking proposal. In *Dr. Dobb's Journal*, pages 14–24, February 1989.

[KS90] David B. Kirk and Jay K. Strosnider. SMART (strategic memory allocation for real-time cache design using the MIPS R3000. In *Proceedings of the Real-Time Systems Symposium*, pages 322–330. IEEE Computer Society Press, December 1990.

[KS91] Carol Kilpatrick and Karsten Schwan. Chaosmon – application-specific monitoring and display of performance information for parallel and distributed systems. In *ACM Workshop on Parallel and Distributed Debugging*, pages 57–67. ACM SIGPLAN Notices, Vol. 26, No. 12, May 1991.

[KS92] G. Koren and D. Shasha. D-over: an optimal on-line scheduling algorithm for overloaded real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 1992.

[KSC86] J. F. Kurose, S. Singh, and R. Chipalkatti. A study of quasi-dynamic load sharing in soft real-time distributed computer systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1986.

[KSH93] Gilad Koren, Dennis Shasha, and Shih-Chen Huang. MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–181. IEEE Computer Society Press, December 1993.

[KSO90] Carol Kilpatrick, Karsten Schwan, and David Ogle. Using languages for describing capture, analysis, and display of performance information for parallel and distributed applications. In *International Conference on Computer Languages '90, New Orleans*, pages 180–189. IEEE, March 1990.

[KTKS87] M. Kobayashi, S. Takenouchi, Y. Kushiki, and K. Sakamura. The software structure of extended nucleus based on btron specification. In *Proc. Fall Joint Computer Conference on Exploring Technology:Today and Tomorrow*, October 1987.

[KW91] Nick I. Kameoff and Nelson H. Weiderman. Hartstone distributed benchmarks: Requirements and definitions. In *Proceedings of the Real-Time Systems Symposium*, pages 199–208. IEEE Computer Society Press, December 1991.

[LA87] S. Levi and A. Agrawala. Temporal relations and structures in real-time operating systems. Technical Report CS-TR-1954, Department of Computer Science, University of Maryland, 1987.

[LA90] Shem-Tov Levi and Ashok K. Agrawala. *Real Time System Design*. McGraw Hill, 1990.

[Law81] E. Lawler. Scheduling periodically occuring tasks on multiprocessors. *Information Processing Letters*, 12(1):9–12, February 1981.

[Law83] E. L. Lawler. Recent results in the theory of machine scheduling. In A. Bachem et. al., editor, *Mathematical Programming: The State of the Art*, pages 202–233. Springer-Verlag, 1983.

[Lei80] D. W. Leinbaugh. Guaranteed response time in a hard real-time environment. *IEEE Transactions on Software Engineering*, January 1980.

[Lio91] M. Liou. Overview of the px64 kbit/s video coding standard. *Communications of the ACM*, 34(4), April 1991.

[LL73] C. W. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLN87] J. Liu, K. Lin, and S. Natarajan. Scheduling real-time, periodic jobs using imprecise results. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 252–260, December 1987.

[LM80] J. Y. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[LNL87] Kwei-Jay Lin, Swaminthan Natarajan, and Jane W.S. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *IEEE Real-time Systems Symposium*, pages 210–217, 1987.

[Loc86] C. Douglas Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie Mellon University, 1986.

[LPS85] L. Lamport and P.M.Melliar-Smith. Synchronizing clocks in the presence of faults. In *Journal of the ACM*, pages 52–78. ACM Press, January 1985.

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotone scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 10th Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[LY82] D. W. Leinbaugh and M. Yamini. Guaranteed response time in a distributed hard real-time environment. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1982.

[MA90] D. Mosse and A. Agrawala. On fault tolerance in real-time environments. Technical report, Department of Computer Science, University of Maryland, March 1990.

[Mar91] E. P. Markatos. Multiprocessor synchronization primitives with priorities. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 1–7, May 1991.

[MB93] Daniel Mosse and Rodrigo Botafogo. Some applications DON'T need real-time; better use it in applications that NEED guarantees. In *Proceedings of Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems*, November 1993.

[MCS91] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[MD78] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceeding of The Seventh Texas Conference on Computer Systems*, November 1978.

[MEG94] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. In *Operating Systems Review of the ACM Special Interest Group on Operating Systems*, pages 33 – 47, January 1994.

[MHM+90] John F. Muratore, Troy A. Heindel, Terri B. Murphy, Arthur Rasmussen, and Robert Z. McFarland. Real-time data acquidion at mission control. In *Communications of the ACM*, volume 33 (12), pages 18–31. ACM Press, December 1990.

[MI79] Robert B. McGhee and G. I. Iswandhi. Adaptive locomotion of a multilegged robot over rough terrain. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-9(4):176–182, April 1979.

[MIT90] Clifford W. Mercer, Yutaka Ishikawa, and Hideyuki Tokuda. Distributed hartstone: A distributed benchmark suite. In *Proceedings of the Real-Time Systems Symposium*, pages 70–77. IEEE Computer Society Press, December 1990.

[Mok83] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment.* PhD thesis, M.I.T., 1983.

[Mon87] H. Monden. Introduction to itron, the industry-oriented operating system. *IEEE Micro*, pages 53–65, April 1987.

[Moo68] J. M. Moore. Sequencing n jobs on one machine to minimize the number of late jobs. *Management Science*, 17(1), 1968.

[MS82] William C. McDonald and R. Wayne Smith. A flexible distributed testbed for real-time applications. *IEEE Computer Magazine*, 15(10):25–39, Oct. 1982.

[MS93a] Jim McDonald and Karsten Schwan. Ada dynamic load control mechanisms for distributed embedded battle management systems. In *First Workshop on Real-time Applications, New York*, pages 156–160. IEEE, May 1993.

[MS93b] Bodhisattwa Mukherjee and Karsten Schwan. Application dependent heterogeneous thread schedulers. Technical Report GIT-CC-93/43, College of Computing, Georgia Tech, August 1993. To be completed.

[MS93c] Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a reconfigurable micro-kernel. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 45–60, September 1993.

[MST89] T. Marchok, J. Strosnider, and H. Tokuda. Token-ring adapter-chipset architectural considerations for real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1989.

[MSZ90] L. D. Molesky, C. Shen, and G. Zlokapa. Predictable synchronization mechanisms for multiprocessor real-time systems. *Journal of Real-Time Systems*, 3(2), 1990.

[MT90] C. Mercer and H. Tokuda. The arts real-time object model. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 2–10, 1990.

[MT92] Clifford W. Mercer and Hideyuki Tokuda. Preemptability in real-time operating systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 1992.

[Muk91] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of the Sun User Group Technical Conference*, pages 101–112, June 1991.

[MZ93] Nicholas Malcolm and Wei Zhao. Hard real-time communication in multiple-access networks. Technical Report 93-039, Department of Computer Science, Texas A&M University, 1993. To Appear in Journal of Real-Time Systems.

[MZ94] Nicholas Malcolm and Wei Zhao. The timed-token protocol for real-time communication. In *IEEE Computer*, pages 35–41. IEEE Computer Society Press, January 1994.

[NCS+90] J. Northcutt, R. Clark, S. Shipman, D. Maynard, E. Jensen, F. Reynolds, and B. Dasarathy. Threads: A programming construct for reliable real-time distributed programming. In *Proc. International Conf. on Parallel and Distributed Computing and Systems*, October 1990.

[NEN+92] Henry Neugass, Geoffrey Espin, Hidefume Nunoe, Ralph Thomas, and David Wilner. VxWorks: An interactive development environment and real-time kernel for GMicro. In *Proceedings of The Eighth TRON Project Symposium (International), 1992*, pages 196–207. IEEE Computer Society Press, December 1992.

[Nie89] Douglas Niehaus. Program representation and translation for predictable real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 53–63. IEEE Computer Society Press, December 1989.

[NNS91] Douglas Niehaus, Erich Nahum, and John A. Stankovic. Predictable real-time caching in the spring kernel. In *Proceedings of the Joint IEEE Workshop on Real-Time Operating Systems and Software and IFAC Workshop on Real-Time Programming, Atlanta, GA*. IEEE, May 1991.

[NP91] V. Nirkhe and W. Pugh. A partial evaluator for the maruti hard real-time system. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

[NRS+93] Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles Weems. The spring scheduling co-processor: Design, use and performance. In *Proceedings of the Real-Time Systems Symposium*, pages 106–111. IEEE Computer Society Press, December 1993.

[NYM92] Jun Nakajima, Masatomo Yazaki, and Hitoshi Matsumoto. Multimedia/realtime extemsions for mach 3.0. *Proceedings of the USENIX Workshop on Micro-Kernel and Other Kernel Architectures*, pages 161–175, April 1992.

[OSS90] David M. Ogle, Karsten Schwan, and Richard Snodgrass. The dynamic monitoring of real-time distributed and parallel systems. Technical report, College of Computing, Georgia Institute of Technology, ICS-GIT-90/23, Atlanta, GA 30332, May 1990. Also in IEEE TPDS July 1993.

[OSS93] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[OWK87] T. Ohkubo, T. Wasano, and I. Kogiku. Configuration of the ctron kernel. *IEEE MICRO*, april 1987.

[PHOA89] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abbot. Rpc in the x-kernel. In *Twelfth ACM Symposium on Operating Systems*, pages 91–101. ACM, Dec. 1989.

[PM93] Michael L. Powell and James J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, volume 27 (5), pages 69–79. ACM Press, December 1993.

[QD92] T. Quiggle and G. Dismukes. An analysis of the implementation and execution-time impact of ada 9x real-time features. Technical Report LSN-040-UI, Ada 9X Language Study Note, March 1992.

[Rea90] James F. Ready. Os kernel design considerations for real-time systems. *COMPCON Spring 1990: Thirty-Fifth IEEE Computer Society International Conference*, pages 278–281, February 1990.

[RKS90] P. Ramanathan, D.D. Kandlur, and K.G. Shin. Hardware-assisted software clock synchronization for homogeneous distributed systems. In *IEEE Transactions on Computers*, pages 514–524. IEEE Computer Society Press, April 1990.

[RS84] K. Ramamritham and J.A. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Software*, 1(3):65–75, July 1984.

[RSL87] R. Rajkumar, L. Sha, and J. P. Lehoczky. On countering the effects of cycle-stealing in a hard real-time environment. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 2–11, December 1987.

[RSL88] R. Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of Real-Time Systems Symposium, Huntsville, AL*, pages 259–269. IEEE, December 1988.

[RSL89] R. Rajkumar, L. Sha, and J. Lehocsky. An experimental investigation of synchronization protocols. In *Proceedings of 6th IEEE Workshop on Real-time Operating Systems and Software*, pages 11–17, May 1989.

[RSS90] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. In *IEEE Transactions on Parallel and Distributed Systems*, volume 1(2), pages 184–194, April 1990.

[RSZ89] Krithi Ramamritham, John A. Stankovic, and Wei Zhao. Distributed scheduling of tasks with deadline and resource requirements. In *IEEE Transactions on Computers*, August 1989.

[Sai92] Kazunori Saitoh. ITRON standards. In *Proceedings of The Eighth TRON Project Symposium (International), 1992*, pages 16–24. IEEE Computer Society Press, December 1992.

[Sak87a] K. Sakamura. Architecture of the tron vlsi cpu. *IEEE Micro*, pages 17–31, April 1987.

[Sak87b] K. Sakamura. Btron: The business-oriented operating system. *IEEE Micro*, pages 53–65, April 1987.

[Sak87c] K. Sakamura. The tron project. *IEEE MIcro*, pages 8–14, April 1987.

[Sak89a] TRON ASSOCIATION (Supervisor: Dr. K. Sakamura), editor. *Original CTRON Specification Series, Vols 1-8 and Separate Vol 1*. Ohmsha, 1989.

[Sak89b] TRON ASSOCIATION (Supervisor: Dr. K. Sakamura), editor. *Outline of CTRON*. Original Ctron Specification Series. Ohmsha, 1989.

[Sal93] Lou Salkind. Unix for real-time control: Problems and solutions. In John A. Stankovic and Krithi Ramamritham, editors, *Advances in Real-Time Systems*, pages 223–236. IEEE Computer Society Press, 1993. Also available as New York University Techical Report No. 400, Robotics Report no. 171.

[SB87] Karsten Schwan and Ben Blake. Experimental evaluation of a real-time scheduler for a multiprocessor system. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-5/87-TR16, Sept. 1987. Also in IEEE TSE, Jan. 1991.

[SB89] M. Schroeder and M. Burrows. Performance or firefly rpc. In *Twelfth ACM Symposium on Operating Systems, SIGOPS, 23, 5*, pages 83–90. ACM, SIGOPS, Dec. 1989.

[SB90] Karsten Schwan and Win Bo. Topologies – distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.

[SBB87] Karsten Schwan, Thomas E. Bihari, and Ben Blake. Adaptive, reliable software for distributed and parallel, real-time systems. In *Sixth Symposium on Reliability in Distributed Software, Williamsburg, Virginia*, pages 32–44. IEEE, March 1987.

[SBWT87] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[Sch88] Karsten Schwan. Developing high-performance, parallel software for real-time applications. *Information and Software Technology, Butterworths Scientific Limited*, pages 218–227, May 1988.

[SFG+91] Karsten Schwan, Harold Forbes, Ahmed Gheith, Bodhisattwa Mukherjee, and Yiannis Samiotakis. A cthread library for multiprocessors. Technical Report GIT-ICS-91/02, College of Computing, Georgia Institute of Technology, 1991.

[SGB87] Karsten Schwan, Prabha Gopinath, and Win Bo. Chaos – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[SGZ90] Karsten Schwan, Ahmed Gheith, and Hongyi Zhou. From chaos-min to chaos-arc: A family of real-time multiprocessor kernels. In *Proceedings of the Real-Time Systems Symposium, Orlando, Florida*, pages 82–92. IEEE, Dec. 1990.

[Sha86] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 198–204. IEEE, May 1986.

[Shi91] Kang G. Shin. HARTS: A distributed real-time architecture. In *IEEE Computer*, pages 25–35. IEEE Computer Society Press, May 1991. Also in ‘Readings in Real-Time Systems’, Eds. Y.H.Lee and C.M.Krishna, pp. 30-49.

[SK91] D. B. Stewart and P. K. Khosla. Real-time scheduling of sensor-based control systems. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[SL92] Wei-Kuan Shih and Jane W. S. Liu. On-line scheduling of imprecise computations to minimize error. In *Proceedings of IEEE Real-Time Systems Symposium*, 1992.

[SLJ88] L. Sha, J. P. Lehoczky, and E. D. Jensen. Modular concurrency control and failure recovery. *IEEE Transactions on Computers*, 37(2), 1988.

[SLR86] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of IEEE Real-Time Systems Symposium*, 1986.

[SLS88] B. Sprunt, John P. Lehoczky, and Lui Sha. Expoiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of Real-Time Systems Symposium, Huntsville, AL*, pages 251–258. IEEE, 1988.

[Sof92] RR Software. Runtime implementation strategies for protected records, multiway entry cal, and asynchronous transfer of control. Technical Report LSN-041-UI, Ada 9X Language Study Note, March 1992.

[SR84] Karsten Schwan and Rajiv Ramnath. Adaptable operating software for manufacturing systems and robots: A computer science research agenda. In *Proceedings of the 5th Real-Time Systems Symposium, Austin, Texas*, pages 255–262. IEEE, Dec. 1984.

[SR85] Zary Segall and Larry Rudolph. Pie: A programming and instrumentation environment for parallel processing. *IEEE Software*, 2(6):22–37, Nov. 1985.

[SR87] J. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 146–157, December 1987.

[SR89] John A. Stankovic and Krithi Ramamritham. The spring kernel: A new paradigm for real-time operating systems. *A Quarterly Publication of the Special Interest Group on Operating Systems*, 23(3):54–71, July 1989.

[SR90] John A. Stankovic and Krithi Ramamritham. Editorial: What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, November 1990.

[SR91] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. In *IEEE Software*, pages 62–72, May 1991.

[SR93a] John. A. Stankovic and Krithi Ramamritham. Introduction. In *Advances In Real-Time Systems*, pages 1–16. IEEE Computer Society Press, 1993.

[SR93b] John. A. Stankovic and Krithi Ramamritham. Scheduling. In *Advances In Real-Time Systems*, pages 47–52. IEEE Computer Society Press, 1993.

[SR94] J. A. Staankovic and K. Ramamritham. A reflective architecture for real-time operating systems. In *Principles of Real-Time Systems, Sang Son, Ed., PrenticeHall*, 1994.

[SRC85] J. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a bidding algorithm for hard real-time distributed systems. *IEEE Transactions on Computers*, C-34(12), December 1985.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[SRS93] Chia Shen, Krithi Ramamritham, and John A. Stankovic. Resource reclaiming in multiprocessor real-time systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 382–397, April 1993.

[SS87] J. Stankovic and L. Sha. The principle of segmentation. Technical report, Department of Computer Science, University of Massachusetts, 1987.

[SSL89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[Sta92] Professor John A. Stankovic. Real-time operating systems: What's wrong with today's systems and research issues. In *Real-Time Systems Newsletter of the IEEE Technical Committee on Real-Time Systems*, volume 8 (1/2), pages 1–10, Spring/Summer 1992.

[Str] J. Strosnider. *Real-Time Communication*. PhD thesis, Department of ECE, Carnegie Mellon University.

[SZ92] Karsten Schwan and Hongyi Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, Aug. 1992.

[SZG91] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Multiprocessor real-time threads. *Operating Systems Review*, 25(4):35–46, Oct. 1991. Also appears in the Jan. 1992 issue of Operating Systems Review.

[Ten90] D. L. Tennenhouse. The viewstation research program on distribured video systems. In *Proceedings of the First International Workshop on Network and Operating System Support for Audio and Video*, November 1990.

[Tho90] L. M. Thompson. Using psos+ for embedded real-time computing. *COMPCON Spring 1990: Thirty-Fifth IEEE Computer Society International Conference*, pages 282–288, February 1990.

[TK88] H. Tokuda and M. Kotera. A real-time tool set for the arts kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1988.

[TM89] H. Tokuda and C. Mercer. Arts: A distributed real-time kernel. *Operating Systems Review*, 23(3):29–53, July 1989.

[TMIM89] H. Tokuda, C. Mercer, Y. Ishikawa, and T. Marchok. Priority inversions in real-time communication. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1989.

[TML90] H. Tokuda, C. Mercer, and J. Lehoczky. Scheduling theory and practice in arts. Technical report, School of Computer Science, Carnegie Mellon University, August 1990.

[TN91] Hideyuki Tokuda and Tatsuo Nakajima. Evaluation of real-time synchronization in real-time mach. *Proceedings of the USENIX 1991 Mach Workshop*, October 1991.

[TNR90a] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, October 1990.

[TNR90b] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards predictable real-time systems. *Proceedings of the USENIX 1990 Mach Workshop*, October 1990.

[To73] K.H. Timmesfeld and others. Pearl: A proposal for a process and experiment automation real-time language. Technical report, Gesellschaft fuer Kernforschung mbH, Karlsruhe, PDV-Bericht KFK-PDV1, April 1973.

[Tok88] Hideyuki Tokuda. A real-time tool set for the arts kernel. In *Real-Time Systems Symposium, Huntsville, Alabama*, pages 289–299. IEEE, Dec. 1988.

[TS92] Hiroaki Takada and Ken Sakamura. Advances in the ITRON specifications – supporting multiprocessor and distributed systems. In *Proceedings of The Ninth TRON Project Symposium (International), 1992*, pages 89–95. IEEE Computer Society Press, December 1992.

[TTCM92] H. Tokuda, Y. Tobe, S. Chou, and J. Moura. Continuous media communication with dynamic qos control using arts with an fddi network. Personal Communication, October 1992.

[Ver93] Paulo Verissimo. Real-time communication. In Sape Mullender, editor, *Distributed Systems*, pages 447–490. ACM Press and Addison-Wesley, 1993.

[VK93] Paulo Verissimo and Hermann Kopetz. Design of distributed real-time systems. In Sape Mullender, editor, *Distributed Systems*, pages 511–530. ACM Press and Addison-Wesley, 1993.

[Wal91] Gregory K. Wallace. The JPEG still picture compression standard. In *Communications of the ACM*, volume 34 (4), pages 30–44. ACM Press, April 1991.

[WC87] C. M. Woodside and D. W. Craig. Local non-preemptive scheduling policies for hard real-time distributed systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 12–16, December 1987.

[Wei84] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. In *Communications of the ACM*, volume 27 (10), pages 1013–1030. ACM Press, October 1984.

[Wel92] A. Wellings. Ada run-time environment working group meeting summary. *Ada Letters*, 12(5):30–35, September 1992.

[Wir77] N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, August 1977.

[WK92] Nelson H. Weiderman and Nick I. Kameoff. Hartstone uniprocessor benchmark: Definitions and experiments for real-time systems. In *Real-Time Systems Journal*, volume 4 (4), pages 353–383. Kluwer Academic Publishers, December 1992.

[WOK+87] T. Wasano, M. Ohminami, Y. Kobayashi, T. Ohkubo, and K. Sakamura. Design principles and configurations of ctron. In *Proceedings of Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow*, pages 159–166, October 1987.

[Woo86] Michael H. Woodbury. Analysis of the execution time of real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 89–96. IEEE Computer Society Press, December 1986.

[YA89] X. Yuan and A. Agrawala. A decomposition approach to nonpreemptive real-time scheduling. Technical Report CS-TR-2345, Department of Computer Science, University of Maryland, November 1989.

[Zha91] L. Zhang. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, March 1991.

[Zho92] Hongyi Zhou. *Task Scheduling and Synchronization for Multiprocessor Real-Time Systems.* PhD thesis, College of Computing, Georgia Institute of Technology, Atlanta, GA, April 1992.

[ZRS87a] W. Zhao, K. Ramamritham, and J. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, May 1987.

[ZRS87b] Wei Zhao, Krithi Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, August 1987.

[ZS91] Hongyi Zhou and Karsten Schwan. Dynamic scheduling for hard real-time systems: Toward real-time threads. In *Proceedings of the Joint IEEE Workshop on Real-Time Operating Systems and Software and IFAC Workshop on Real-Time Programming, Atlanta, GA*. IEEE, May 1991.

[ZSA91] Hongyi Zhou, Karsten Schwan, and Ian Akyildiz. Performance effects of information sharing in a distributed multiprocessor real-time scheduler. Technical report, College of Computing, Georgia Tech, GIT-CC-91/40, Sept. 1991. Abbreviated version in 1992 IEEE Real-Time Systems Symposium, Phoenix.

[ZSG92] Hongyi Zhou, Karsten Schwan, and Ahmed Gheith. The dynamic synchronization of real-time threads for multiprocessor systems. In *Symposium on Experiences with Distributed and Multiprocessor Systems, Newport Beach*, pages 93–107. Usenix, ACM, March. 1992.