A Time-Sharing Architecture for Complex Real-Time Systems

Jair Jehuda
Department of Electrical Engineering
Technion - Israel Institute of Technology
Technion City, Haifa 32000, ISRAEL

Gilad Koren
Computer Science Institute
Bar-Ilan University
Ramat-Gan, 52900, ISRAEL

Daniel M. Berry
Department of Computer Science
Technion - Israel Institute of Technology
Technion City, Haifa 32000, ISRAEL

Abstract

In this paper we show how a real-time time-sharing *RtTS architecture can be very useful in resolving many of the formidable problems generally posed by complex real-time systems. In particular, we address dynamic multiple job systems, running on shared-memory multi-processor platforms. Each job is multi-tasked, with task characteristics assumed to be complex, e.g. some critical, some dependent, some aperiodic. Each job may also have multiple states, and may support several alternate modes of operation. Concurrent job sets, modes, states, and even available processor capacities, are all assumed dynamic. To accommodate such complexities, the RtTS architecture adopts a practical* divide-and-conquer *approach, which is shown to be very effective. The architecture incorporates three distinct and independent software control layers, which together facilitate automatic near-optimal mode selection, dynamic load-balancing, and reliable real-time time-sharing, in a fully integrated manner. The* job-oriented *strategy allows each job to be developed independently, as a* black box *with uniform control requirements, herein described. The unique capabilities of this RtTS architecture are illustrated in a dynamic multimedia context, where it is shown to have several advantages over conventional dynamic load-balancing techniques, in supporting complex task characteristics, best-effort system values, dynamic* critical *task sets, scalability, and portability.*

1 Introduction

In this paper, we show how a practical, job-oriented, *real-time time-sharing* RtTS architecture, can dynamically facilitate reliable real-time scheduling, job load-balancing, and a maximized system value, in complex, multiple job, shared-memory *multi-processor* systems. Each *job*, here, is an independently developed real-time application, with several alternate *modes* of operation. Each job *mode* dictates a different work load, and a distinct job *reward* contribution to an overall *system value*. Modes of operation are selected automatically to provide the highest attainable system value without overloading system resources. Jobs are internally *complex* in the sense that they must simultaneously cater to several dimensions of real-time systems [1], e.g. state-dependent work loads and job rewards, independent and non-independent task sets, tight and loose time constraints, hard and soft deadlines, periodic and aperiodic tasks. Concurrent jobs execute independently, sharing no resources other than the CPU.

A typical example of such a system is an integrated multimedia system designed to support multiple audio, video, MIDI, and fax/modem I/O channels[1]. Each job in this example will facilitate any of several forms of interactive entertainment, or provide any of a variety of programmable banking, information gathering, electronic shopping, and fax/answering machine services. Such jobs are usually developed independently and provided as *black box*, plug-in modules. Jobs are usually multi-tasked (or multi-threaded), and they may have complex real-time characteristics and requirements, some of them *critical*. The *active* set of concurrent jobs is determined online by user preferences and requirements, and is therefore time-variant. We also assume that available shared-memory processing resources may vary due to possible faults and to accommodate portability. So the active set of jobs, must be dynamically *load balanced*, i.e. partitioned over the available set of processors, in a manner which will guarantee the *real-time schedulability* of all task sets assigned to each processor. We assume that task sets for each job are already known to be schedulable when running on a dedicated processor with an adequate given speed[2]. But the real-time scheduling requirements for any job might be too complex to be reconciled online with those of other jobs sharing the same processor.

Furthermore, each job in this example may support several operational *mode* alternatives to choose from, e.g. various modem baud rates, audio sampling rates, and video frame rates and resolutions. Each such mode clearly has different processing requirements and provides a different quality of service, referred to as the *job reward*. The overall *system value* is a function of the current job rewards. So job modes must therefore be automatically selected in a manner which maximizes the system value without overloading available resources. Mode selection and load balancing decisions are collectively referred to as *meta*-control, because real-time scheduling decisions are dictated by them.

Also to be considered are internal job *states* which may affect current job rewards and work loads, e.g. when time-driven activities are overdue, during time-triggered fax transmission, or when an information gathering job awaits results being provided by remote internet activities. Meta-control decisions must therefore be reconsidered with each significant job state transition and with each attempt to alter the active set of jobs.

Another useful example of such a system is an *upgraded* real-time system in which several applications used to run on a set of dedicated processors and we now wish to integrate them on a smaller set of faster processors to reduce production and maintenance costs. Finding a solution for such systems also provides us with a low-cost method for enhancing already developed applications, if extensions can be developed as independent jobs.

To satisfy these needs we have devised a comprehensive software control architecture capable of supporting dynamic load balancing, near-optimal mode selection, and the real-time scheduling of arbitrary sets of complex jobs, in an *integrated* manner. An integrated approach is essential to critical real-

[1] The RtTS architecture was designed with this application in mind. The musical ATLAS testbed [2] is currently being implemented to demonstrate the effectiveness of the RtTS architecture.

[2] Throughout this paper we use the processor clock frequency as a measure of processor speed or capacity.

0-8186-7123-8/95 $4.00 © 1995 IEEE

time systems because decisions in each of realms are influenced by each other, e.g. independent mode selection may determine a set of critical tasks which cannot be load-balanced, and a load-balancing decision may generate job subsets which cannot be reliably scheduled. As we show in this paper, the RtTS architecture provides practical and effective solutions to all of these problems.

1.1 Previous Work

Current solutions to dynamic load balancing [3, 4, 5, 6] are inadequate because they are *task-oriented*, i.e. individual incoming tasks are accepted or rejected by online schedulers on an FCFS (first-come-first-serve) basis. Such an approach could probably be extended to support jobs as atomic groups of tasks, but it would be practical only if all task characteristics internal to each job can be made known to the load balancing system, and if scheduling complexities are sufficiently low. Unfortunately, none of these conditions appear realistic when dealing with complex black box jobs with multiple states and modes. Another problem is that a FCFS policy cannot maximize system value, since this sometimes requires the suspension of currently running jobs to accommodate incoming jobs with higher rewards. We also know of no existing dynamic load-balancing solution which can efficiently accommodate dynamic sets of *critical* tasks requiring 100% scheduling guarantees.

Software meta-controllers have already been developed for several experimental real-time systems, but the scope of considered adaptations have always been severely limited. In the Spring kernel [3], for example, Meta Level Controller adaptations are confined to scheduling algorithms and parameters. RESAS adaptations [7] include processor assignment (load-balancing) and the management of various reallocating time/space redundancies, but only one adaptation form (one degree of freedom) is actually treated at a time. The GEM Adaptation Controller [8] supports both mode selection and load balancing, but decisions are only *partially* automated. None of the cited examples provide measures of performance which enable us to evaluate their performance in absolute terms. The RtTS meta-control model, on the other hand, can *simultaneously* accommodate a very large class of adaptations, with well-defined performance measures which can be shown to be near-optimal.

1.2 Summary of Results

Control Layer	Level	Function	Typ. Rate
Automated Meta-Control	3	mode selection & load balancing	secs
Real-Time Time-Sharing	2	job time-sharing	msecs
	1	job scheduling	

Figure 1. Software Control Architecture

To greatly reduce scheduling and load-balancing complexities, the RtTS architecture uses a *job-oriented*, divide-and-conquer strategy, which requires that each job be internally responsible for the scheduling of its own tasks, using arbitrary real-time scheduling policies. As we will show, arbitrary sets of such jobs can then be reliably *time-shared* on any given processor, using a very simple joint schedulability test (see Equation (1)), without being exposed to their inner complexities. Conventional time-sliced time-sharing is shown to be inadequate, so we hereby introduce an earliest-deadline-first (EDF) time-sharing scheme capable of reliably interleaving the internal job schedules with minimal overhead. All time-sharing overheads are fully accounted for in a manner which enables the independent development of each job and its internal scheduling solution, regardless of which jobs will be time-shared with it.

As depicted in Figure 1, the RtTS architecture comprises of three fully independent software control layers. The bottom two control layers facilitate EDF job time-sharing and internal job scheduling on any given processor, enabling each job to use an arbitrary and independent real-time scheduling policy to best meet its complex requirements. The top *automated meta-control* layer is responsible for job mode selection and for the load balancing of the entire set of active jobs over all available processors. Probably the most important feature of the real-time time-sharing scheme is that it enables *simultaneous* resolution of near-optimal mode selection and job load-balancing. Two practical and very effective approximation algorithms, QDP and G^2, have been developed for this purpose, each of them requiring only fractions of a second to produce system values which are rarely suboptimal by more than a few percent. This enables us to reevaluate, with reasonable overhead, our decisions every few seconds to accommodate varying job sets and job states. The RtTS architecture is also shown to have considerable advantages in its load-balancing support for dynamic *critical* task sets, and in its inherent scalability and portability.

The rest of this paper is organized as follows. Section 2 introduces the *black box* job paradigm, describes EDF job time-sharing, and illustrates by example how it works. Section 3 briefly introduces automated software meta-control, with more details being provided by [9]. Section 4 shows how job scheduling, time-sharing, and meta-control layers are incorporated into a comprehensive software architecture, most suitable for dynamic, multiple job, shared memory systems. We conclude in Section 5.

2 Real-Time Time-Sharing

Real-time time-sharing is a method by which a set of real-time jobs, J_c, can reliably share a given processor, c, even though each job, $j \in J_c$, has been developed independently as a *black box*. Each job, j, consists of an arbitrary task set, τ_j, with real-time requirements and characteristics which might be complex. In the *black box* paradigm, we assume that it would be unreasonable to require that each job reveal all of its complex task requirements and characteristics to external software controllers. Even if we could require this revelation, we assume that complexities might be too significant for these details to be of any use. So we require that each job be already equipped with a scheduling policy, π_j, which has already been found capable of satisfying all τ_j requirements when provided a processor with a given adequate processing speed. The *minimal* such speed is called the job *bandwidth*, b_j, and an appropriate bandwidth must be determined for every possible job mode and state. As we will show, this job bandwidth is computed in a manner which also accommodates worst-case time-sharing overheads.

As depicted in Figure 1, the middle *job time-sharing* control layer determines when a specific job in J_c will be dispatched to processor c. Whenever Job $j \in J_c$ is dispatched, the internal *job scheduling* of τ_j tasks is carried out by the π_j scheduler within the bottom control layer. The outcome is that if the shared processor has a processing speed equal to the sum of all bandwidths, then all π_j schedules are essentially *interleaved* without being altered or violated.

In this section, we begin by describing the *black box* paradigm and listing the responsibilities that each job has to the upper control layers in the RtTS architecture. We then provide a simple time-sharing schedulability test, show how job bandwidths are calculated, and illustrate by example how it works.

2.1 The Black Box Paradigm

The black box paradigm clearly defines how each job must appear to the upper two RtTS control layers. The following are required of each individual job:

1. Most important, the black box paradigm requires that each Job j can internally schedule its τ_j task set using an arbitrary predetermined scheduling policy, π_j.
2. Secondly, the black box must export a *bandwidth function*, $b_j(m_j)$, which reveals the minimal processing speed required by π_j to schedule τ_j when in job mode m_j. The value returned may vary according to the current internal job state, s_j. Referring back to our multimedia example, typical fax job states would include being idle, sending, and receiving, while typical video job states would include play, scan, fast-forward, pause, and rewind. Ideally, bandwidth values for all modes and states should be computed offline for efficient use. When calculating τ_j computation times, and producing appropriate schedules, the π_j scheduler will always assume that it is running on a processor with this minimal processing speed.
3. Each job must also inform the upper control layers of any change in the internal job state, so that the new bandwidth requirments can be reconsidered.
4. To support EDF time-sharing, each job, j, must also keep the time-sharing control layer informed of its earliest pending deadline, d_j.
5. To facilitate automated meta-control, each job must also export a *job reward function*, $r_j(m_j)$, which reflects the contributed job reward if mode m_j is selected in the current internal job state, s_j.

Thus all the upper control layers need to know are bandwidth and reward functions, when a state-transition has taken place, and what the earliest pending deadline is. The internal scheduling policy and all other real-time characteristics and requirements are hidden, so that each job can indeed be treated as a black box. Most important, it enables us to develop each job independently, without having to consider the complex requirements and characteristics of other jobs.

2.2 Time-Sharing Schedulability

As previously noted, the current job bandwidth, b_j, is determined such that a feasible schedule for τ_j is guaranteed if provided a processor with a processing speed of at least b_j. Let β_c denote the processing speed of processor c, henceforth referred to as the processor *capacity*. Real-time time-sharing ensures that any job set, J_c, can reliably time-share a processor, c, if the sum of their bandwidths does not exceed the capacity of processor c, i.e.

$$\sum_{j \in J_c} b_j \leq \beta_c. \qquad (1)$$

Thus two real-time jobs can reliably time-share a 20 MHz processor if each of them requires a 10 MHz bandwidth.

Let's assume for a moment that job time-sharing overheads do not exist. Then Equation (1) essentially guarantees that the number of processor execution cycles on the shared processor, for any given time period, is sufficient for all jobs sharing it. Thus the total processor utilization will never exceed one during any given period, so EDF schedule interleaving must succeed [10].

There are, of course, various overheads which must be treated such as critical sections where jobs must not be preempted[3], job preemption costs, and the costs of updating and maintaining a list of all d_j deadlines so that the job with the earliest d_j is always dispatched. As we soon show, all of these time-sharing overheads can be accommodated when computing b_j.

```
procedure UpdateEarliestJobDeadline( j, d_j );
/* Job j notifies that closest deadline is d_j */
begin
/* Find executing Job j' and its deadline d': */
      Let (j', d') = first element in JobDeadlines ;
      Update j's entry in JobDeadlines ;
      if d_j < d' and j ≠ j'
      /* then the scheduler of job j will now take over: */
        then preempt j' and dispatch j;
end;
procedure EventsHandler( j, event_type );
begin
/* This might call upon UpdateEarliestJobDeadline: */
      Execute the relevant event handler of Job j;
end;
```

Figure 2. Updating Earliest Job Deadlines

2.3 EDF Time-Sharing

Before we deal with time-sharing overheads, lets see by example how EDF time-sharing works. In the following code, JobDeadlines is a deadline-ordered list of $\{j, d_j\}$ pairs where d_j is the earliest active deadline in Job j. Each processor, $c \in C$, has a list of such pairs belonging to the set of jobs, J_c, which share it. The job that has the closest (smallest) deadline in each processor is dispatched to execute whatever task is appropriate to its scheduling method, i.e. not necessarily the one with the earliest deadline among ready tasks of that job. *Dispatched*, here, means that the scheduling method of the dispatched tasks takes over. As required by item 4 of Section 2.1, the *UpdateEarliestJobDeadline* procedure in Figure 2 is used by all jobs to keep JobDeadlines up-to-date with the current d_j for Job j. The d_j parameter must be zero when Job j has no pending deadlines, whereupon the Job j element in JobDeadlines must be deleted. The procedure must first find the earliest element in JobDeadlines to determine the currently dispatched job, j', and its earliest deadline, d'. After the d_j update, the procedure must then determine if d_j is earlier than d', whereupon j must be dispatched if it isn't already.

The earliest job deadline, d_j, can only change when a job task *completes* or when a new task is *released*, i.e. becomes

[3] For the sake of simplicity, we will overlook overheads incurred by critical sections and the atomic nature of processor execution cycles. Interested readers should refer to [11].

ready. Periodic tasks are usually released when a timer event indicates that its time has come. Other tasks are usually released by other tasks or events. As illustrated by the *EventsHandler* procedure in Figure 2, each event must be promptly handled by the event handlers of each job so that they can call upon *UpdateEarliestJobDeadline* if necessary.

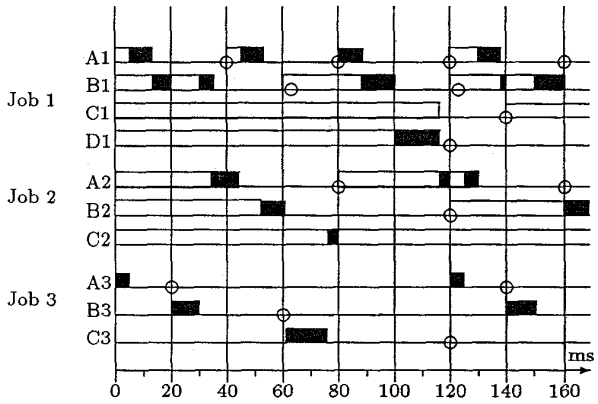

Figure 3. EDF Job Time-Sharing

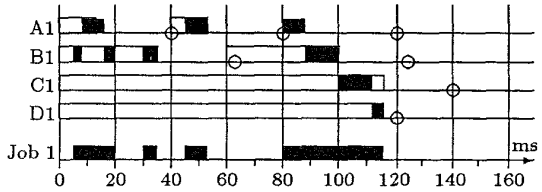

Figure 4. Actual RM+PCP Scheduling in Job 1

Figure 3 illustrates how three sample jobs are EDF dispatched by the time-sharing mechanism when sharing a 4 MHz processor. Each of these jobs have several tasks, e.g. $\tau_1 = \{A_1, B_1, C_1, D_1\}$. Each job is also using a different internal scheduling policy, π_1 for job 1 is dynamic Rate Monotonic policy (RM) [12] using a priority ceiling protocol (PCP) [13, 14] to prevent priority inversion, π_2 for job 2 is EDF [12, 10], and π_3 for job 3 is a static cyclic executive (CE) [15, 16]. To the right of each task we have frames which indicate when each task was released and when it completed. Following each frame we find a circle which marks the actual d_j deadline associated with each frame. The shaded areas within these frames indicate when the d_j deadline associated with that frame was the earliest of all d_j deadlines, causing the corresponding job to be dispatched. Jobs are thus EDF dispatched, but the internal tasks are actually scheduled in accordance with the π_j scheduling policies employed by each job. To illustrate this point, we provide Figure 4 which shows how RM+PDP scheduling is carried out for the tasks of job 1 whenever job 1 is dispatched. Unlike in Figure 3, the shaded areas in this case indicate when each *task* was actually scheduled in accordance with the internal RM+PDP scheduling policy being used. Letting $P(i)$ denote the period for task i, then $P(A_1) < P(B_1) < P(C_1) < P(D_1)$, so C_1 has a higher priority than D_1 in an RM policy. Thus, for example, task C_1 is scheduled by the internal Job 1 policy at time 100 ms, even though it was the d_j associated with D_1 which prompted job 1 to be dispatched at that time, because the D_1 deadline is earlier than that of C_1. Job schedules are thus *interleaved* without changing or violating the original π_j task schedules within each job. For a more formal treatment, the interested reader is referred to [11]

This EDF real-time time-sharing scheme might seem trivial at first glance. A deeper study, however, reveals that this is not so. It is essential to the RtTS architecture that time-sharing overheads be fully accommodated for arbitrary J_c job sets without having to recompute bandwidths, online, as a function of J_c. It is also essential that time-sharing overheads be minimized. Conventional time-slicing, for example, is found to be completely inadequate, since we can always devise a J_c task set which would require very minute time-slices, thereby incurring unacceptable time-sharing overheads. As we now show, EDF time-scheduling, on the other hand, achieves both objectives.

2.4 Time-Sharing Overheads

As previously described, to facilitate EDF time-sharing, the time-sharing control layer must maintain a list containing the earliest deadline, d_j, for each job, $j \in J_c$, and each black box job scheduler must ensure that its d_j deadline is up-to-date. As illustrated by Figure 2, all time-sharing activities are associated with these deadline updates, which must be carried out by the job tasks at well-defined points of execution, e.g. at task completion, or when an event-driven or timer-driven interrupt handler determines that a task should be released. Worst-case time-sharing overheads are therefore readily accommodated by appropriately increasing the number of processing cycles required by each task.

For efficient maintenance, all of these deadline values are stored in an indexed heap. Letting N denote the maximum number of jobs in J_c, then the earliest deadline can always be found at the top with $O(1)$ complexity, while a deadline can be inserted or deleted with $O(\log N)$ complexity. Thus worst-case costs for deadline updates can be *a priori* determined as a function of N.

As already noted, a task can cause a job preemption only when it completes or is released. This implies that the number of job preemptions in EDF time-sharing cannot exceed double the total number of tasks released and completed in J_c. We can therefore account for worst-case job preemptions by augmenting the computational requirements of each J_c task by the number of execution cycles required for two job preemptions.

2.5 Computing Job Bandwidths

We now refer again to the three sample jobs of Figure 3 to demonstrate how job bandwidths are computed and time-sharing overheads are accommodated. Table 1 lists essential task characteristics and requirements for these three sample jobs. Each job has three *critical* tasks, with the fourth D_1 task in job 1 being non-critical. This paves the way to two possible job 1 modes, with and without D_1. In the following discussion we consider the latter job 1 mode without D_1. For simplicity, all task sets in this example are periodic, but those of jobs 1 and 3 are not independent. As previously noted in Section 2.3, and as indicated by Table 2, the dependencies of job 1 were resolved by adopting RM+PCP, while those of job 3 were avoided by a static CE.

Internal job scheduling requirements are first evaluated in terms of execution cycles before being translated into minimal frequency bandwidths. Bandwidth computation is clearly policy dependent, with an *optimal* π_j being that which would provide the minimal b_j bandwidth. For simplicity, we assume that the π_j policies of Table 2 have already been determined to be optimal, and we show only how the minimal b_j bandwidths were derived.

Net task computational requirements for task i are provided in Table 1 as numbers of execution cycles, n_i. When testing the schedulability of each job task set, the π_j schedulers must also consider worst-case overheads, o_j, for internal job scheduling, e.g. for task preemption, also provided in execution cycles. As indicated by Table 2, RM+PCP scheduling overheads are generally lower than EDF scheduling overheads and higher than CE scheduling overheads. To further accommodate worst-case job preemption overheads in a time-shared configuration, these o_j overheads must be augmented accordingly. In the above example, it was found that adding 2000 execution cycles to the dedicated o_j overheads was sufficient to cover two earliest job deadline updates and two potential job preemptions per job task.

To maximize processor utilization, we seek a job bandwidth, b_j, which is equal to the slowest processing speed required for τ_j to be schedulable by π_j while accommodating time-sharing overheads. To measure the loss of utilization due to time-sharing, we also compute f_j as the slowest processing speed required for τ_j to be scheduled on a *dedicated* processor. In both cases, minimal processing speeds must be determined by schedulability tests appropriate to each π_j policy. In the case of job 1 with π_1 = RM+PDP, we employed Rate Monotonic Analysis (RMA) [17, 18], to consider all factors in job execution, e.g. task periods, release times, deadlines, computation times, maximum block times, and context switching overheads. As previously noted, the latter three factors were provided in units of processor execution cycles, so that schedulability tests can be conducted for various processor speeds until the *minimal* speeds are determined. In an automated schedulability test, we can use a binary search strategy to converge to f_j.

Job 2, on the other hand, uses EDF scheduling, because all tasks in τ_2 are independent. EDF schedulability test are much simpler, letting us compute minimal processing speeds directly as a function of π_j characteristics. Letting T_i and C_i represent the period and computational times for task $i \in \tau_j$, we can compute the job utilization, u_j, by

$$u_j = \sum_{i \in \tau_j} C_i/T_i, \qquad (2)$$

whereupon τ_2 is schedulable if u_j does not exceed one [12]. So the minimal processing frequency for EDF is equal to $\sum_{i \in \tau_j} n_i/T_i$, n_i being the number of execution cycles required by task i, since this would maximize the processor utilization.

The C_i values in Table 1 provide an additional intuitive insight into why all π_j schedules are maintained. When running on a time-shared $4MHz$ processor, the required computation times are much shorter than when running on a processor with an assumed frequency of f_j. The f_j and b_j processing speeds of Table 2, essentially differ in the assumed computation times, C_i, for each task i. When computing f_j, the C_i values correspond to the number of computational execution cycles, n_i, plus the scheduling overhead execution cycles, o_j, on a dedicated processor. When computing b_j, the C_i values are augmented to include time-sharing deadline updates and worst-case job preemptions as determined by the $o'j$ values for shared processors. The α_j values in Table 2 are equal to the b_j/f_j ratios, thereby reflecting the very minor utilization losses introduced by time-sharing.

As indicated by the b_j values in Table 2 and Equation (1), a processor c with a capacity of β_c = 3.9 MHz can therefore reliably accommodate all three jobs using EDF time-sharing. As previously mentioned, job 1 has a second mode which also includes task D_1, which requires a slightly higher bandwidth which is still less than 2.1 MHz. Thus, with a capacity of 4 MHz we were actually able to time-share all three jobs in Figure 3, with job 2 operating at a higher reward mode which also required D_1.

2.6 Accommodating Interrupts

For the sake of completeness, we must point out that the above analysis must also consider processor cycles consumed by event handlers, e.g. the timer interrupt handler, such as *EventsHandler* in Figure 2. To simplify the example in Table 1 we have assumed there that these overheads are negligible[4]. Nevertheless, these overheads *are* readily accounted for in our model. The most straight-forward way would be to account for each job's event handlers *within* each job's bandwidth, by treating them as internal, event-driven, tasks with zero slack time[5]. Additional methods are discussed in [11].

2.7 Time-Shared Utilization

It follows from Equation (1), that overall processor utilization in the RtTS time-sharing scheme is generally only as good as the u_j utilizations provided by each π_j solution. In the above example, all u_j utilizations are near-optimal, so these high utilizations are fully maintained when time-sharing. If not for our time-sharing scheme, we would have to apply a uniform scheduling policy, e.g. EDF+PCP, to the union of *all* tasks in J_c, which could actually produce a *lower* total utilization, since it would have to consider worst-case blockages such as those which would be introduced by τ_3 without CE. Thus time-sharing can sometimes *boost* overall processor utilization when internal job utilizations, u_j, are efficient. In other cases, of course, a new integrated scheduling solution might increase processor utilization when u_j values are low. When dealing with complex task sets, however, seeking integrated solutions may be impractical, so that time-sharing in such systems is the only alternative. Thus the only relevant way to maximize processor utilization is to design the independent jobs in a manner which can produce high u_j factors. This would appear to be a very reasonable design objective for any software product.

We note, however, that there are ways to improve the time-shared utilization such that it is better than the *sum of its parts*, i.e. higher than the overall utilization which would have been obtained by running each Job j with a utilization of u_j on a dedicated processor with a processor speed of b_j. For $u_j < 1$, the surplus capacity on each dedicated processor could be exploited by each job only for soft real-time tasks belonging to it. When time-sharing on processor c, these surplus capacities

[4] Such assumptions are often reasonable because potential job preemption overheads are already covered by the o_j augmentations so that only the deadline update overheads must be considered.

[5] As implied by the EventsHandler procedure, job event handlers are actually called upon immediately, even if the job is not currently dispatched. This complies with EDF interleaving because their zero slack time would require a job context switch anyway.

can be better utilized collectively by using global *best-effort* [19] scheduling techniques to run the higher-reward soft tasks within the *entire* set of jobs, J_c.

Time-shared utilization is also improved by reducing capacity *fragmentation*, i.e. the loss of utilization due to the need to allocate processing capacities, β_c, which *exceed* the required minimal bandwidth, b_j. Take, for example, a job with two modes, the first requiring a bandwidth, b_j, of 8 MHz and the second being a higher service mode requiring $b_j = 11$ MHz. If available β_c capacities are 10 MHz and 20 MHz, then we could allocate the 10 MHz capacity to run this job in its first mode, or we could allocate the 20 MHz capacity to run the second mode. In the former case, capacity fragmentation would be 2 MHz, in the latter case it would be 9 MHz. Using time-sharing, we could actually run *two* of these jobs on the 20 MHz processor, one using the first mode and the other using the second mode, with a fragmentation loss of only 1 MHz. Thus real-time time-sharing can increase system value and and total processing utilization by reducing capacity fragmentation throughout the system.

3 Automated Meta-Control

The additive schedulability criteria (Equation (1)) is a unique feature of the RtTS time-sharing scheme which enables the software meta-control layer to simultaneously carry out near-optimal mode selection and job load balancing for a set of jobs, J, running on a set of processors, C. Each job, $j \in J$, has a set of selectable alternate modes, M_j, to choose from. As described in the black box paradigm of Section 2.1, each job must provide a bandwidth function, $b_j(m_j)$, and a reward function, $r_j(m_j)$, which fully encapsulate their current time-sharing requirements and their contribution to the system value, v, for each possible mode $m_j \in M_j$. The primary objective of the automated meta-controller is to select job modes $m_j \in M_j$ which will produce the *optimal* system value, v^*, i.e. the highest obtainable system value which is still schedulable. Let J_c represent the job subset allocated to processor $c \in C$, and let β_c denote the processor c capacity. Then, as described in Section 2, when using a time-sharing scheme, a given selection of job modes is *schedulable* if and only if there exists a partitioning of the job set J over the set of available processors C such that

$$\forall c \in C : \sum_{j \in J_c} b_j(m_j) \leq \beta_c. \qquad (3)$$

Finding such a schedulable partitioning is the job of the load-balancer, whereupon fully reliable time-sharing is guaranteed for all jobs in J.

A meta-controller *decision* consists, therefore, of a set of selected modes and a schedulable partitioning of jobs over processors. Job sets are dynamic. Job bandwidths and rewards are state-dependent. Thus meta-controller decisions must be automatically made with each altered job set and each state transition. In [9] we show that this automated meta-control problem is equivalent to a composite binpacking and zero-one multiple-choice knapsacking problem, which is strictly NP-hard. Nevertheless, we have devised two very effective approximation algorithms, QDP and G^2, capable of producing near-optimal mode selection and load-balancing solutions in fractions of a second. Table 3 lists typical performance levels and response times for meta-control decisions made for 12 or 24 jobs (njobs) with 4 modes per job, running on 3 or 5 processing nodes (nprocs), in cases which total bandwidths for maximum reward modes were either 2 or 4 times the total capacity of the system (maxload). As indicated by Table 3, performance for both G^2 and QDP are found to be considerably better than conventional *density greedy* (DG) [20], while being very comparable in their response times. These results enable us to reevaluate our decisions every few seconds to accommodate varying job sets and states with reasonable overhead. QDP relies primarily on dynamic programming, while G^2 is a gradient-greedy *sieve* function [21]. Both algorithms have the ability to tradeoff time for performance, and both can provide measures of performance with each meta-controller decision made. Detailed descriptions of these algorithms, simulations, and analytic methods, are all beyond the scope of this paper. Interested readers are kindly referred to Appendix of [22] for more information and references.

4 The RtTS Architecture

As illustrated in Section 1, complex real-time systems require a software control architecture capable of carrying out real-time time-sharing, critical load balancing, and optimal mode selection in an integrated manner. As described in previous sections, this is accomplished by the RtTS architecture using three software control layers which are fully independent. The job scheduling layer (see Figure 1) lets each job resolve its own scheduling problems using any arbitrary policy found to be effective. The job time-sharing layer uses an EDF policy to facilitate job time-sharing regardless of the policies used by the underlying job scheduling layer. Together, these two layers facilitate a practical real-time time-sharing scheme, which enables each job to be developed as an independent black-box, which maintains the efficiencies of the black-box solutions, and which provides us with a simple additive schedulability test for arbitrary job sets on any given processor. The automated meta-control layer can then simultaneously facilitate near-optimal mode selection and load-balancing without being concerned about the underlying real-time scheduling policies being used.

Early on, we described a portable, fault-tolerant, integrated multimedia system, with dynamic job sets, modes, states, and processor capacities. System users may request different sets of concurrent jobs at any time, and committed jobs must continue to run seamlessly when switching from one set of jobs to another. Such systems clearly require powerful and automated software control mechanisms for ensuring the schedulability of these jobs while maximizing some notion of system value. Probably the most acute difficulty in this system is in the need to cater to dynamic sets of *hard* tasks. The human brain is quite sensitive to audio and video input, so the steady flow of audio samples and video frames, and an adequate synchrony between audio and video, are all *essential* to such systems. The authors are not aware of any existing dynamic load-balancing solution which can meet these challenges.

As we have already shown, the RtTS architecture has been designed to support *all* of these formidable requirements using practical methods. Nevertheless, the following limitations must be considered. The software meta-control layer, for example, cannot respond instantly and decision epochs should be adequately spaced to maintain reasonable overheads. These limitations are quite acceptable in the multimedia example since all changes in the active job set and all state transitions do not occur at high rates and they can be deferred for fractions of a second. Other important assumptions in our current architecture include that the computed job bandwidths are transparent to job placement at any of the processors, and that job migration and mode transition costs can be overlooked. These assumptions are also quite reasonable for shared-memory platforms and when mode transitions consist of merely altering various rates and resolutions. A nice feature in the automated meta-control

model is that for each job, we can consider only those processors and modes which would not require costly migrations or transitions. For quicker responses to state-induced overloads, we could initially reconsider only local mode selections without load balancing, and then improve the system value with a complete global evaluation after the overload has been contained. The RtTS architecture is therefore a very flexible framework which can be customized to meet the specific needs of each system.

From a software engineering perspective, the RtTS architecture also has significant advantages in its ability to scale well for larger systems, and in its inherent portability. Time-sharing schedulability tests for J_c have $O(N)$ complexity, where N is the maximum number of jobs in J_c. As described in [9] software meta-control complexities are somewhat higher, but still very reasonable. Internal job scheduling is accomplished without prior knowledge of the number of platform processors and their capacities. When porting to a platform, all we need to know are the minimal bandwidth requirements for each job to determine whether there is any processor capable of hosting it. System values automatically improve as technology advances and platform capacities increase. Such characteristics are rare in conventional real-time solutions.

5 Conclusions

We have outlined a practical *job*-oriented approach to real-time software control, found to be particularly efficient for complex real-time systems. Rather than attempt to online tackle integrated and complex real-time scheduling problems, we show how ready solutions to smaller and simpler problems can be time-shared while maintaining their scheduling integrities and utilizations. Rather than attempt to resolve dynamic load-balancing at the individual task level, we show how job-level load-balancing can be integrated with value-driven mode selection and still be resolved by very practical and effective approximation algorithms. We have also demonstrated that the dynamic RtTS control architecture is appropriate for dynamic sets of *critical* tasks, which have no known alternate solutions.

The practicality and efficiency of the approach have already been proved by extensive simulation and analysis. The benefits of this approach are most readily illustrated in a dynamic, multiple job, shared-memory system. Nevertheless, we believe that it can be extended to benefit a much wider range of complex systems, including fault-tolerant systems. An ATLAS testbed [2], is currently being implemented to explore these possibilities.

Acknowledgments

The authors wish to thank Amos Israeli, Avi Mendelson, Jack Stankovic, and Neeraj Suri, for reviewing earlier versions of this paper.

References

[1] J. A. Stankovic and K. Ramamritham, "Editorial: What is predictability for real-time systems?", *Journal of Real-Time Systems*, vol. 2, no. 4, pp. 247–254, November 1990.

[2] J. Jehuda and D. M. Berry, "A top-layer architecture for ATLAS", Tech. Rep., Department of Electrical Engineering, Israel Institute of Technology, Technion, Haifa 32000, ISRAEL, November 1994, EE Pub. 946.

[3] J. A. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time operating systems", *ACM Operating Systems Review*, vol. 23, no. 3, pp. 54–71, July 1989.

[4] S. R.-T. and J. P. Lehoczky, "On-line scheduling of hard deadline aperiodic tasks in fixed priority systems", in *Proceedings 11th Real-Time Systems Symposium*, December 1993, pp. 160–171.

[5] C. McElhone, "Adapting and evaluating algorithms for dynamic schedulability testing", Tech. Rep., Department of Computer Science, University of York, England, February 1994.

[6] G. Koren and D. Shasha, "MOCA : A multiprocessor on-line competitive algorithm for real-time system scheduling", *Theoretical Computer Science*, vol. 128, pp. 75–97, 1994.

[7] T. Bihari and K. Schwan, "A comparison of four adaptation algorithms for increasing the reliability of real-time software", in *Proceedings Ninth Real-Time Systems Symposium*, March 1988, pp. 232–243.

[8] K. Schwan, T. Bihari, B. Weide, and G. Taulbee, "High-performance operating system primitives for robotics and real-time control systems", *ACM Transactions on Computer Systems*, vol. 5, pp. 189–231, August 1987.

[9] J. Jehuda and A. Israeli, "Automated meta-control for adaptable real-time software", Tech. Rep., Department of Electrical Engineering, Israel Institute of Technology, Technion, Haifa 32000, ISRAEL, November 1994, EE Pub. 943.

[10] M. L. Dertouzos, "Control robotics: The procedural control of physical processes", in *Proc. IFIP Congress*, 1974, pp. 807–813.

[11] J. Jehuda and G. Koren, "Hybrid bandwidth scheduling for distributed real-time systems", Tech. Rep., Department of Electrical Engineering, Israel Institute of Technology, Technion, Haifa 32000, ISRAEL, January 1995, EE Pub. 945.

[12] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[13] M.-I. Chen and K.-J. Lin, "Dynamic priority ceilings: A concurrency control protocol for real time systems", *Journal of Real-Time Systems*, vol. 2, no. 4, pp. 325–346, November 1990.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization", *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[15] T. P. Baker and A. C. Shaw, "The cyclic executive model and Ada", in *Proceedings Ninth Real-Time Systems Symposium*, 1988, pp. 120–129.

[16] C. Locke, "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives", *Journal of Real-Time Systems*, vol. 4, no. 1, pp. 37–54, 1992.

[17] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic Publishers, 1993.

[18] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior", in *Proceedings IEEE Real-Time Systems Symposium*, December 1989, pp. 166–171.

[19] C. D. Locke, *Best-Effort Decision Making for Real-Time Scheduling*, PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, Pittsburg, Pa. 15213, 1986.

[20] M. R. Garey and D. S. Johnson, *Computers and Intractability - Guide to the Theory of NP-Completeness*, Bell Telephone Laboratories, Inc., 1979.

[21] J.-Y. Chung, J. W. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results", *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1156–1173, September 1990.

[22] J. Jehuda, G. Koren, and D. M. Berry, "A time-sharing architecture for complex real-time systems", Tech. Rep., Department of Electrical Engineering, Israel Institute of Technology, Technion, Haifa 32000, ISRAEL, May 1995, EE Pub. 965.

Table 1. A Sample Job Set

	Job 1				Job 2			Job 3		
Task	A_1	B_1	C_1	D_1	A_2	B_2	C_2	A_3	B_3	C_3
Period (msecs)	40	60	140	200	80	120	180	120	120	120
Deadline (msecs)	40	64	140	120	80	120	180	120	120	120
Net Cycles n_i	30K	46K	46K	6K	33K	33K	33K	18K	36K	54K
Max Blocking Cycles		8K								
C_i Comput. (ms) @ 4 MHz	8	12	12	4	9	9	9	5	10	15
C_i (ms) @ f_j (see Table 2)	16	24	24	8	40	40	40	20	40	60
B_i Blocking (ms) @ 4 MHz		4							3	4
B_i (ms) @ f_j (see Table 2)		8							12	16

Table 2. Computing Job Bandwidths

job number		j	1	2	3
job scheduling policy		π_j	RM+PCP	EDF	CE
job utilization factor		u_j	0.97	1.00	1.00
job scheduling overhead cycles	dedicated	o_j	200	1000	20
	shared	o'_j	2200	3000	2020
min processor freq. (MHz)	dedicated	f_j	1.94	0.88	0.95
	shared	b_j	2.00	0.90	1.00
time-sharing overhead ratio		α_j	1.03	1.02	1.05

Table 3. Typical Meta-Control Performance and Response Times

				Performance			Response Times (msecs)		
njobs	nmodes	nprocs	maxload	DG	G^2	QDP	DG	G^2	QDP
24	96	5	2.0	0.569	0.989	0.990	142.461	85.859	269.161
			4.0	0.625	0.978	0.974	96.316	63.547	248.482
12	48		2.0	0.552	0.989	0.981	144.224	82.351	271.383
			4.0	0.609	0.978	0.976	94.883	58.236	255.837
24	96	3	2.0	0.634	0.977	0.973	77.158	47.939	244.434
			4.0	0.685	0.963	0.967	48.938	48.413	242.036
12	48		2.0	0.657	0.974	0.973	80.165	48.619	246.296
			4.0	0.680	0.968	0.968	61.178	36.702	242.270