Design and Performance of Configurable Endsystem Scheduling Mechanisms *

Tejasvi Aswathanarayana and Douglas Niehaus
{tejasvi,niehaus}@ittc.ku.edu
Computer Science and Engineering
University of Kansas, Lawrence, KS

Venkita Subramonian and Christopher Gill
{venkita,cdgill}@cse.wustl.edu
Computer Science and Engineering
Washington University, St. Louis, MO

Abstract

This paper describes a scheduling abstraction, called group scheduling, that emphasizes fine grain configurability of system scheduling semantics. The group scheduling approach described and evaluated in this paper provides an extremely flexible framework within which a wide range of scheduling semantics can be expressed, including familiar priority and deadline based algorithms. The paper describes both OS and middleware based implementations of the framework, and shows through evaluation that they can produce the same behavior for a non-trivial set of application computations. We also show that the framework can support application-specific scheduling constraints such as progress, to improve performance of applications whose scheduling semantics do not match those of traditional scheduling algorithms.

1. Introduction

Distributed real-time and embedded (DRE) systems are increasingly common across a wide range of application domains. Constraints on application 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 several application domains we have examined recently, including industrial automation, military command and control, and life science laboratory experiment control/management.

A key challenge these systems pose for application designers is the increasing complexity of the application semantics and the resulting difficulty of expressing those semantics in terms of commonly available programming models. The scheduling model in particular provides a prominent example of this difficulty. The challenge addressed by this paper, which is faced by both application and system designers, is that no single scheduling model is adequate for expressing the full range of scheduling semantics for all 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 generally assume that the underlying OS provides a single scheduling model; typically 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 [12], earliest deadline [12], maximum urgency [19], and least laxity [13, 19] schemes.

The flexibility of priority scheduling makes it suitable for the design of many DRE systems. However, endsystem designers have continued to provide this priority scheduling model even when it places an undue burden on the application designers to map application semantics directly into priority assignments. For example, Linux by default provides a dynamic priority scheme fairly typical of general purpose systems. 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 endsystem scheduling model is appropriate for a fairly wide range of applications, priorities are not adequate to express important classes of DRE application semantics such as coordinated progress of multiple independently time-varying computations.

Endsystem scheduling frameworks have historically increased their flexibility by increasing the configurability of low level resource control mechanisms, rather than focus-

* This work was supported in part by DARPA PCES contract F33615-03-C-4111 and by NSF EHS contract CCR-0311599.

IEEE COMPUTER SOCIETY

ing on direct support for the application level resource control requirements. However, this places an undue burden on application developers who are then made responsible for expressing the application-level resource requirements in terms of low level mechanisms whose semantics may differ considerably from those of the application.

In this paper we focus on 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" (GS) because it emphasizes representation of the groups of computation components comprising an application. Furthermore, it recognizes the diversity of scheduling semantics by permitting each group to use the scheduling algorithm most appropriate to the set of components it controls. The DRE system designer then constructs a scheduler for the system as a whole through hierarchic composition of groups.

Group Scheduling thus provides an extremely flexible model that can be used to express a 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 application in operational terms. Also, 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 systems research issues, including: (1) the fidelity with which application resource requirements can be expressed and enforced, (2) the portability of the framework *and its enforcement capabilities* 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 within which our group scheduling framework will be integrated [3].

The rest of this paper first discusses the motivations and challenges of our work in Section 2, and the essential aspects of the group scheduling framework in Section 3. We then describe details of our middleware group scheduling implementation in Section 4. Section 5 presents experimental results demonstrating the validity of the claims we have made about our approach. We then discuss related work in Section 6, and Section 7 presents our conclusions and discusses future work.

2. Motivation and Challenges

Many DRE systems involve a non-trivial number of computations whose activities must be coordinated. One common scenario is sets of computations operating on one or more streams of data. Each computation can often be viewed as a *pipeline* of computation components. Examples of such systems to which the work described here is relevant include: multi-channel sensor fusion for DRE systems such as sensor nets or laboratory science, multi-channel audio data mixing, or time-critical military applications. These and similar classes of applications have a number of common characteristics that we have abstracted into a representative application architecture, which we used in the design and evaluation of our OS and middleware group scheduling implementations.

In the rest of this paper we focus on an example, drawn from a canonical video processing challenge problem, which exhibits characteristics emphasizing the need for flexible scheduling support for many DRE application areas:

1. Each computation is implemented as a pipeline of computation components operating on a stream of data elements. 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 is true of many applications, but is particularly characteristic of video processing applications.
4. Multiple pipeline computations are supported by the DRE system as a whole. This is an important property for our experimental evaluations because it makes the scheduling problem both realistic and sufficiently complex to show that priority driven and other popular scheduling algorithms have important limitations.
5. Balanced progress by multiple computations can be an important application requirement. This is true of many applications which fuse data from several sources, but is particularly relevant to supporting multiple video streams that are meant to be viewed concurrently.

Three kinds of resource contention must be resolved by the scheduling policies on each 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 allows DRE system

IEEE COMPUTER SOCIETY

designers to address these types of conflicts using separate group representations, and provide an individual scheduling policy for each group.

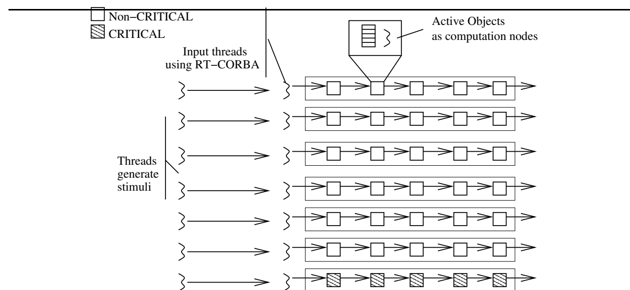


Figure 1. Computation pipelines

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 an unmanned aerial vehicle to a shipboard computer. If a particular pipeline is identified as containing a sensitive image like an enemy battle-tank, that pipeline is designated as *critical*.

There are several interesting performance metrics for this application. First, streams are divided into critical and non-critical classes, with execution of critical class members taking strict precedence over that of non-critical streams. Second, the processing of streams within each class must be *balanced* in terms of their progress on a stream of frames. This is important for a number of sensor fusion situations, such as when several streams of video will be viewed together. Third, we are interested in the overall frame throughput of the application. Finally, we are also interested in how well the scheduling framework can partition the CPU resource used by the pipeline computations from that of other computations on the system.

Each pipeline is implemented as a set of active objects [11] under the ACE ORB (TAO)[4]. Each active object provides 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 thread executed periodically on another machine, which is not a member of the set of computations under group scheduling control. It is important to note that a thread within the TAO middleware receives the stream of frames, 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 thread-pools [16], 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. Thus, a computation stream contains a message receiver RTCORBA thread and a set of pipeline threads that process each frame. Each frame is received by the computation stream's RTCORBA thread which waits on a socket that is used by 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 permits TAO to start the execution of the active objects that will pass the frame through the pipeline.

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

3. 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 [18, 17, 9]. What distinguishes the group scheduling framework is (1) an emphasis on grouping of computation components to represent application computations, (2) the association of an arbitrary scheduling decision function with each group to produce an extremely flexible scheduling framework capable of clearly and easily expressing a wide range of scheduling semantics, and (3) efficient OS and middleware implementations of the framework in the popular open source Linux platform.

3.1. Group Scheduling Implementation Core

A group is defined as a collection of computation components with an associated scheduling decision function (SDF) that selects from 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 explicitly under SSDF control because both the OS and middleware group scheduling implementations permit

IEEE COMPUTER SOCIETY

the default OS scheduler to make a decision if the SSDF does not make a choice. Computations can be placed under exclusive control of the SSDF or under joint control of the SSDF and the default OS scheduler.

The group scheduling framework emphasizes modularity of the SDF implementations and thus makes it relatively easy for users to implement their own SDFs if a library of available functions does not include one matching the scheduling semantics they desire. The group scheduling framework can subsume many of the popular scheduling models by providing matching SDFs (e.g. RMS or EDF). However, it can also support application specific 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 5.

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 in the remainder of this section. For example, the application based specialization of the frame-progress scheduler depends on information from the application about progress supplied to the SSDF in a variety of ways. The OS level group scheduling implementation has direct access to much of the progress information directly, while the middleware level group scheduling implementation requires additional mechanisms to transmit progress information from the application to the scheduler.

3.2. OS Level Group Scheduling

The OS level implementation of group scheduling enjoys several advantages over the middleware version. First, it offers direct integrated control over *all* computation components in the Linux system, including hardware interrupt handlers, soft interrupt (Soft-IRQ) handlers, tasklets and bottom halves [7]. In contrast, the middleware implementation can only control threads. Second, the overhead of performing context switches is lower in the OS implementation because the scheduling decision made by the SSDF and the actual context switch both occur in the OS context. The middleware version must use an indirect mechanism based on 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, again 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, for example, integrated group scheduling control of interrupt handlers and network protocol processing, have semantics that can only be implemented under the OS group scheduling implementation. One disadvantage of the OS group scheduling implementation is that it requires access to the source of the OS, and involves subtle modifications to how the OS manages computation components to produce a unified scheduling framework. To address these issues, we have also implemented our group scheduling framework at the middleware level, as we describe next.

3.3. 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* thread which helps detect state changes of the controlled threads. In contrast, these mechanisms are implicit parts of the OS level group scheduling implementation. The API which application code can use to provide scheduler-specific parameters and to construct the SSDF are based on shared-memory in the middleware implementation, as opposed to *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 4, to track computation state changes. As shown in Section 5, the behavior of the application controlled by the middleware scheduler closely matches that of the application under OS scheduler control. The middleware version pays a price in context switch overhead but enjoys the advantage of greater portability since it only depends on commonly available OS capabilities.

4. Middleware Implementation

This section describes details of the middleware level group scheduling implementation, which are relevant for understanding our experimental results for the example application described in Section 5. In the following discussion, we refer to threads under the control of the middleware group scheduler as *controlled* threads. We used the native OS priority model as an enforcement mechanism for the decisions made by the SSDF. To achieve this, we used five different priority levels as shown in Table 1. `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 1, for its own functioning. The Reaper thread helps in shutting down and gaining control of the system in case of a faulty group scheduling hierarchy. Scheduling is done in the context of the System Scheduler Decision Thread (SSDT). The Block Catcher thread helps detect blocking of a controlled task by running

IEEE COMPUTER SOCIETY

immediately after a controlled task blocks. Requests to the group scheduler are handled by an API thread to decouple scheduling data update and scheduling decision execution.

Reaper thread	MAX_PRIO
Blocked Controlled threads	MAX_PRIO-1
GS 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

Table 1. Middleware GS priority levels

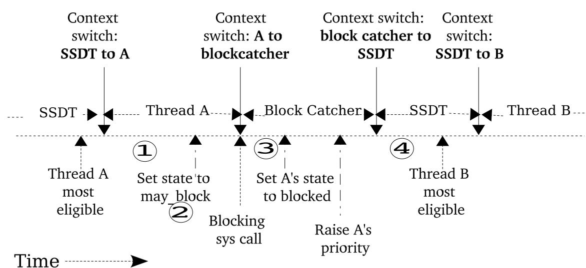


Figure 2. A controlled task blocks

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 localized and hence this approach is highly scalable. All system calls that may block are wrapped as follows.

```
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, changes the status of the blocked thread to BLOCKED (3). and increases the priority of the blocked thread to MAX_PRIO-1 as shown in Table 1. The block catcher thread then wakes up the scheduler thread (4), and the scheduler thread calls the SSDF which picks up the most eligible thread to run.

After a (possibly blocking) system call returns, its controlled thread calls the *after_system_call_hook* function (5).

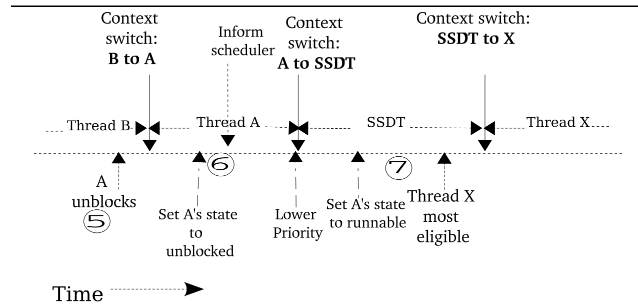


Figure 3. A controlled task unblocks

This function ascertains whether or not the thread actually blocked on the system call. If the thread had not blocked on the system call it simply changes its status to RUNNABLE and continues to run. If it did block, it changes its status from BLOCKED to UNBLOCKED, and awakens the scheduler thread (6) to make a new scheduling decision. The scheduler then changes the status of the (possibly multiple) UNBLOCKED threads to RUNNABLE, so that they are all considered for scheduling. The scheduler then again chooses the most eligible thread to run (7).

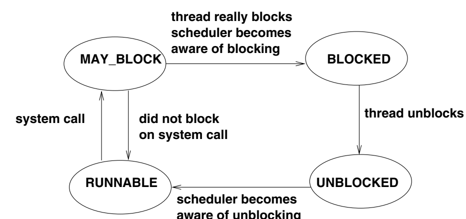


Figure 4. State transitions for threads

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 thus an eligible candidate that could be picked to run by the SSDF. The execution of these threads is controlled by sending SIGSTOP and SIGCONT signals. A 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 starts running again changing its state to UNBLOCKED. The scheduler is informed immediately about the unblocking and changes the controlled thread's state to RUNNABLE again.

5. Evaluation

We now proceed to evaluate our implementation described in Section 4 in the context of the challenges pro-

IEEE COMPUTER SOCIETY

vided by the motivating application described in Section 2. The goal of our evaluation is to demonstrate the following:

- the necessity and sufficiency of the group scheduling paradigm;
- that applications may require *simple* policies which are nonetheless difficult to express in terms of existing lower level scheduling policies;
- that group scheduling 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; and
- that efficient implementation of group scheduling middleware is possible, making our solution directly applicable to a wide variety of platforms and operating systems.

For this evaluation, we used a random payload to represent an image frame that is being sent by an image source. We state a simple but realistic set of application policy goals (APG) for our example application:

1. preference of critical streams over non-critical streams;
2. a computation pipeline is drained as fast as possible;
3. receiving an input frame has preference over frame processing in a pipeline; and
4. balanced progress is desired, in terms of the number of frames processed, among the non-critical streams in the face of variable computation times for frames passing through the pipeline.

5.1. Qualitative Evaluation

Expressing the above high-level application policies using existing low-level scheduling policies and priority assignments can be tedious and error-prone. Moreover, static policies 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 is a generalization of conventional scheduling paradigms and is 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 SDFs, 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 classes of applications whose desired scheduling semantics do not map reliably onto conventional scheduling semantics.

We now demonstrate the *sufficiency* of group scheduling to express our application-specific policies using the computational model described in Section 3. In the course of doing this, we show the inadequacy of existing low-level policies to capture APG-4 which then demonstrates the *necessity* of group scheduling for many applications.

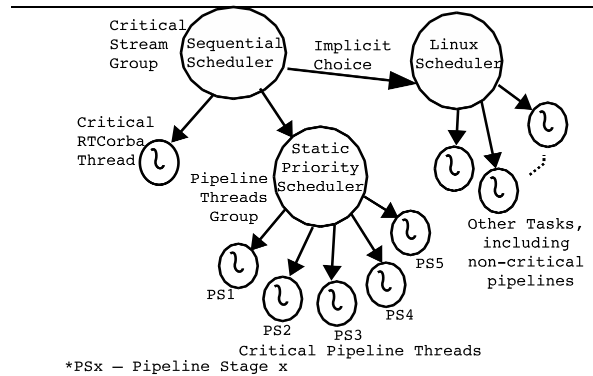


Figure 5. Group hierarchy for scenario 1

5.1.1. Scenario 1 – Non-critical Streams Outside GS Control: In this scenario, we try to map the four desired 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 APG-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 SSDF, shown in Figure 5 is used to specify this policy. A *Critical Stream* group is created to represent a critical stream. The SSDF is consulted before the default Linux scheduler whenever a scheduling decision is required. The default Linux scheduler is invoked only if no threads controlled by the SSDF are in the RUNNABLE state. This is indicated in Figure 5 by the arrow marked "Implicit choice".

To specify APG-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 APG-3. Hence a sequential scheduling policy is chosen for the *Critical Stream* group which checks to see if the RTCORBA thread can run before it checks the *Pipeline Threads* group.

To specify APG-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

IEEE COMPUTER SOCIETY

will be processed. For APG-4, in this scenario, we rely on the default policy of the default OS scheduler since all the non-critical stream threads are under its control.

5.1.2. Scenario 2 – Non-critical Streams Under RR SDF Control: This scenario attempts to address APG-4, which we left to the control of the default OS scheduler in Scenario 1. 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, with each stream supported by a *Pipeline Threads* group similar to that in Scenario 1. The critical stream is again given preference over this collection of non-critical streams. To express this policy, the non-critical stream groups are all grouped together under the *Non-critical Streams* group and this group gets lower preference than the *Critical Streams* group.

To address balanced frame progress (APG-4), we try to map this policy in terms of a round-robin scheduling policy for the *Non-critical streams* group. This would achieve balanced progress if the computation times were equal for each frame and each stream, but they are not. Note that Figure 6 shows the group hierarchy for both Scenarios 2 and 3. Scenario 2 uses the Round-Robin(RR) SDF for the *Non-critical streams* group, while Scenario 3 uses the Frame Progress SDF.

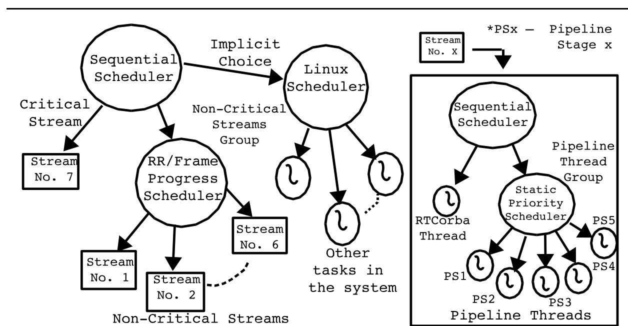


Figure 6. Group hierarchy for scenarios 2 and 3

5.1.3. Scenario 3 – Non-critical Streams Under Frame Progress Control: The requirement of balanced progress is very often a feature of the kinds of applications discussed in Section 2. Hence it is *necessary* for those applications to be able to specify scheduling policies in terms of higher level programming models within which alternative SDFs can be expressed.

If the computation times on a pipeline node can vary based on the type of frame that it is processing, then the RR approach used in Scenario 2 can lead to an imbalance in the number of frames that can be processed by different pipelines within the same time duration. Scenario 3 therefore uses a Frame Progress SDF for the *Non-critical streams* group. The group scheduling hierarchy for Scenario 3 is also shown in Figure 6, and is similar to the hierarchy for Scenario 2, except for the scheduling policy that is associated with the *Non-critical Streams* group.

The aim of our application-specific Frame Progress SDF is to maintain a balance of progress across multiple pipelines. In the Frame Progress SDF, a specified quantum of time is allotted for each non-critical stream, to balance the progress among the different streams. The Frame Progress SDF also ensures that no non-critical stream gets more than N frames ahead of the other non-critical streams, where N is an application defined constant (or in some cases could even be a function). This effect can be shown visually as described in Section 5.2.

5.2. Instrumentation, Logging and Visualization

Having demonstrated an intuitive style for specifying application policies in the group scheduling model, we now demonstrate that our group scheduling implementation indeed can enforce the application policies. We use a combination of visualization tools and quantitative analysis for this purpose.

We used the DataStream Kernel Interface (DSKI) and Datastream User Interface (DSUI) logging framework and post-processing tools to instrument our example application, the group scheduler, and the Linux kernel [15]. We used the visualizer tool to generate execution interval diagrams, like the one shown in Figure 7, to examine qualitatively how well our group scheduling implementations enforced the application specified policies discussed previously. For example, Figure 7 shows the visualizer tool displaying information mined via post-processing tools from the DSUI and DSKI event logs for an experimental run of Scenario 3 using the middleware group scheduler implementation.

The execution time-line depicted in Figure 7 clearly verifies the desired behavior. The visualizer has been configured to show the execution time-lines for all the pipelines. In our experiment, as seen in Figure 7, we chose messages with lower processing times 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 until it processes a frame completely, *i.e.* the frame passes through all stages of the pipeline, before any other pipeline starts processing its own frames. This is also confirmed by the visualizer, as none of the non-critical pipeline executions overlap. Arrival of a frame for the critical stream causes the threads related to the critical stream to run. As seen in the

IEEE COMPUTER SOCIETY

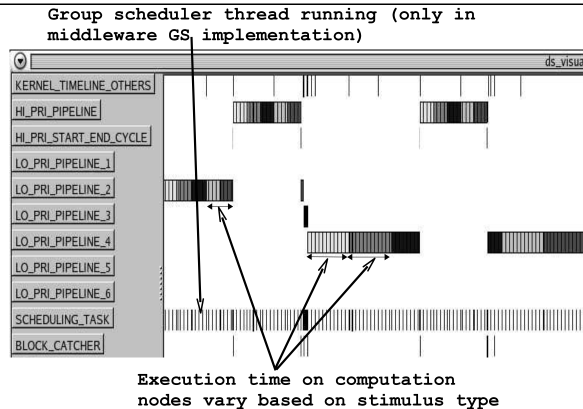


Figure 7. Scenario 3 execution (MW GS)

visualizer, Pipeline 1 was processing a frame when a frame for the critical stream arrived, as indicated by an event in the START-END cycle row of the visualizer. This event caused 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 shown by the visualizer. This is because the group scheduling task is run at least once every time quantum, which is set to 10 msec in our implementation, equal to the Linux system heartbeat frequency in the 2.4 kernel series. This is a tunable system parameter affecting context switching overhead in the middleware implementation.

5.3. Quantitative Evaluation

We also evaluated our kernel and middleware group scheduling implementations in the face of background load competing with the application, in the face of non-critical computations competing with critical computations, and the interference of the non-critical computations with each other. We ran Scenario 3 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 - and a CPU-intensive background task. Tables 2 and 3 summarize the results of our evaluation.

5.3.1. Effect of Background Load: The results in Table 2 show the CPU utilization of the different threads in the application. Each entry prefixed with "Pipe" in the table show the total utilization of all threads in that pipeline. The results of our experiments show that the competing load did

	Kernel		Middleware	
values in %	L	NL	L	NL
App	78.5	78.4	79.1	79.1
Total	99.0	78.5	98.9	79.1
RTCORBA	0.2	0.2	0.2	0.2
Pipe1	7.8	7.8	7.9	7.9
Pipe2	15.3	15.3	15.3	15.3
Pipe3	8.3	8.3	8.2	8.3
Pipe4	3.2	3.2	3.2	3.2
Pipe5	15.8	15.8	15.8	15.8
Pipe6	15.6	15.6	15.7	15.7
Pipe7	12.0	12.0	12.0	12.0
Scheduler	NA	NA	0.427	0.424
Block Catcher	NA	NA	0.063	0.065
CPU bound task	14.5	NA	14.7	NA
Frame source	0.3	0.3	0.3	0.3
Idle task	0.0	21.4	0.0	20.8

Table 2. CPU utilization with load (L) and with no load (NL)

not affect the application pipelines, which ran exactly as though there was no other load on the machine, with a consistent utilization of around 79% (entry "App"). With background load (L), the total utilization (entry "Total") was around 99% whereas with no load (NL) the total utilization was very close to the CPU utilization of the application. In spite of background load that causes a significant number of disk interrupts, network interrupts and intensive demand for CPU, the CPU utilization of the individual pipelines did not vary meaningfully between the two experiments or between the middleware and kernel implementations. This shows that both the kernel and middleware schedulers provide robust resource partitioning in the face of CPU, disk I/O, and network I/O bound background loads.

These results also demonstrate that our group scheduling implementation does not preclude the scheduling of computations that are not under the purview of group scheduling. For example, in Table 2, the CPU bound background task gets around 14% of the CPU, since the application utilizes only around 79%. There is also some minimal utilization (0.3%) by the client tasks (entry "Frame Source") that are responsible for periodically supplying frames to the computation streams. The remaining utilization can be attributed to the 2 background kernel compilations, which we have not shown here.

5.3.2. Middleware Scheduler Overhead: The middleware scheduler incurs overhead for maintaining two threads to control scheduling - the scheduler and block catcher threads. The overhead in terms of context switches is notable in the case of the middleware implementation.

IEEE COMPUTER SOCIETY

	Kernel		Middleware	
values in %	L	NL	L	NL
Scheduler	NA	NA	6.6	27.2
Block Catcher	NA	NA	2.2	9.1

Table 3. Context switch overhead

Table 3 shows the percentage of the total number of context switches during the experiment that are attributed to the scheduler and block catcher threads. This occurs because of the extra context switches involved in switching to the scheduler and block catcher threads, when a thread other than the currently running controlled thread becomes potentially eligible to run. Though the number of context switches is higher in the middleware implementation, the CPU time taken by these additional threads is insignificant, as shown by the utilization values in Table 2. For the kernel implementation, there are no additional context switches involved since the scheduling decision is made in the kernel itself. Moreover there is no need for the scheduler and block catcher threads in the kernel implementation and hence no data are shown in the tables for these entries.

5.3.3. Balanced Progress of Non-critical Streams: We now proceed to analyze the interference among the non-critical pipelines. To do this, we modified our experimental parameters so that the pipelines are kept busy by adjusting the send rate of the frame 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 goal APG-4 in terms of existing low-level scheduling policies, though we do attempt to do such mappings in Scenario 1 and Scenario 2. We now demonstrate that this approach fails, and hence we need to be able to specify the application policy directly as in Scenario 3.

We desire 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 8a-8f.

Figures 8a and 8b show the frame progress for each non-critical pipeline under Scenario 1 with kernel and middleware group scheduling implementations respectively. 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 experiences starvation, there is a lot of jitter and divergence in the streams' relative frame progress, which violates application policy goal APG-4. The middleware and kernel based group scheduling implementations show similar behavior, although it should be noted that Scenario 1 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 8c and 8d show the respective kernel and middleware scheduled frame progress for each non-critical pipeline under Scenario 2. 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 the non-critical pipelines. As seen in Figure 7, we set up our experiment so that the frames passing through pipelines 1 and 3 cause lesser computation times, 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 also a violation of application policy goal APG-4.

Figures 8e and 8f show the Scenario 3 non-critical pipeline frame progress. All non-critical pipelines are under control of an application-specific group scheduling SDF. Feedback data, the number of frames processed by a pipeline, is made available to the application-specific SDF as each pipeline completes processing of a frame. The application-specific SDF considers the number of frames processed by each pipeline when making a scheduling decision. The graphs show that the application policy of balanced progress is satisfied very well.

The variable execution times for a message in the different stages of the pipeline make it difficult to choose *a priori* a static scheduling policy that will let us maintain balanced progress. Our frame progress based scheduler maintained balance by ensuring that no non-critical pipeline is more than one frame ahead of the other non-critical streams.

The cumulative distributions in Figures 8g and 8h summarize the balanced frame progress for Scenario 3 and the imbalance in frame progress for the other scenarios. We calculated the maximum frame progress imbalance, i.e., the difference between the number of frames processed by the pipeline that has processed the most frames, and by the pipeline that has processed the least frames. Moreover, in Scenario 3 this balance is maintained over time, whereas for the other scenarios the imbalance increases over time. The maximum imbalance for Scenario 3 is 1. There is one instance of an imbalance of 2 for Scenario 3 at the top of the curve. We verified this to be an artifact of experiment termination where one thread processed one more frame when all the other pipeline threads were shutting down.

5.3.4. End-to-End Response Times: We also measured the end-to-end response time for processing each frame.

IEEE COMPUTER SOCIETY

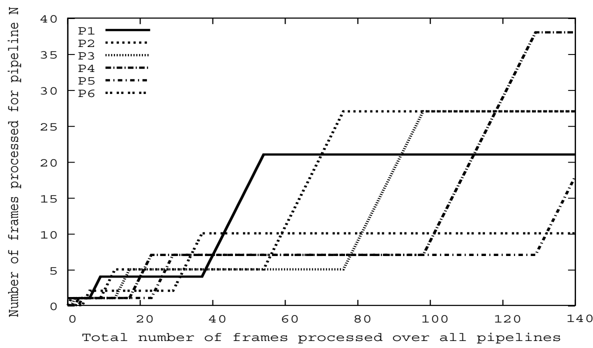


Figure 8a : Scenario 1 (Kernel)

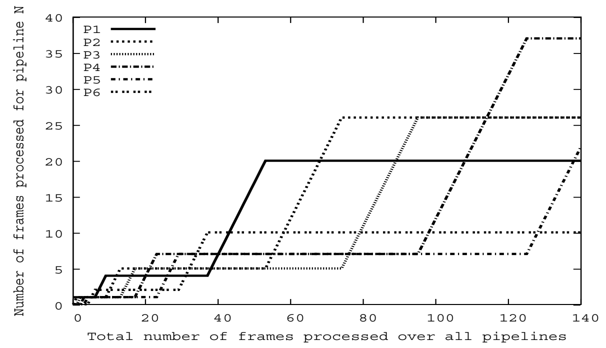


Figure 8b : Scenario 1 (Middleware)

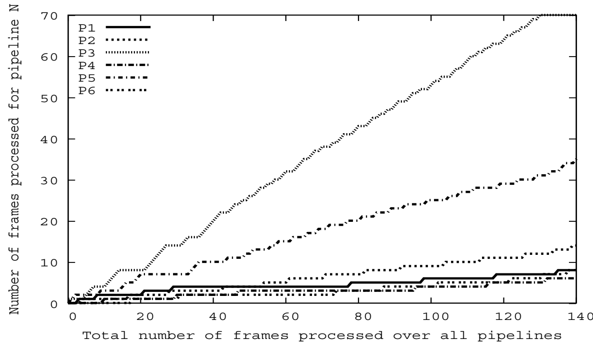


Figure 8c : Scenario 2 (Kernel)

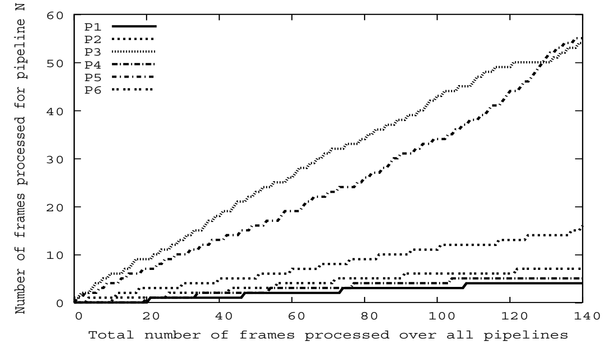


Figure 8d : Scenario 2 (Middleware)

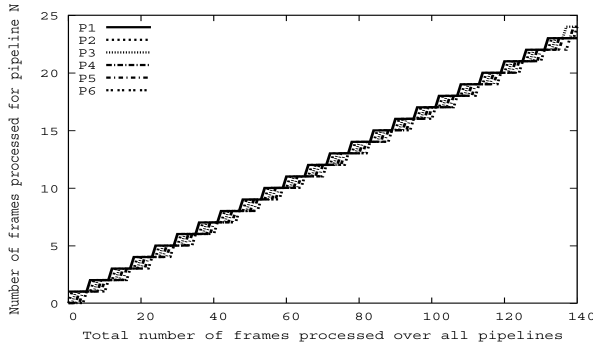


Figure 8e : Scenario 3 (Kernel)

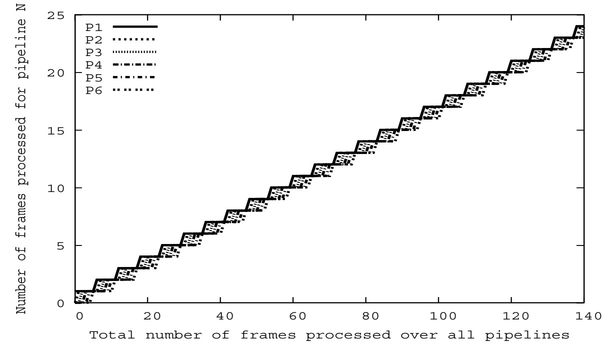
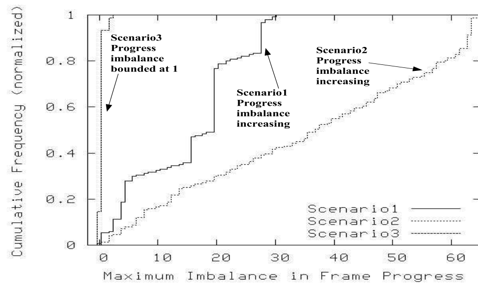


Figure 8f : Scenario 2 (Middleware)



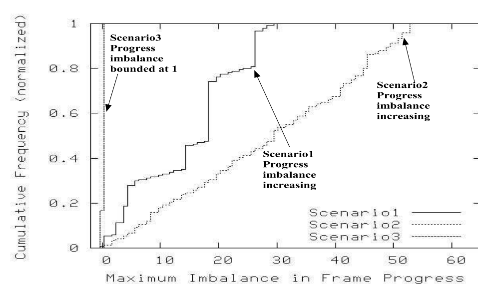


Figure 8g : Frame progress (Kernel)

Figure 8h : Frame progress (Middleware)

Figure 8. Frame progress of non-critical streams

IEEE COMPUTER SOCIETY

The end-to-end response time is the elapsed time between the frame source generating a frame to be sent to a pipeline, and the time when the frame has been completely processed by the last stage of the pipeline.

(in msec)	Kernel		Middleware	
Scenario	Min	Max	Min	Max
1	354	363	356	365
2	357	368	359	368
3	359	368	361	370

Table 4. End-to-end response times

Table 4 gives the minimum and maximum end-to-end response times for the critical stream for the different scenarios. The end-to-end response times for the critical computation is consistent across all scenarios and also between the kernel and middleware implementations. The processing times for each frame varies between the non-critical streams and hence the minimum and maximum end-to-end response times of non-critical streams is not a relevant metric and is not shown.

6. Related Work

Examples of related work can be organized into several categories by considering their dominant decomposition [20] as a classification criterion. The first category consists of approaches that adapt and wrap the common priority based scheduling scheme, without fundamentally changing that scheme. 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 [12] assumes straightforward priority scheduling, but provides an analysis method which maps real-time constraints onto computation priorities. Similarly, earliest deadline first [12] takes the basic real-time scheduling criterion of deadline and uses it as the basis for a priority driven choice. Maximum urgency first [13] and least laxity [13] schemes use a similar technique, but vary the method by which the numeric basis of the priority evaluation is calculated.

In Kokyu [8], the dominant decomposition is slightly different, concentrating on the mechanisms for re-ordering the queues of schedulable entities. That creates a slightly richer approach to describing some aspects of system scheduling semantics, particularly with respect to middleware enforcement mechanisms, 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 [5]. The higher level abstractions given by the resource broker include 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 adopt a different decomposition for the scheduling semantics. For example, the Scout operating system [14] 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 implementation of Dynamic Scheduling Real-Time CORBA 1.2 (previously designated Real-Time CORBA 2.0) [10], the view of a computation becomes 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 usually assumed for the endsystem OS scheduler.

In contrast, familiar hierarchical scheduling frameworks [18, 17, 9] 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.

Our previous work on the group scheduling approach [6, 7] 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.

In the research described in this paper, we have further shown the ability of the basic group scheduling approach to integrate both hierarchical and path-oriented scheduling semantics within a single scheduling model. For example, the group scheduling model used in Scenario 3 was able describe a computation path by grouping components of a pipeline together. The same group scheduling model also provided hierarchical separation of critical and non-critical stream scheduling, and integrated the semantics of multiple pipelines under the frame progress SDF.

7. Conclusions and Future Work

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. This expressive range includes many existing

scheduling approaches that are normally considered disjoint, and permits the use of different approaches within different portions of composite system scheduling decision functions. Moreover, we have shown that both OS based and middleware based implementations of the framework can 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. Furthermore, 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 in a wider range of contexts, including: time division multiplexing of Ethernet to implement reliable real-time communication, extensions of the group scheduling approach to manage distributed computations, and its use for supporting a wider range of applications requiring nuanced real-time QoS assurances.

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] Kevin Byran, David Fleeman, Matthew Murphy, Jiangyin Zhang, Lisa DiPippo, Victor Fay Wolfe, David Juedes, Chang Liu, , Lonnie Welch, Douglas Niehaus, and Christopher Gill. Integrated corba scheduling and resource management for distributed real-time embedded systems. In *11th IEEE Real-time Technology and Application Symposium (RTAS)*, San Fransisco,CA, March 2005. IEEE.

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

[5] D. Fleeman and et. al. 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.

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

IEEE COMPUTER SOCIETY