Report Number: WUCSE-2004-69

2004-11-01

Design and Performance of Configurable Endsystem Scheduling Mechanisms

Tejasvi Aswathanarayana, Douglas Niehaus, Venkita Subranmonian, and Christopher Gill

This paper describes a scheduling abstraction, called group scheduling, that emphasizes fine grain configurability of scheduling system semantics. The group scheduling approach described and evaluated in this paper is an extremely flexible framework within which a wide range of scheduling semantics can be expressed. The paper describes both the OS and middleware based implementations of the framework, and shows through evaluation that they produce the same behavior from a non-trivial set of application computations. Further, the evaluation shows that the framework can easily support application-aware scheduling algorithms to improve performance.
... Read complete abstract on page 2.

Design and Performance of Configurable Endsystem Scheduling Mechanisms

Tejasvi Aswathanarayana, Douglas Niehaus, Venkita Subranmonian, and Christopher Gill

Complete Abstract:

This paper describes a scheduling abstraction, called group scheduling, that emphasizes fine grain configurability of scheduling system semantics. The group scheduling approach described and evaluated in this paper is an extremely flexible framework within which a wide range of scheduling semantics can be expressed. The paper describes both the OS and middleware based implementations of the framework, and shows through evaluation that they produce the same behavior from a non-trivial set of application computations. Further, the evaluation shows that the framework can easily support application-aware scheduling algorithms to improve performance.

2004-69

Design and Performance of Configurable Endsystem Scheduling Mechanisms

Authors: Aswathanarayana, Tejasvi; Subramonian, Venkita; Niehaus, Douglas; Gill, Christopher

Abstract: This paper describes a scheduling abstraction called group scheduling, that emphasizes fine grain configurability of scheduling system semantics. The group scheduling approach described and evaluated in this paper is an extremely flexible framework within which a wide range of scheduling semantics can be expressed. The paper describes both the OS and middleware based implementations of the framework, and shows through evaluation that they produce the same behavior from a non-trivial set of application computations. Further, the evaluation shows that the framework can easily support application-aware scheduling algorithms to improve performance.

Type of Report: Other

Design and Performance of Configurable Endsystem Scheduling Mechanisms

Tejasvi Aswathanarayana and Douglas Niehaus
Information and Telecommunication Technology Center
University of Kansas, Lawrence, Kansas
{tejasvi,niehaus}@ittc.ku.edu

Venkita Subramonian and Christopher Gill
Distributed Object Computing Group
Department of Computer Science and Engineering
Washington University, St. Louis, MO
{venkita,cdgill}@cse.wustl.edu

***Abstract*— This paper describes a scheduling abstraction, called *group scheduling*, that emphasizes fine grain configurability of scheduling system semantics. The group scheduling approach described and evaluated in this paper is an extremely flexible framework within which a wide range of scheduling semantics can be expressed. The paper describes both the OS and middleware based implementations of the framework, and shows through evaluation that they produce the same behavior from a non-trivial set of application computations. Further, the evaluation shows that the framework can easily support application-aware scheduling algorithms to improve performance.**

I. Introduction

Distributed real-time and embedded (DRE) systems are increasingly common across a wide range of application domains. Constraints on application computation behavior in DRE systems are becoming ever more detailed and diverse as the applications become more complex. This trend is illustrated by the range of constraints associated with the application domains we have been considering over the last year, which include: industrial automation, military command and control, and life science laboratory experiment control/management.

The challenge these systems pose for the application designer is the increasing complexity of the application semantics and the resulting difficulty of expressing those semantics in terms of commonly available programming models, within which the scheduling model is the most prominent aspect. The challenge addressed by this paper, which faces both application and systems designers, is that no single scheduling model is adequate for expressing the full range of scheduling semantics for DRE endsystems. We use the term "DRE endsystem" to refer to an individual computational node within a (possibly distributed) DRE system.

Traditional DRE endsystem designs have assumed that the underlying OS provided a single scheduling model. The most typical endsystem scheduling model is some form of priority scheduling. DRE endsystem designers have tended to provide a single scheduling model for three major reasons. First, priority scheduling is relatively simple to understand and to implement. Second, the execution semantics of many DRE applications are appropriately expressed in terms of priorities. Third, the range of application semantics for which priority scheduling can be used has been extended by the development of theories for mapping application semantics to priority assignments. Examples of this approach include: rate monotonic analysis and scheduling [11], earliest deadline first [11], maximum urgency first [18], and least laxity [18], [12] schemes.

The flexibility of priority scheduling for DRE endsystems has obviously been a design strength. However, it has also been a weakness because endsystem designers have continued to provide this fixed scheduling model even when it placed an undue burden on the application designers to map application semantics to priority assignments. For example, Linux provides a dynamic priority scheme fairly typical of general purpose systems by default. It also provides a fixed priority (SCHED_FIFO) scheme for use by computations that take precedence over computations within the default scheduling class. While this is appropriate for a fairly wide range of applications, priorities are not adequate to express many of the ever more complex DRE application semantics

Endsystem scheduling frameworks have historically increased their flexibility by increasing the configurability of low level resource arbitration mechanisms, rather than focusing on direct support for the application level resource control requirements. However, this places an undue burden on application developers who are then responsible for expressing the application-level resource requirements in terms of the low level mechanisms whose semantics may differ considerably.

In this paper we focus on the problem of providing an endsystem scheduling framework within which we can maximize the correspondence between the application scheduling semantics and the semantics of the endsystem scheduling model supporting the application. We call our framework "Group Scheduling" because it emphasizes representation of the groups of computation components comprising an application. Further, it recognizes the diversity of scheduling semantics by permitting each group to use the scheduling algorithm most appropriate to the set of computations it controls. Hierarchic composition of groups permits the DRE system designer to construct the scheduler for the system as a whole.

Group Scheduling thus provides an extremely flexible model that can be used to express an extremely wide range of application and endsystem scheduling semantics. The group scheduling model emphasizes the ease and clarity with which developers can express the scheduling semantics of the appli-

cation in operational terms. Further, as we demonstrate in this paper, the group scheduling model can be implemented at both the OS and Middleware levels.

Implementation of the group scheduling model raises a number of important issues, including: (1) the fidelity with which application resource requirements can be expressed and enforced, (2) the portability of the framework across a range of OS platforms, and (3) the degree of augmentation of commonly available system capabilities required to support application semantics in particular contexts. We address these issues in the rest of the paper. However, it is also important to note that in a DRE system, each endsystem must not only control access to its local resources, it must also participate in end-to-end resource allocations for distributed application computations crossing endsystem boundaries. In collaboration with colleagues at URI and Ohio University, we have developed and evaluated a multi-level scheduling and resource management architecture.

The rest of the paper first discusses the motivations and challenges of our work in Section II, and the essential aspects of the group scheduling framework in Section III. We then describe the implementation details for the work describe int his paper in Section IV. Section V presents the experimental results demonstrating the validity of the claims we have made about our system. We then discuss related work in Section VI, while Section VII presents our conclusions and discusses future work.

II. Motivation and Challenges

Many DRE systems involve a non-trivial number of computations whose activity must be coordinated. One common scenario is sets of computations operating on one or more streams of data. Each computations can often be viewed as a *pipeline* of computation components. Examples of such systems that we have been using to motivate and guide the work described here include: multi-channel sensor fusion for DRE systems such as sensor nets or laboratory science, multi-channel audio data mixing, or sensor data processing in military applications such as time constrained targeting (TCT). These and other classes of applications have a number of common characteristics that we have abstracted into a representative application architecture we used to help with the design and evaluation of our OS and middleware based group scheduling framework.

In the rest of this paper we focus on an example, drawn from the DARPA PCES canonical video processing challenge problem, which exhibits several characteristics emphasizing the need for flexible scheduling support for many DRE application areas. (1)Each computation is implemented by a set of components operating on a stream of data elements in the form of a pipeline. These sets of computation components may be supported on a single computer, or by several computers in a distributed system. For the results reported here we have concentrated on computations supported by a single endsystem. (2)Data elements, called frames, move through the pipelines as they are processed. (3)Processing of a frame by a computation component can require widely varying CPU time. This can be true of many applications, but is particularly true of video processing applications. (4)Multiple pipeline computations are supported by the DRE system as a whole. This is an important property of the experimental conditions because it makes the scheduling problem both realistic and complex enough to show that priority driven and other popular scheduling algorithms have important limitations.(5)Balanced progress by multiple computations can be an important application behavior constraint. This is true of many applications which fuse data from several sources, but is particularly easy to see in the case of supporting multiple video streams that are meant to be viewed concurrently.

Resource contention of three types must be resolved by the scheduling policy of the DRE endsystem: (1)conflict between computation components, (2)conflict between computations, and (3) conflict between DRE application computations and other computations on a given endsystem. The group scheduling framework provides the DRE system designer with the ability to address these types of conflicts using separate group representations, and thus individual scheduling policies.

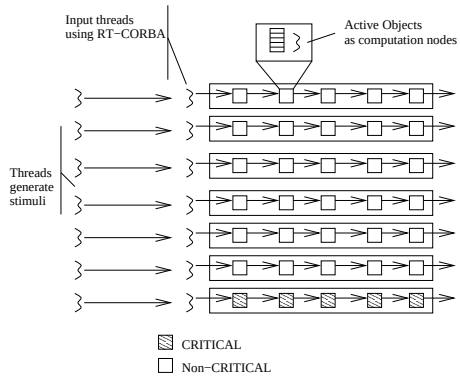

Fig. 1. Computation Pipeline

Our example application consists of a collection of computation pipelines as shown in Figure 1. The urgency of computations performed by a pipeline determines the *criticality* of that pipeline. For example, each pipeline could represent a series of image processing computations on an image sent by a UAV to a shipboard computer. If a particular pipeline is identified as sending a sensitive image like an enemy battle-tank, the shipboard computer increases the *criticality* of this pipeline. There are several interesting performance metrics for this application. First, the processing of a critical stream must take preference over the processing of others. Second, the processing of streams within the same class must be *balanced* since we assume they represent streams of video that will generally be viewed together. Third, we are interested in the overall frame throughput of the applications. Finally, we are also interested in how the pipeline based computations can share CPU resources with other computations on the system.

Each pipeline is implemented as a set of active objects [10] under the ACE ORB (TAO)[3]. An active object is essentially the combination of a worker thread and a queue on which the work items are placed. The set of frames processed by each

pipeline is created by a separate thread executed periodically. It is important to note that a thread within TAO middleware receives the messages, and must be included in the group that controls the execution of the pipeline to properly control each computation. We use the RTCORBA features in TAO, including the thread-pools [15], to achieve individual control of each pipeline. We define a computation stream as a combination of an RTCORBA thread and a pipeline of computation nodes (A computation stream = Message receiver RTCORBA thread + Pipeline threads that process the message). Each frame is received by the computation stream's RTCORBA thread which waits on a socket that is advertised to the frame generating thread for the stream. The nature of the TAO implementation dictates that the sending of a frame by the source thread, through the enqueuing of the frame at the input of the first pipeline stage, and the return to the source thread completes the TAO message exchange operation. This then permit TAO to start the execution of the active objects which will pass the frame through the pipeline.

Note that the combination of balanced progress (5) and variable execution time (3) constraints is particularly difficult for traditionally popular scheduling algorithms to satisfy in this example. The group scheduling framework allows us to explicitly represent multi-level computation structures as hierarchically defined groups and to associate scheduling decision functions having appropriate semantics with each group. In turn this enables us to create a representation of the application semantics that is both easy to understand and effective during execution.

III. Group Scheduling Model

The group scheduling model used to describe the scheduling semantics in this paper is a simple, but important, variation on hierarchical descriptions of scheduling that have been developed by several other researchers [17], [16], [8]. What distinguishes the group scheduling framework is (1) an emphasis on arbitrary grouping of computation components for representation of application computations as well as other reasons, (2) association of arbitrary scheduling decision functions with the groups to produce an extremely flexible scheduling framework capable of clearly and easily expressing a wide range of scheduling semantics, and (3) existence of both OS and middleware implementations of the framework in the popular open source Linux platform.

A. Group Scheduling Implementation Core

A group is defined as a collection of computation components with an associated scheduling decision function (SDF) that selects among the group members when invoked. Each member of a group can have information associated with it, as required by the SDF. Groups can also be members of other groups, thus supporting hierarchical composition of more complex SDFs, culminating in the creation of an SDF for the system as a whole, the system SDF (SSDF). A subset of the computations on a system can be placed under SSDF control because both the OS and middleware implementations of group scheduling permit the default Linux scheduler to make a decision if the SSDF does not make a choice. Computations can be placed under exclusive control of the SSDF or joint control of the SSDF and the default Linux scheduler as the user desires.

The group scheduling framework emphasizes modularity of the SDF implementations and thus makes it relatively easy for users to implement their own SDFs if the library of available functions does not include one matching the scheduling semantics they desire. The group scheduling framework can thus easily subsume all of the popularly scheduling models by providing matching SDFs. However, it can support application specific and otherwise highly specialized scheduling semantics with equal ease, as illustrated by the frame-progress scheduler used to control frame processing by pipelines in Scenario 3 described in Section V. The application specific specialization of the frame-progress scheduler depends on application specific information about computation progress that can be supplied through the scheduling parameter modification interface, or through use of a memory segment shared among all applications and the SDF.

The semantics of the OS and middleware implementations of the group scheduling model are the same in most ways, but differ in some important aspects that are described next.

B. OS Level Group Scheduling

The OS level implementation of group scheduling enjoys several advantages over the middleware version. First, it provides integrated control over *all* computation components in the Linux system, including hardware interrupt handlers, soft interrupt (Soft-IRQ) handlers, tasklets and bottom halves [6]. In contrast, the middleware implementation can only control threads. Second, the overhead of performing the context switch is lower in the OS implementation because the scheduling decision made by the SSDF and the actual context switch are both in OS context. The middleware version must use an indirect mechanism, priority manipulation and signals to accomplish the same objective. Third, the SDFs have a much wider range of computation and system state information available at minimal cost because the SSDF executes in OS context.

These advantages enjoyed by the OS implementation are real but not definitive for the frame processing example implementation discussed in this paper. However, other applications involving the integrated group scheduling control of interrupt handlers and network protocol processing are examples of application semantics that can only be implemented under the OS group scheduling implementation. the obvious disadvantage of the OS group scheduling implementation is that it requires access to the source of the OS and a wide range of subtle modifications to the OS management of computation components to produce the unified scheduling framework.

C. Middleware Level Group Scheduling

The group scheduling API is consistent across middleware and OS implementations, but the mechanisms differ because

the middleware version requires several utility threads, of which the most important are the *scheduling* thread which evaluates the SSDF, and the *block catcher* which helps detect state changes of the controlled threads which are an implicit part of the OS group scheduling implementation. The API which permit application code to provide scheduler-specific parameters and to construct the SSDF are socket based in the middleware version as opposed to the *ioctl* based calls in the OS implementation. The most significant limitation of the middleware version is that it lacks the easy access to computation state (RUNNABLE, BLOCKED, etc.) enjoyed by the OS version. Instead the middleware version must go to some lengths, described in Section IV, to track the computation state change. As shown in Section V, the behavior of the middleware version closely matches that of the OS version, but must pay a price in context switch overhead. However, the middleware version enjoys the advantage of significantly easier portability to other platforms since it only depends on commonly provided OS capabilities.

IV. Middleware Implementation

This section provides several implementation details for the middleware version of group scheduling that are relevant for understanding the evaluation and results of the example applicaiton described in Section V. In the following discussion, we refer to threads under the control of the middleware scheduler as *controlled* threads. We use the native OS priority model as an enforcement mechanism for the decisions made by the SSDF. To achieve this, we use five different priority levels as shown in Table I. `MAX_PRIO` is the maximum priority in the `SCHED_FIFO` scheduling class.

Other than the controlled threads, the group scheduler uses 4 internal threads, listed in Table I for its own functioning. The Reaper thread helps in shutting down and gaining control of the system in case of a fault hierarchy. Scheduling is done in the context of the Group Scheduler thread. The Block Catcher thread is a thread that helps us to detect blocking of a controlled task and this runs immediately after a controlled task blocks. Requests to the group scheduler are handled by the API thread.

TABLE I
PRIORITY LEVELS FOR MIDDLEWARE SCHEDULING

Reaper thread	`MAX_PRIO`
Blocked Controlled threads	`MAX_PRIO-1`
Group Scheduler threads (SSDT and API)	`MAX_PRIO-2`
Currently Scheduled thread	`MAX_PRIO-3`
Block Catcher thread	`MAX_PRIO-4`
Other Controlled threads	`MAX_PRIO-5`

Figures 2 and 3 illustrate the sequence of events that take place during blocking and unblocking of a thread.

A controlled thread is about to make a system call that may cause the thread to block (1). Since we are using the ACE toolkit that provides a wrapper layer for all system calls, these changes were very much localized and hence this approach is highly scalable. All system calls that may block are wrapped as follows.

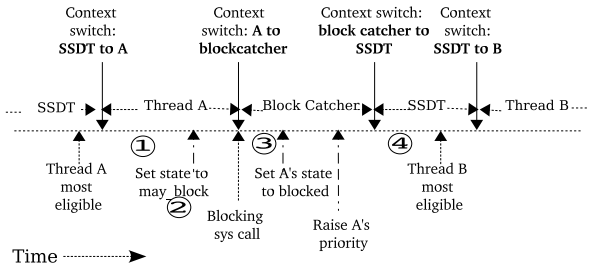

Fig. 2. Scenario when a controlled task is about to make a blocking system call

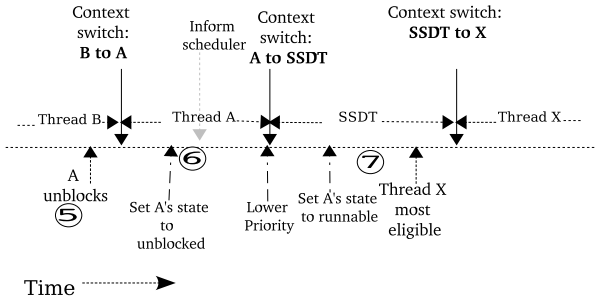

Fig. 3. Scenario when a blocked controlled task unblocks

```
wrapped_system_call(){
  before_system_call_hook()
  system_call()
  after_system_call_hook()
}
```

The *before_system_call_hook* sets the status of the calling thread to `MAY_BLOCK` (2). This is to explicitly assist the scheduler with information about possible blocking. If the thread *really* blocks, the block catcher thread wakes up. The status of the blocked thread is changed to `BLOCKED` (3). Apart from changing the status of the blocked thread to `BLOCKED`, the block catcher bumps up the priority of the blocked thread to that of the blocked controlled threads in Table I. The block catcher thread wakes up the scheduler thread (4). The scheduler thread calls the SSDF which picks up the most eligible thread to run.

After a (possibly blocking) system call returns, a thread calls the *after_system_call_hook* function (5). This function ascertains whether the thread was really blocked on the system call. If it did really block, its status is changed from `BLOCKED` to `UNBLOCKED`. If the thread had not blocked on the system call, the status of the thread is changed to `RUNNABLE` and the thread continues to run.

The scheduler thread is woken up (6) to indicate the unblocking of a thread and hence the need for a scheduling decision. The scheduler changes the status of (possibly multiple) `UNBLOCKED` threads to `RUNNABLE`, so that they could all be considered for scheduling. The scheduler chooses the most eligible thread to run (7).

Figure 4 summarizes the different states of a controlled thread. A thread is in the `RUNNABLE` state if it is ready to run and is an eligible candidate that could be picked up to run by the SSDF. The execution of these threads are controlled by sending SIGSTOP and SIGCONT signals. A

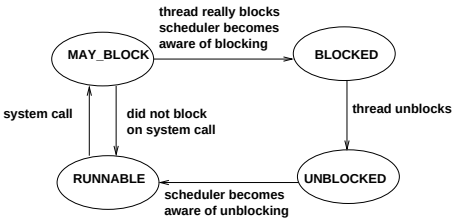

Fig. 4. State Transition for Threads

controlled task that is about to make a system call that *could* block, makes a state transition to the MAY_BLOCK state. If a controlled task *really* blocked after making a system call, then its state is marked as BLOCKED. Once the system call unblocks the controlled thread unblocks and starts running again changing its state to UNBLOCKED. The scheduler gets informed immediately about the unblocking and changes its state to RUNNABLE again.

V. Evaluation

We now proceed to evaluate our implementation described in Section IV in the context of the challenges provided by the motivating application described in Section II. The goal of our evaluation is to demonstrate the following -

- Necessity and Sufficiency of the group scheduling paradigm
- Several applications require *simple* policies which are quite hard (sometimes impossible) to express in terms of existing lower level scheduling policies
- Group scheduling paradigm raises the level of abstraction for specifying application policies so that applications can directly express their policies in terms of the computation hierarchy offered by the group scheduling programming model
- Efficient implementation of group scheduling middleware is possible which makes our solution directly applicable to a wide variety of platforms and operating systems

For this evaluation, we put some raw bytes as the payload in the stimuli for the application. Realistically, this could be an image frame that is being sent by an image source. In the following discussion, we use the term *frames* instead of stimuli. We state a simple (and quite realistic) application policy for our example application -

1) Preference to critical streams over non-critical streams
2) A computation stream needs to be drained as fast as possible.
3) Receiving an input frame is more important than frame processing in a pipeline
4) Balanced progress, in terms of number of frames processed, among the non-critical streams in the face of variable computation times for frames passing through the pipeline

A. Qualitative Evaluation

It is obvious that expressing the above high-level application policies using existing low-level scheduling policies and priority assignments is tedious and error-prone. Moreover, they are inadequate to express the notion of balanced progress due to lack of *a* priori knowledge of varying execution times on the computation nodes.

Group scheduling paradigm is a generalization of conventional scheduling paradigms and hence *sufficient* to express existing low-level scheduling policies like FIFO and round-robin. Moreover, it enables us to let application-specific policies be directly expressed as scheduling policies, thus closing the gap between higher level application policies and lower level scheduling policies for computation elements. Not only does this raise the level of abstraction for the application developer, but it is also *necessary* for several applications.

We now demonstrate the *sufficiency* aspect as to how we can express our application policies in terms of the group scheduling computational model described in Section III. In the course of doing this, we find the inadequacy of existing low-level policies to capture application policy (4) which then leads to the *necessity* aspect of group scheduling.

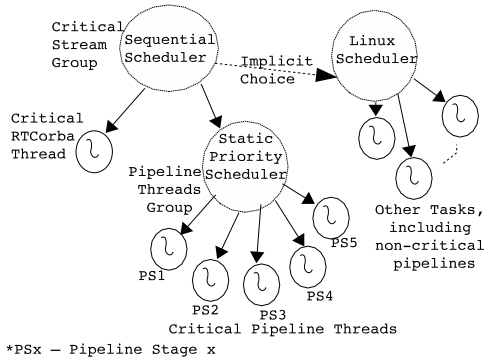

Fig. 5. Group hierarchy for Scenario1

1) Scenario1: In this scenario, we try to map the four application policies to existing lower-level scheduling policies expressed within the group scheduling computational model. The group scheduling model allows us to specify each of the policies intuitively. To express policy (1) we want the scheduler to give preference to the critical stream over all other threads in the system including threads in the non-critical stream. The Group scheduling computational model, shown in Figure 5 is used to specify this policy. A *Critical Stream* group is created to represent a critical stream. By default, any thread registered with the group scheduler is considered to be more eligible than one that is not registered and hence the group scheduler picks up an unregistered thread for execution only if none of the registered threads are in the RUNNABLE state. This is indicated in Figure 5 by the arrow marked “Implicit choice”.

To specify policy(3), we intuitively divide the processing of a frame in two computational parts - (1) input processing of a frame (by an RTCORBA thread) and (2) the pipeline computation. The computation threads in the critical pipeline are represented by the *Pipeline Threads* group. Between these two computations, the input processing and hence the RTCORBA thread has to be given preference, as per policy (3). Hence a sequential scheduling policy is chosen for the *Critical Stream* group. In this policy, threads/groups are given

preference in the order of their registration (joining a group) with the group scheduler. In all the scenarios that we describe here, the RTCORBA thread joins its parent group (in this case, *Critical Streams* group) before the *Pipeline Threads* group and hence is preferred over the latter.

To specify policy(2), we choose a static priority scheduler, with priorities increasing across the pipeline stages with the last stage having the highest priority. This ensures that a message flows across the pipeline before a new message will be processed.

For policy (4), in this scenario, we rely on the default policy of the vanilla Linux scheduler since all the non-critical stream threads are under its control.

2) Scenario2: This scenario attempts to address policy (4), that we left to the control of the vanilla Linux scheduler in Scenario1. To this end, the non-critical streams are also brought under the control of the group scheduler. Intuitively, we divide the different non-critical streams into groups as shown in Figure 6, each *Pipeline Thread* group is similar to that in Scenario1. The critical stream should be given preference over this collection of non-critical streams. To express this policy, the non-critical stream groups are all grouped together under the group *Non-critical Streams* and this group gets less preference when compared to the *Critical Streams* group.

To achieve balanced frame progress (policy 4), we try to map this policy in terms of a round-robin scheduling policy for the *Non-critical streams* group. Note that Figure 6 shows the group hierarhcy for both Scenerios 2 and 3 - Scenario 2 uses the Round-Robin(RR) scheduler for the *Non-critical streams* group and Scenario 3 uses the Frame progress scheduler. A specified quantum of time is allotted for each non-critical stream hoping that this would balance the progress among the different streams.

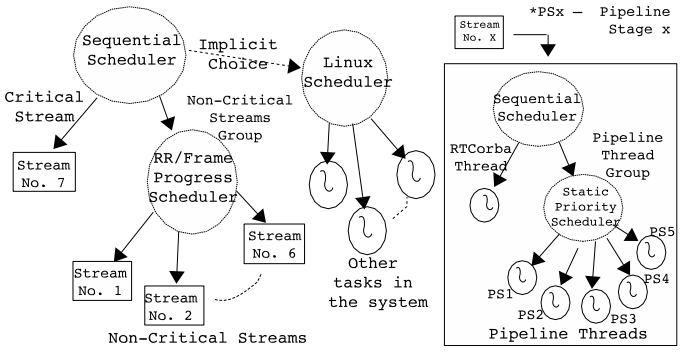

Fig. 6. Group hierarchy for Scenario2/3

3) Scenario3: In Scenario2, we map policy(4) to a round-robin scheduling policy among the non-critical streams. If the computation times on a pipeline node can vary based on the type of frame that it is processing, then it creates an imbalance in the number of frames that can be processed by different pipelines within the same time duration. The requirement of balanced progress is very often true in the kinds of applications discussed in Section II. Hence it is *necessary* to be able to specify such policies in terms of higher level programming models. Group scheduling provides such a programming model.

The group scheduling hierarchy for this scenario is shown in Figure 6, which is similar to Scenario2, except for the scheduling policy that is associated with the *Non-critical Streams* group. In this scenario, an application specific scheduler is used for scheduling the non-critical pipelines. The aim of our application specific scheduler is to maintain balance of progress across multiple pipelines. The scheduler ensures that no non-critical stream gets more than *N* stimuli ahead of the other non-critical streams, where *N* is application defined. This can also be observed visually using the visualizer tool described in Section V-B

Having demonstrated the very intuitive style of specifying application policies, we now proceed to demonstrate that our group scheduling implementation indeed satisfies the application policies. We used a mix of visualization tools and quantitative analysis for this purpose.

B. Instrumentation, Logging and Visualization tools

We used the Datastream Kernel Interface (DSKI) and Datastream User Interface (DSUI) logging framework and postprocessing tools to instrument our application, the group scheduler and the Linux kernel [14]. We used the visualizer tool to generate the execution interval diagram, shown in Figure 7, to qualitatively examine how well our group scheduling implementations enforce the application specified policies discussed previously. For example, Figure 7 shows a screenshot of the visualizer tool displaying information mined using post-processing tools from the logged DSUI and DSKI events from an experimental run of Scenario3 using the middleware implementation of the group scheduler.

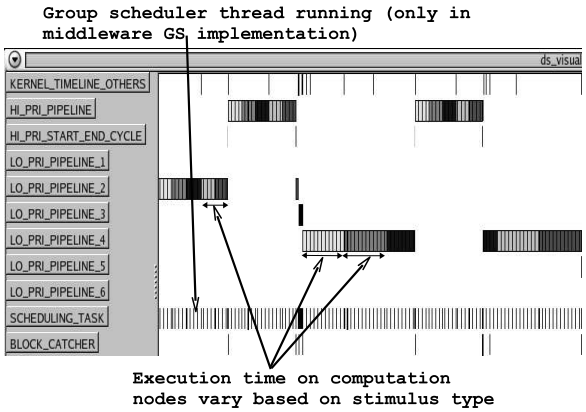

Fig. 7. Execution Intervals of tasks in scenario3 - Middleware

The execution time-line depicted in Figure 7 clearly verifies this behavior. The visualizer has been configured to show the execution time-lines for all the pipelines.

Note that different message processing times could be possible for the different pipelines. In our experiment, as seen in Figure 7, we chose message types with lower processing time to be passed through pipelines 3 and 5, although this is configurable in our experimental setup.

A non-critical pipeline, when chosen to run, runs till it processes a stimulus completely, *i.e.* the stimulus passes through

all stages of the pipeline, before any other pipeline starts processing its own stimuli. This is also confirmed through the visualizer. None of the non-critical pipeline executions overlap. Arrival of a stimuli for the critical stream makes the threads related to the critical stream to run. As seen in the visualizer, pipeline1 was processing a stimulus when a stimuli for the critical stream arrived. This is indicated by an event in the START-END cycle row of the visualizer. This causes the control to shift to the tasks in the critical stream.

The group scheduling thread and the block catcher thread seen in the last two rows of the visualizer incur context switch overhead in the middleware implementation as opposed to the kernel-based implementation of the group scheduler, which does not have this extra overhead. The execution of the group scheduling thread is very frequent as seen in the visualizer. This is because the group scheduling task is run atleast once every time quantum, which is set to 10msec in our implementation, and which is equal to the Linux schedulers quantum in the 2.4 kernel series.

C. Quantitative Evaluation

The purpose here is to evaluate our group scheduling implementation (both kernel and middleware) in the face of background load competing with the application, in the face of non-critical computations competing with critical computations within the application and the interference of the non-critical computations with each other. We ran Scenario3 under two experimental conditions - without any competing load and with a competing load of two kernel compilations, one on the local partition, while the other was on a NFS mounted partition. Table II summarizes the results of our evaluation and shows the CPU utilization of the individual computation components in our application.

1) Effect of background load: The results in Table-II show that the competing load did not affect our experiments which ran exactly as though there was no other load on the machine. Inspite of background load, the CPU utilization of the individual pipelines dont vary between the two experiments or between the middleware and kernel implementations. This shows that both the kernel and middleware schedulers are robust with respect to resource partitioning in the face of background system load.

2) Middleware scheduler overhead: The middleware scheduler incurs the overhead of maintaining two tasks to control scheduling. The overhead in terms of context switches is notable in the case of the middleware implementation. With the middleware scheduler, the number of context switches raised from around 2277 in the kernel experiment to 16846 for the middleware experiment with no background load (See Table-II). The overhead in terms of context switches is notable in the case of the middleware implementation. This is because of the extra context switches involved in switching to the scheduling task and the block catcher tasks. For example, consider the situation where the middleware scheduler had picked thread A to run. At the end of a quantum, there is a context switch to the scheduling task. Even if the scheduling task picks the same thread A again, there is an extra context switch back to A. This context switch will not be seen in case of the kernel group scheduling implementation.

TABLE II
MIDDLEWARE OVERHEAD AND EFFECT OF BACKGROUND LOAD

	Middleware		Kernel	
	1	2	1	2
Experiment Duration(ms)	78452	78489	74755	74756
CPU % (Application)	79.86	79.83	77.28	77.32
CPU % (Total)	82.17	79.88	81.39	77.34
RTCORBA Threads(%)	0.267	0.274	0.244	0.263
Pipeline1(%)	7.768	7.765	7.621	7.620
Pipeline2(%)	14.979	14.980	14.613	14.616
Pipeline3(%)	8.383	8.383	8.222	8.222
Pipeline4(%)	3.478	3.479	3.440	3.442
Pipeline5(%)	15.774	15.773	15.381	15.381
Pipeline6(%)	16.251	16.234	15.736	15.735
Pipeline7(%)	12.009	12.010	11.731	11.734
Stimuli source(%)	0.291	0.294	0.288	0.309
Scheduling task(%)	0.594	0.569	NA	NA
Block Catcher(%)	0.068	0.067	NA	NA
Idle task(%)	17.673	20.075	18.306	22.52
Total Context Switches(CS)	30056	16846	47924	2277
CS to scheduler	18.255	32.761	NA	NA
CS to block-catcher	3.782	6.826	NA	NA

1 - With 2 kernel compilations, 2 - With no competing load
* - Pipeline tasks, RTCORBA threads, Stimuli Threads (for kernel)
* - Pipeline tasks, RTCORBA threads, Stimuli Threads, Group Scheduling Task, Block Catcher (for middleware)

The group scheduler overhead itself was also found to be very low. It is around 0.60% of the total CPU utilization for the experiment in middleware, while in the kernel group scheduling version it was 0.01%.

3) Balanced Progress of Non-critical streams: We now proceed to analyze the interference among the non-critical pipelines. To achieve this, we modified our experiment parameters in such a way that the pipelines are kept busy by adjusting the send rate of the stimulus and the computation times of the messages. Pipelines 3 and 5 were chosen to have relatively lower message computation times when compared to other pipelines.

When the frame processing time varies for different frame types as shown in Figure 7, it becomes difficult to express policy (4) in terms of existing low-level scheduling policies. We attempt to do such a mapping in Scenario1 and Scenario2. We now demonstrate that this approach fails and hence we need to be able to specify the application policy directly as in Scenario3.

We expect a balanced progress for each of the non-critical streams in terms of the number of frames processed over a period of time. We plotted the number of frames processed by each non-critical pipeline against the total number of frames processed by all the non-critical pipelines. Plots for the three scenarios are shown in Figures 8-13.

Figures 8 and 9 show the frame progress for each non-critical pipeline under Scenario1 with kernel and middleware implementations respectively of the group scheduler. Here only the critical stream is under group scheduling control and all the non-critical streams are under the control of the Linux scheduler. Even though none of the pipelines undergo starva-

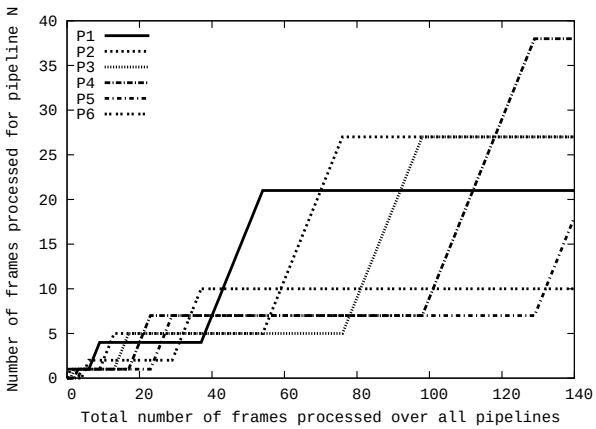

Fig. 8. Frame Progress with kernel Group Scheduling- Scenario1

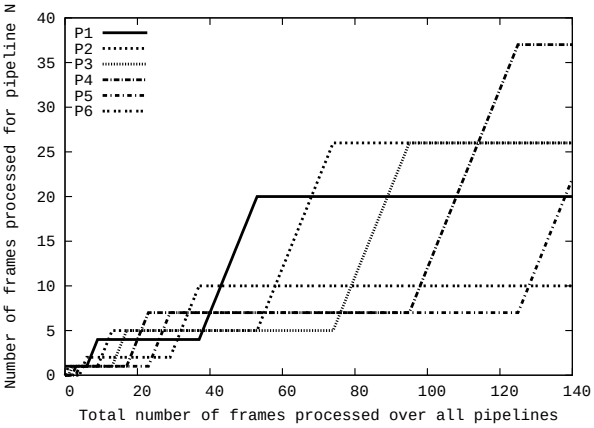

Fig. 9. Frame Progress with middleware Group Scheduling- Scenario1

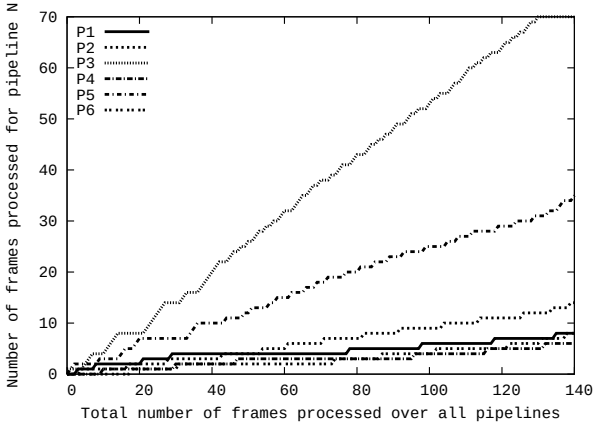

Fig. 10. Frame Progress with kernel Group Scheduling- Scenario2

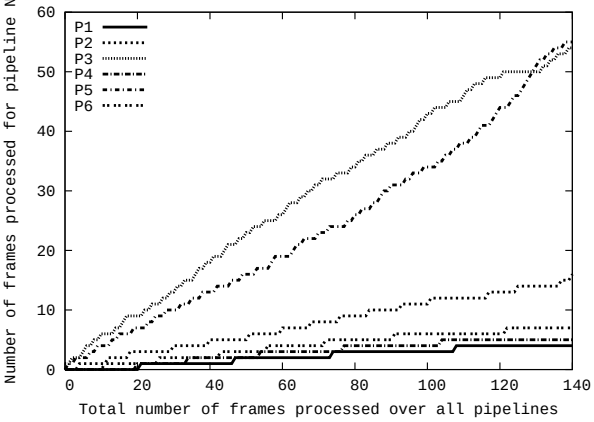

Fig. 11. Frame Progress with middleware Group Scheduling- Scenario2

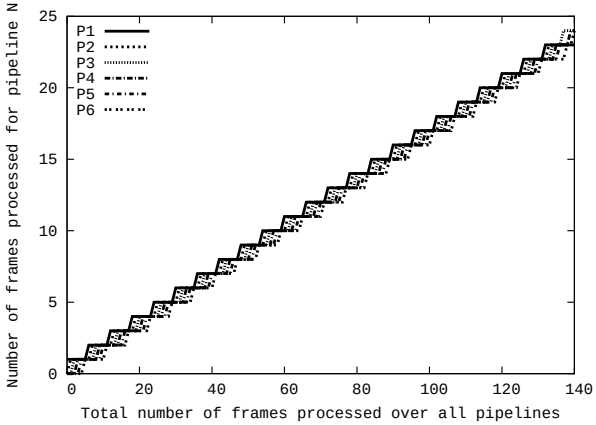

Fig. 12. Frame Progress with kernel Group Scheduling- Scenario3

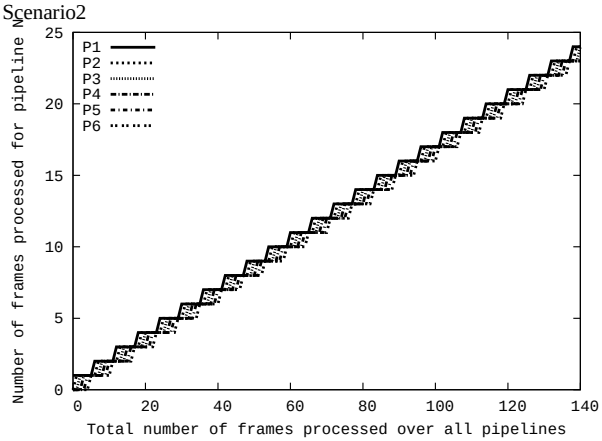

Fig. 13. Frame Progress with middleware Group Scheduling- Scenario3

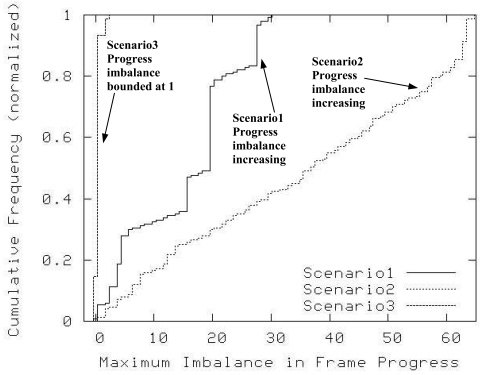

Fig. 14. Difference in progress across pipelines - Kernel

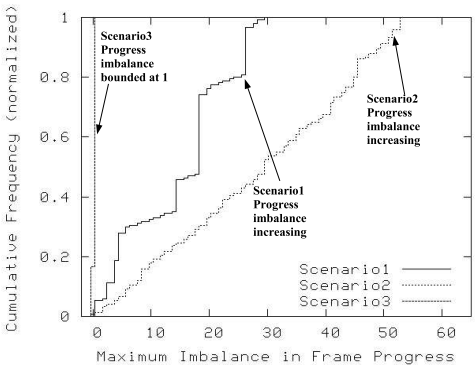

Fig. 15. Difference in progress across pipelines - Middleware

tion, there is a lot of jitter in the frame progress which violates application policy (4). The middleware and kernel based group scheduling implementations show a similar behavior, although it should be noted that Scenario1 is not affected by the group scheduler implementation as far as the non-critical streams as concerned, since they are under the control of the Linux scheduler rather than the group scheduler and hence have no influence over the scheduling decisions of the Linux scheduler.

Figures 10 and 11 show the frame progress for each non-critical pipeline under Scenario2. Here all the non-critical streams are under the control of the group scheduler. A round-robin policy with a fixed time quantum was chosen for scheduling among the non-critical pipelines. As seen in Figure 7, we have setup our experiment in such a way that the frames passing through pipelines 1 and 3 cause lesser computation times on the computation nodes on these pipelines, when compared to the other pipelines. The result is that these two pipelines get to process more frames in a specified period of time due to the round-robin policy and the lower computation times. Clearly this is a violation of application policy (4).

Figures 12 and 13 show the frame progress for each non-critical pipeline under Scenario3. Here all the non-critical pipelines are under the control of the group scheduler and an application-specific scheduling policy is used. The application specific scheduling policy uses the number of frames processed by a pipeline as an input for making a scheduling decision. The graphs show that the application policy of balanced progress is satisfied very well. A feedback of the number of frames processed so far by a pipeline is sent to the group scheduler on the completion of processing of each frame on that pipeline.

The variable execution time for a message in the different stages of the pipeline make it difficult to chose apriori a scheduling policy that will let us maintain the balance. Our frame progress based scheduler maintains balance by ensuring that no non-critical pipeline is more than one stimuli ahead than the other non-critical streams.

The cumulative distributions in Figures 14 and 15 illustrates this balanced frame progress for Scenario3 and imbalance in frame progress for the other scenarios. We calculated the maximum frame progress imbalance - the difference between the number of frames processed by the pipeline that finished processing the maximum number of frames, and the pipeline that finished processing the minimum number of frames. Moreover, in Scenario3 this balance is maintained over time, whereas for the other scenarios the imbalance increases over time. The maximum imbalance for Scenario3 is 1. There is one instance of an imbalance of 2 for Scenario3 at the top of the curve. We verified this to be an experiment termination condition where one thread processed one more frame when all the other pipeline threads were shutting down.

4) End-to-End Response time: We also measured the end-to-end response time for processing a stimuli. The end-to-end response time is the elapsed time between the stimulus source generating a stimuli to be sent to a pipeline, and the time when the stimuli has been completely processed by the last-stage of the pipeline.

TABLE III
END-TO-END RESPONSE TIMES

(in msec)	Kernel		Middleware	
Scenario	Minimum	Maximum	Minimum	Maximum
1-C	354	363	356	365
2-C	357	368	359	368
3-C	359	368	361	370
1-NC	42707	78257	42983	78580
2-NC	3767	78382	343	75330
3-NC	626	49931	2510	49102

Table III gives the minimum and maximum end-to-end response times for the critical stream(C) and all the non-critical streams(NC). The end-to-end response times for the critical computation is consistent across all scenarios. The variations in end-to-end response times among non-critical streams across scenarios is due to varying scheduling policies used in choosing among the non-critical computation groups. The difference between the minimum and maximum values is due to varying message processing times among the non-critical streams, scheduling policy and critical stream processing.

VI. RELATED WORK

Related work can be organized into several categories if we consider the dominant decomposition [19] as a classification criterion. The obvious first category are those approaches that adapt and wrap the common priority based scheduling scheme, without introducing any fundamental change. Their dominant decomposition concentrates on the semantics of the scheduling algorithm, layering additional semantics on top of the basic priority scheme in various ways. For example, rate monotonic analysis and scheduling [11] assumes straightforward priority scheduling, but provides an analysis method which maps real-time constraints onto computation priorities. Similarly, earliest deadline first [11] takes the basic real-time scheduling criterion of deadline and uses it as the basis for a priority driven choice. Maximum urgency first [12] and least laxity [12] schemes use a similar technique, but vary the method by which the numeric basis of the priority evaluation is calculated.

In Kokyu [7], the dominant decomposition is slightly different, concentrating on the mechanisms for re-ordering the queues of schedulable entities. That created a slightly richer approach to describing some aspects of system scheduling semantics, but the fundamental assumption of priority scheduling semantics remained unchanged. A slightly higher level abstraction using essentially the same dominant decomposition is the CORBA based resource broker service [4]. The higher level abstraction of the resource broker includes the ability to consider a range of system state information and to use either priority or share based underlying scheduling semantics, but the approach still assumes a static and uniform underlying scheduling semantics from the endsystem.

Other approaches do adopt a different decomposition for the scheduling semantics. For example, the Scout operating system [13] concentrates on execution paths as the basis for scheduling computations. This clearly changes the view of a schedulable computation used by the scheduler, but it

continues the underlying assumption that all computations are scheduled using the same view. Similarly, in TAO's Dynamic Scheduling Real-Time CORBA 2.0 implementation [9], the view of a computation is changed to that of a distributable thread, but a single view is still assumed to be adequate for all computations, and the familiar priority scheduling model is assumed for endsystem decision making.

In contrast, familiar hierarchical scheduling frameworks [17], [16], [8] use the decision functions themselves as the dominant decomposition, but make no modifications to their view of the computations being scheduled as individual threads of execution on the endsystem. The BERT [2] scheduling algorithm slightly exceeded this characterization by applying a slack stealing scheduling algorithm to the path-oriented computation view of Scout.

The group scheduling approach described here and elsewhere [5], [6] emphasizes clarity of expression in an effort to provide the best possible support for application semantics. The group abstraction is sufficiently general that it can be used to implement the semantics of any of the previously cited scheduling approaches, as well as composite scheduling functions using more than one approach in different sections of the SSDF. For example, group scheduling can be used to describe a computation path by grouping the set of computation components within a group. The driving example described here did this by grouping components of a pipeline together. The group scheduling model can also be used to adopt the decomposition used by hierarchical scheduling. The grouping of the non-critical computations into a group controlled by the RR or frame-based scheduler is an example of this.

VII. Conclusions and Future Work

The group scheduling approach described and evaluated here is an extremely flexible framework within which a wide range of scheduling semantics can be expressed. This expressive range includes that of several existing scheduling approaches normally considered disjoint, and permits the use of different approaches within different portions of the composite system scheduling decision function. Further, we have shown that both an OS based and middleware based implementations of the framework exist which support the same semantics, with few limitations. The middleware implementation does incur an unavoidable increase in context switching overhead, but this is also essentially the minimum overhead possible given the need for a user-level scheduling thread. Further, the middleware implementation is not able to include interrupt handler or other OS computation components under its control as the OS implementation does, but this is also to be expected. The greatest limitation of the middleware implementation is the need for a mechanism by which the scheduling thread can be notified when a controlled thread becomes unblocked. We demonstrated a mechanism requiring a wrapper for each potentially blocking system call. Alternately, some form of OS support similar to scheduler activations [1] can be used.

Our future work includes investigation of how group scheduling may be used for a variety of applications, including: time division use of Ethernet to implement reliable real-time QoS, extension of the group scheduling approach to management of distributed computations, and its use for supporting life science laboratory experiments requiring real-time QoS.

References

[1] Anderson, Bershad, Lazowska, and Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992.

[2] Bavier, Peterson, and Mosberger. BERT: A Scheduler for Best Effort and Realtime Tasks. Technical Report TR-602-99, Princeton University, 1999.

[3] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[4] et. al. D. Fleeman. Quality-based adaptive resource management architecture (qarma): A corba resource management service. In *12th International Workshop on Parallel and Distributed Real-Time Systems*, Santa Fe, NM, apr 2004.

[5] M. Frisbie. A unifi ed scheduling model for precise computation control. Master's thesis, University of Kansas, June 2004.

[6] M. Frisbie, D. Niehaus, V. Subramonian, and Christopher Gill. Group scheduling in systems software. In *Workshop on Parallel and Distributed Real-Time Systems*, Santa Fe, NM, apr 2004.

[7] Gill, Schmidt, and Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), January 2003.

[8] Goyal, Guo, and Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In 2^{nd} *Symposium on Operating Systems Design and Implementation*. USENIX, October 1996.

[9] Krishnamurthy, Pyarali, Gill, and Wolfe. Design and Implementation of the Dynamic Scheduling Real-Time CORBA 2.0 Specifi cation in TAO. In *OMG Workshop on Real-time and Embedded Distributed Object Computing*, Alexandria, VA., July 2003. OMG.

[10] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Proceedings of the* 2^{nd} *Annual Conference on the Pattern Languages of Programs*, pages 1–7, Monticello, Illinois, September 1995.

[11] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, January 1973.

[12] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, New Jersey, 2000.

[13] Mosberger and Peterson. Making Paths Explicit in the Scout Operating System. In 1^{st} *Symposium on Operating Systems Design and Implementation*. USENIX Association, October 1996.

[14] D. Niehaus. Improving support for multimedia system experimentation and deployment. In *Workshop on Parallel and Distributed Real-Time Systems*, San Juan, Puerto Rico, April 1999. Also appears in Springer Lecture Notes in Computer Science 1586, Parallel and Distributed Processing, ISBN 3–540–65831–9, pp 454–465.

[15] Irfan Pyarali, Marina Spivak, Ron K. Cytron, and Douglas C. Schmidt. Optimizing Threadpool Strategies for Real-Time CORBA. In *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, pages 214–222, Snowbird, Utah, June 2001. ACM SIGPLAN.

[16] Regehr, Reid, Webb, Parker, and Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In 24^{th} *IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

[17] Regehr and Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. In 22^{nd} *IEEE Real-Time Systems Symposium*, London, UK, December 2001.

[18] David B. Stewart and Pradeep K. Khosla. Real-Time Scheduling of Sensor-Based Control Systems. In W. Halang and K. Ramamritham, editors, *Real-Time Programming*. Pergamon Press, Tarrytown, NY, 1992.

[19] Tarr, Harrison, Ossher, Finkelstein, Nuseibeh, and Perry. Mdscse workshop description. In *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, Limerick, Ireland, June 2000.