

Group Scheduling in Systems Software*

Michael Frisbie and Douglas Niehaus

{frizz,niehaus}@eecs.ku.edu

Electrical Engineering & Computer Science Department
Information and Telecommunication Technology Center
University of Kansas, Lawrence

Venkita Subramonian and Christopher Gill

{venkita,cdgill}@cse.wustl.edu

Department of Computer Science and Engineering
Center for Distributed Object Computing
Washington University, St.Louis

Abstract

Previous approaches to scheduling computer systems have focused primarily on system-level abstractions for the scheduling decision functions or for the mechanisms that are used to implement them. This paper introduces a new abstraction called group scheduling that focuses primarily on the progress of application-level computations and provides a flexible mapping into system-level scheduling abstractions so the progress of computations can be assured across a wide range of platforms and platform features.

This paper makes three main contributions to the state of the art in computer system scheduling. First, it defines a model for group scheduling that augments and complements the hierarchical scheduling model introduced by Regehr and Stankovic. Second, it describes how the semantics of a computation's progress can be mapped to several alternative scheduling mechanisms at the operating system and middleware levels. Third, it presents preliminary empirical studies of the performance of group scheduling in a realistic systems software and hardware environment.

1. Introduction

The purpose of a computer system scheduler is to organize how the set of computations sharing the computer system resources execute. The policy goals of the scheduler can vary with the nature of the system. General purpose systems emphasize computation progress and response time for interactive applications. Batch systems such as scientific simulations, data base servers, and data mining applications often emphasize throughput. Real-time systems

such as avionics mission computers, manufacturing machines, and search and rescue robots often emphasize timeliness of execution relative to advertised deadlines, and predictability of that execution under a range of operating conditions.

The mechanisms used to achieve scheduling policy goals typically vary less widely than the policies themselves, relying on commonly available services like POSIX thread priorities, in which an integer priority is associated with each thread of execution. The scheduling decision is thus reduced to simply selecting the thread with the best priority from among those ready to run. The core concept is appealingly simple: the integer priority values provide information which the scheduling decision function then uses to select the thread that should access the CPU at any given moment.

However, decades of practice have shown that the appealingly simple model is also inadequate for describing the full range of execution constraint semantics which computer system users need. One obvious problem is that several threads may have equal priority values, so a secondary scheduling decision function must be composed with the priority based one to determine how the CPU should be shared among threads of equal priority value. In some systems this additional decision function has no policy content, making it an essentially random choice. In other cases, such as for a multi-level priority queue in which a round-robin policy is used to share resources equally among threads with equal priority, policy content is more evenly divided among the components of the system scheduling decision function (SDF).

In other systems scheduling policy components are combined not through composition to form a single integrated system SDF, but by dynamic manipulation of the thread priority value under a priority based SDF. One common example is that one policy of a system may be that even the lowest priority threads

* This work was supported in part by the DARPA PCES program (contract F33615-03-C-4111).

should “make progress” at some minimal rate. A typical mechanism used to implement this policy is “aging” of the thread priority, where the dynamic priority steadily improves as a thread remains ready to run, but is not granted use of the CPU.

Scheduling has been the focus of an impressive amount of both theoretical and implementation effort over many years, producing a wide range of different SDFs and different ways to use them to implement an equally wide range of scheduling policies. The majority of both theoretical and implementation efforts have, however, been *thread centric* in that they have concentrated on scheduling decision functions that view the computational load on that system as a set of individual threads. Furthermore, the vast majority of system platforms available for both research and application implementation have selected particular SDF semantics, without allowing system designers to influence how scheduling decisions affecting their applications are made, apart from specifying parameters used by the underlying system SDF.

This paper presents an approach to system SDF design and usage that addresses the limitations on system designers that we believe are imposed by these typically thread centric and monolithic system SDFs. We call our approach *group scheduling* because it explicitly addresses several important trends in system design: computations are implemented as sets of threads, and the most appropriate scheduling semantics for those threads depends on the semantics of the computation. Group scheduling addresses these issues by permitting threads to be members of groups representing computations, and by permitting each group to have an associated SDF. The system SDF is formed by hierarchically composing groups SDFs into a decision tree that controls the execution of all threads on the system.

This approach provides explicit support for a number of important design and implementation issues. First, it explicitly represents *computational structure*, so that the progress of computation execution can be considered in scheduling decisions. Second, it significantly increases the customizability of the system because it permits developers to select, or to implement, the SDF that is most appropriate for each group representing an application computation. Third, but more subtly, it increases the *transparency* of the system because execution logs from the system SDF can provide an understandable narrative for each scheduling decision. Finally, it is important to note that the group scheduling approach to implementing SDFs can be applied to systems using abstractions other

than threads for elements of a computation’s execution environment, including: events, distributable threads, and CORBA GIOP messages. Thus, while we have begun by implementing group scheduling for threads and events at the system and middleware levels, we foresee a much more diverse range of potential implementations.

This paper is structured as follows. Section 2 discusses examples of several categories of related work. Section 3 describes the group scheduling model and gives examples of how it can be used to implement both familiar general purpose scheduling policies and customized application-specific computation control. Section 4 describes how we have implemented the group scheduling model in the KURT-Linux kernel, along with several approaches we have taken to implementing it in middleware. Section 5 presents results from several experiments with both the KURT-Linux implementation and the most similar of the middleware implementations to the one in KURT-Linux. Finally, Section 6 offers conclusions and discusses future work.

2. Related Work

Three main abstractions make up the group scheduling model presented in Section 3: the semantics for progress of each *computation*, the *execution environment* within which the computation is run, and the *scheduling decision function* used to schedule access to resources within each execution environment. Two areas of previous work are most relevant to group scheduling, and are the focus of this section: (1) making the execution environment flexible so that alternative SDFs can be expressed in it, and (2) composing SDFs for analytical assurance of real-time performance across multiple interacting execution environments. The main distinction between the work presented in this paper and the related work is that our work on group scheduling offers a *general* model for mapping computation semantics *explicitly* into the execution environments and SDFs, as a third dimension for design, measurement, and optimization.

A key theme in comparing our group scheduling abstraction to other related work is that of the *dominant decomposition* [15] used in each approach. In Kokyu, the dominant decomposition was along the mechanisms for re-ordering the execution of schedulable entities. In the Scout operating system and in TAO’s Dynamic Scheduling Real-Time CORBA 2.0 (DSRTC2) implementation, the dominant decomposition is still along the execution

environments, albeit those environments are customized for particular computation activities: *paths* in Scout, and *distributable threads* in DSRTC2. In the hierarchical scheduling framework and the BERT scheduling algorithm, the dominant decomposition is along the decision functions themselves.

With group scheduling, the dominant decomposition is along the semantics of progress for the computation itself, as represented by the group structure. In group scheduling, SDFs are composed along sets of paths constituting the execution environments within which parts of the computation are run. For example, in an audio-video application, distinct paths for the audio and video streams are defined, and the scheduling functions along those paths operate *in concert* to ensure timely delivery of both streams.

We view SDFs, execution environments, and scheduling mechanisms, as fundamental abstractions upon which our group scheduling abstraction builds. While functionally equivalent scheduling representations can be realized in each of these approaches in many cases, we view the dominant decomposition as a crucial issue for how these representations can be *programmed* in practice.

2.1. Execution Environments

The Kokyu scheduling framework [5] was designed to support flexible composition of customized middleware scheduling mechanisms for a wide range of SDFs. In particular, Kokyu offers a mapping from the ordering policy of a given SDF into thread, queue, and task parameter configurations to enforce that policy. Kokyu was originally implemented within TAO's real-time event channel. We have since factored out the Kokyu implementation as a stand-alone framework, and have applied it to different group scheduling execution environments as described in Section 4.5.

The first prototype implementation of the Dynamic Scheduling Real-Time CORBA 2.0 specification in TAO [6] introduced fixed priority based and condition variable based execution control mechanisms which form the basis for the mechanisms discussed in Sections 4.3 and 4.4 respectively. Our work on group scheduling also includes single and multiple condition variable blocking strategies, along with dynamic thread scheduling mechanisms with detection of thread blocking and unblocking.

The Scout operating system [7] introduced the *path* abstraction, which made explicit the previously implicit notion of the computation's execution environment cross-cutting system layers. In Scout, paths are

composed to combine segments of the computation with mechanisms that manage their progress. Our group scheduling approach generalizes the notion of path composition to a wide range of platforms, and also shifts the path abstraction from the execution environment model to the scheduling model: with flexible execution environments like KURT-Linux or Kokyu in which group scheduling mechanisms can be expressed, group scheduling raises the level of abstraction at which computation progress is modeled and enforced.

2.2. Scheduling Decision Functions

The work on flexible scheduling mechanisms in Kokyu was originally motivated by earlier work by Steward and Khosla [14] that introduced the *maximum urgency first* (MUF) SDF and showed that MUF could emulate several other well-known SDFs through different bindings for its parameters and those of the scheduled task set. Regehr and Stankovic extended the notion of a scheduling decision function to include *composition* of SDFs within a *hierarchical scheduling* framework [11]. Recently, Regehr, *et al.*, have extended hierarchical scheduling to cross-cut alternative execution environments so that both concurrency analysis and schedulability analysis can be done together [10].

In the Scout operating system, other abstractions for composition of scheduling functions have been realized, such as the Best-Effort Real-Time (BERT) [2] algorithm. Both the hierarchical scheduling and BERT approaches achieve composition of SDFs across system layers. The BERT algorithm is an example of one kind of SDF, for slack stealing in fair-queuing SDFs, that can be realized in the more general hierarchical scheduling model. Just as our work on group scheduling complements and augments the hierarchical scheduling model, so too does it extend the BERT approach.

3. Group Scheduling Model

The group scheduling model presented here has a simple structure, but addresses several important issues related to the semantics and structure of computer system scheduling. The components of the group scheduling model are: (1) scheduling decision functions, (2) group membership specification, and (3) scheduling information for each member of a group. Furthermore, it is important to note that when more than one group is defined for the system, they are organized as a scheduling decision tree

(SDT) for the system; the hierarchical organization of all schedulers in an end-system results in the system scheduling decision tree (SSDT). Each scheduling decision begins at the root of the SSDT and recursively descends through a series of SDF invocations until a decision is made. There are two possible ways to view the SSDT; as a first-refusal scheduling decision that calls the default system scheduler if no thread is identified, or as the sole system scheduling method. As we will discuss in Section 3.3, the latter view can be used to implement the former.

3.1. Scheduling Decision Functions

Each group has a (possibly unique) SDF associated with it. An essentially unlimited number of SDFs can be implemented, but most systems can be implemented using selections from a standard set. We have implemented a number of SDFs, including: static priority, dynamic priority, explicit plan, cyclic, processor share, round robin, EDF, and sequential. The interface for a scheduling decision function is quite simple: it takes the scheduling information describing the members of the group with which it is associated as input, and it returns a decision, which can take one of three forms: a thread ID, *Pass* or *OS*. The group and SDF API includes the ability to modify the scheduling parameters associated with each member of the group. For example, if the SDF of a particular group is priority based, then the group API gives access to the SDF API that makes it possible to change the priority of group members.

It is worth noting that the simple interface for the SDF has made it easy to provide implementations within which developers can select from a set of existing SDFs, or can choose to implement customized SDFs. This significantly adds to the flexibility and configurability of the systems using the group scheduling approach, compared to systems with a single default OS scheduler of any single type.

The SDF returns a thread ID when it has identified a specific thread as the one that should run next. If the current SDF is associated with the root group of the decision tree, then the system dispatcher switches context to the specified thread. If the SDF is associated with a group below the root of the scheduling decision tree, then the calling SDF recognizes the thread ID and returns it to its calling context until the root of the scheduling decision tree is reached.

The *Pass* return value identifies when the current SDF, and any SDFs subordinate to it in the SSDT were unable to identify a thread that should run. The most

obvious reason for this to happen is that no threads managed by that portion of the SSDT were ready to run, but any given SDF might return this value for reasons of its own. When the called SDF returns *Pass*, the calling SDF is then free to choose another group member. If the SDF at the root of the SSDT returns *Pass*, then there is no thread managed by the SSDT that is ready to run.

The *OS* return value is a special token used to explicitly identify the choice of invoking the default scheduler of the underlying OS. This is useful when a set of processes controlled by a specialized SSDT should coexist gracefully with the normal set of computations on a workstation. For example, an explicit plan based SSDT might be used to control a set of real-time processes under group scheduling, but the schedule might include specific periods during which the set of computations controlled by the default OS scheduler should be given a chance to run. In this case, the explicit plan SDF would return a thread ID when the clock reached the planned execution time for each thread, and would return *OS* at the beginning of a period designated for the general OS operations.

If the SDF selects a member of the group that is itself a group, then the current SDF invokes the SDF of the selected group recursively. The recursive invocations of SDFs will terminate because the SSDT is a tree. Further development of the group scheduling may relax the single-parent-group requirement, permitting the hierarchy of the set of groups to be a DAG instead of a tree, but the recursive invocation of SDFs will be limited in either case.

3.2. Computations, Groups, and Scheduling Decision Trees

The group scheduling model permits threads to join and leave groups at their own request, or at the request of threads running at an adequate authority level. Groups can be made members of higher level groups. This is how a SDT for a computation and the SSDT for the system as a whole can be constructed.

An interesting aspect of our group scheduling approach is that it makes it possible to separate the scheduling semantics for computation components from those controlling how system resources are shared among computations. Each computation is implemented using one or more threads, which can be organized onto one or more groups, forming the SDT for the computation. The higher level organization of the SSDT, above the level for the SDTs of the individual computations, can then be viewed as

the SDT controlling how computations share the system.

This is not the only possible organizing principle for the SSDT, however, nor is it appropriate for all computations. For example, many computations have components with a variety of execution behavior constraints. Some components must meet strict timing requirements, while others can execute under less stringent requirements, or even on a best effort basis. In such a system, the SSDT might best be organized to group computation components with similar execution constraints together.

Not all interesting SSDT structures neatly partition the set of computation threads. While the simplest scheduling semantics limit each thread to membership in only one group, there are situations in which thread membership in multiple groups is helpful. One of the earliest motivations for group scheduling was the problem of how to ensure timely display of graphical data computed by a real-time display process. We were able to ensure that the thread computing the change in the display ran according to its real-time constraints, while the display itself was updated much less predictably. After some measurement and thought we realized that this was because the display of the graphical data was under the control of the X server thread, whose scheduling was not being handled according to the real-time semantics. Making the X server thread a member of the real-time group has helped by ensuring that the X server is promptly given an opportunity to run. It is not a complete solution because as yet we have no way to ensure that the X server acts on the events generated by the real-time computation.

While we are still investigating the expressive limits of the group scheduling model, it is already clear that scheduling decision trees can be used to describe a wide range of scheduling semantics for both individual computations and the set of computations in the system as a whole. We illustrate this expressive range with several examples in the next section.

3.3. Examples

This section illustrates some of the expressive power and range of the group scheduling model with two examples. The first is presented in Figure 1, and illustrates how the first-refusal group scheduling behavior mentioned earlier can be implemented. This SSDT uses the sequential SDF (*Seq*) as the root node to give (1) the group scheduling decision tree for the system a chance to choose a thread to run. If it fails to do so, returning ei-

ther *Pass* or *OS*, then the (2) Linux SDF is called to select a thread.

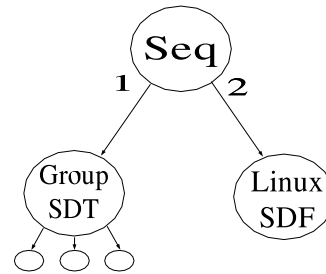


Figure 1. Group Scheduling with Linux Scheduler Default

Figure 2 shows the SDT structure for a multi-level feedback priority queue. In the figure, the root of the SDT is a group using a priority based SDF, and whose members are groups containing threads that are in the same priority equivalence classes. Within each class, a round-robin SDF is used to evenly share the CPU among members of a class. The subtle aspect of this model is that the priority SDF must know about the priority of each thread, so that as the priority changes and a thread needs to shift from one equivalence class to another, the Priority SDF can cause it to do so.

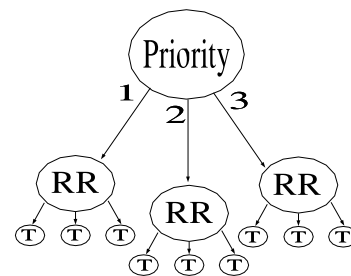


Figure 2. Multi-Level Feedback Queue

4. Group Scheduling Implementation

The group scheduling model can be implemented either in the operating system or at the middle-ware level. The operating system level is attractive because it can support an implementation that is both cleanly designed from the start, and efficient. Middle-ware implementations of group scheduling are

attractive because they require less explicit cooperation from the operating system. Specifically, moving the group scheduling technique up to middleware makes it available for use with operating systems that do not already provide group scheduling services, or for which interested users do not have source access to the kernel. Realizing group scheduling mechanisms in middleware also leads to increased portability of the approach while still maintaining acceptable performance. In particular, middleware implementations of group scheduling can help to mask variations in the set of scheduling features available across operating systems, by using the available features to support middleware group scheduling mechanisms that provide more uniform behavior than the individual system's scheduling model. However, the efficiency with which group scheduling can be realized in middleware depends on (1) the scheduling mechanisms that are available from the OS, (2) the structure of the middleware itself, and (3) the nature of the scheduled entities, several implementation approaches are reasonable.

We currently have several implementations of the group scheduling model: one at the operating system level in KURT-Linux [13], discussed in Section 4.1, and several at the middleware level currently running on top of Linux, discussed in Sections 4.2 through 4.5. The middleware approaches use the standard SCHED_FIFO scheduling class, thread priority scheduling, and condition variable mechanisms offered by Linux and other operating systems to manage the execution of threads under control of the group scheduling model. It is worth noting that in Sections 4.1 and 4.2 the majority of the code implementing the group scheduling model is shared between the kernel and middleware implementations. The shared code uses the same data structures to represent group membership, to support scheduling decision functions, to keep track of scheduling parameters for each member of each group, and to implement the SSDT. The only differences between the kernel and middleware versions lie in how access to the group scheduling API is provided, how the system scheduling decision tree is invoked to identify the next thread to run, and in how that decision is enforced. These two implementations in particular therefore offer an opportunity to examine the fundamental costs of moving group scheduling from the kernel to the middleware layer. We describe empirical results of our evaluation of these two implementations in Section 5.

After presenting the current kernel and middleware implementations, we consider how to extend the

group scheduling concepts discussed in Section 3 to scheduling *entities* at the both the OS and middleware levels. It is important to note that not all entities will necessarily be visible at both levels. At the OS level, the threads are the obvious entities, and are the only schedulable entities absent specialized support. A schedulable *entity* at the middleware layer, however, could be an OS thread, a published *event* in an Event Channel, a GIOP message, or a distributable thread.

In all these cases, the scheduling mechanisms could be the same even though the dispatching mechanisms may be different based on whether the schedulable entities are *active* or *passive*. An OS thread or a distributable thread is an active entity (control flow) whereas an *event* in an Event Channel, or a GIOP message would be a passive entity (data flow). Scheduling involves choosing the most eligible item from a set of items. Middleware frameworks like Kokyu [5] support flexible implementation of mechanisms for different scheduling strategies. Once scheduling is done, the dispatcher *dispatches* the most eligible item. In the following subsections, we discuss several dispatching mechanisms for group scheduling.

4.1. KURT-Linux Implementation

The KURT-Linux implementation of the group scheduling model is quite simple, using the shared group scheduling code, except for specific adaptations to make the API available, to invoke the SSDT, and to enforce the scheduling decision made. The group scheduling model is implemented as a Linux kernel module, and provides access by threads running under the OS to the standard group scheduling API through a standard module interface. An application thread wishing to interact with the group scheduling model opens a file descriptor to the group scheduling module, and then executes various *ioctl* calls to use the group scheduling API. Groups can be created, and threads made members of groups, using this interface, and this is how the group scheduling SSDT is constructed.

The SSDT is invoked by small section of code inserted at the beginning of the Linux scheduler, which calls the SDF associated with the root node of the SSDT. The SSDT then returns a pointer to a thread structure, a *Pass* or *OS* value. If a specific thread is chosen by the SSDT, control is transferred directly to the context switch portion of the Linux scheduler, otherwise the standard Linux code selecting the next thread to run is executed. This approach is both simple and

efficient. It requires a change to the Linux kernel code, but only of the smallest and least intrusive kind.

4.2. Detecting When Threads Block

The middleware implementation of the group scheduling model is more complex for a number of reasons. First, because utility threads must be used to support the scheduling and group API functions. Second, because indirect methods must be used to dispatch a thread, in this case a combination of priority manipulation and the use of the SIGSTOP and SIGCONT signals to control what threads can be run, and to arrange for the SSDT to be invoked. The group scheduling API is supported by a server thread listening for connection requests from threads wishing to interact with the group scheduling API. This is the middleware analog of the group scheduling module and set of *ioctl* calls implementing the group scheduling API in the kernel implementation.

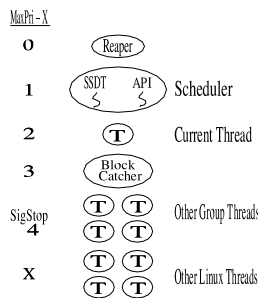


Figure 3. Middleware Group Thread Priorities

Figure 3 shows the set of threads we use to implement group scheduling in the middleware. This approach depends on the semantics of the FIFO scheduling class under Linux, and on the fact that all Linux threads that are not in the FIFO class are members of the same equivalence class, from the FIFO scheduler’s point of view. The left side of each line shows the number of priority levels below maximum at which a thread executes. The *Reaper* task runs at the highest priority, but only exists to create the *SSDT* and the *API* threads at the next level down. The existence of the *Reaper* also helps us detect and handle problems.

The *SSDT* and *API* are concurrent threads in the same process. They run at the same priority, one less than maximum, because all changes to the thread state that comes through the *API* should be done before the

scheduling decision is made by the *SSDT*. These operations could conceivably be implemented by a single thread, but we found using two more convenient. These are the operational threads that run at the highest priority, and thus are granted the CPU whenever they are ready to run. The *API* thread thus runs when a request for a group *API* operation is made, and the scheduler thread runs at times it selects according to the semantics of the *SSDT* it is running. In the experiments discussed in Section [refsec:eval](#), we used a single level round-robin scheduler because our goal was to measure the overhead of thread management under the kernel and middleware implementations.

The controlled thread which is currently supposed to be running uses a priority two less than maximum, so it runs whenever the group *API* or *SSDT* threads are blocked, which is most of the time. The *Block Catcher* thread runs at three less than maximum so that it will be selected if the current thread becomes unable to run, which is most often when it is blocked. It can also run when the current thread exits. All other threads under the control of the group scheduling middleware are sent the SIGSTOP signal, and are given a priority four less than maximum. This is somewhat redundant but we chose to do it for safety.

Choosing a thread to run is fairly simple. When the *SSDT* thread runs it invokes the system scheduling decision tree which decides what thread to run as described in Section 3. Assume for the purposes of this discussion that this is one of the threads under group scheduling control. The *SSDT* thread then decreases the priority of the current process to match that of other non-running threads under group scheduling control and sends it SIGSTOP. It increases the priority of the chosen thread to one less than maximum, and then blocks in a way that is determined by the SDF semantics. In the case of the experiments described here, it blocks in *nanosleep* because the *SSDT* thread is awaiting the end of the RR quantum.

If the current task executes for the entire quantum, then the *SSDT* thread receives a signal in response to the *nanosleep* timer expiration and runs the *SSDT* to select the thread that should run. If the current thread blocks, then the *Block Catcher* thread is selected by the Linux FIFO scheduling policy and runs. The *Block Catcher* exists only to send a SIGUSR1 signal to the *SSDT* thread to cause it to return from the *nanosleep* early. The *SSDT* thread sends the now blocked thread SIGSTOP to ensure that it will stay stopped until selected by the *SSDT*.

All of this works well, albeit with non-trivial overhead compared to the kernel implementation, except for the fact that the *SSDT* thread has no way to know

when a blocked thread becomes unblocked, which is one of the conditions under which the scheduler is run in the underlying OS. This problem was recognized some time ago, and a solution called *scheduler activations* was proposed [1]. It should be fairly easy to implement a simple version of this under Linux, so we will try it to see what effect it has in the near future.

For the moment, and because we want to make sure we fully evaluate the implications of using existing APIs for existing operating systems, we simply try to run the process that blocked, by sending it a SIGCONT and raising its priority, if it is picked by the SSDT. If the process should remain blocked it must now be written to detect this condition and restart the system call, thus blocking again. This need to write the processes to handle this condition is the biggest drawback of this approach.

We have recently received an indication that a popular RTOS, VxWorks [16] provides the capability for a thread to receive a signal when another thread blocks or begins running. This show promise as a basis for implementing a simple scheduler activation function, and we will investigate the feasibility of doing this. We will also implement it in some form under Linux to make experiments easy to perform.

4.3. Detecting When Threads Unblock

Section 4.2 described the migration of a group scheduling mechanism from the kernel into middleware. We now examine how additional constraints imposed by distributed object computing (DOC) middleware require further elaboration of the mechanism. In particular, with remote method invocations, a thread may block on a socket connection or other facility until it receives a reply from an object running on a remote host. Because during the time it was blocked the eligibility of the blocked thread may have either increased or decreased relative to the currently running thread, the unblocking of that thread must be detected and a SDF must be invoked at that point.

Fortunately, open-source middleware implementations like ACE [12] and TAO [3] offer numerous points at which to intercept the unblocking of a thread and invoke an appropriate SDF. Even middleware that is less open but which implements standards like the CORBA Portable Interceptors specification [9] can detect unblocking of threads and act accordingly, albeit with some possible reduction in the precision of when that action occurs.

Other than the adjustments to detect and act on unblocking of a thread, this approach uses dispatching

mechanisms that are still very similar to that of an OS-level dispatcher. It relies on the OS dispatcher to do the actual thread dispatching. The dispatcher enforces scheduler decisions by manipulating the priorities of the threads. The middleware-level scheduler maintains an SDF-ordered queue of threads.

The scheduler also has an executive thread, which runs at the maximum available priority. This thread runs in a continuous loop servicing newly arrived threads and making scheduling decisions. The executive thread is responsible for selecting the most eligible thread from the scheduler queue and changing its priority if necessary while manipulating the priority of the currently running thread, if it is not the most eligible. Four priority levels are required for this mechanism to work as shown in Figure 4 (or five if we also add a block-catcher level as we discuss later) . For example, a thread running at Active priority will preempt a thread running at the Inactive priority level.

1. Executive priority - priority at which the scheduler executive thread runs.
2. Blocked priority - this is the priority to which threads about to block will be changed.
3. Active priority - this is the priority to which the most eligible thread is set.
4. Inactive priority - this is the priority to which all threads except the most eligible thread is set.

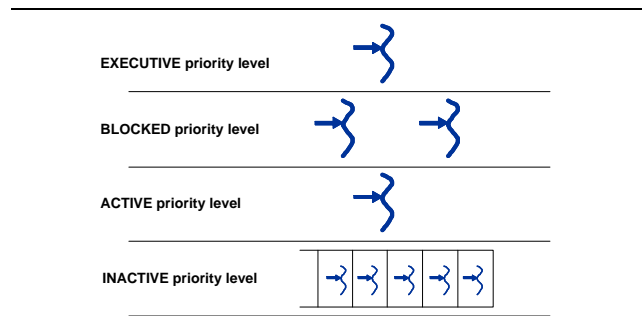


Figure 4. Priority Levels with Unblocking Detection

A newly arriving thread informs the scheduler executive thread by signaling a condition variable that is monitored by the scheduler thread. The scheduler thread picks up the most eligible thread and sets its priority to *Active* level and sets the currently running thread priority to *Inactive* level.

Drawbacks: The drawback to this approach is that it relies on the OS dispatcher to dispatch the threads,

which does not work very well if we run the threads in a non-real-time scheduling class. In contrast, the CV-based approach described in Section 4.4 works even if we run the threads in non-real-time scheduling class.

Multiprocessing: With the above approach, there is only one thread running at active priority and others are all at inactive level. This will create undesirable effects with multi-processor systems, which could select any one of the inactive level threads to be scheduled on a different processor and this could cause eligibility inversions if a thread is chosen over another with higher eligibility. One solution to this problem is to split the *Active* priority level into sub-levels corresponding to each processor on the system or to apply some kind of fair-scheduling SDF. Instead of a single thread running at the *Active* level, there could then be multiple threads that run concurrently on different processors.

Block Catcher: Instead of explicitly informing the middleware scheduler about blocking, the idea of a *block catcher* thread can be used, similar to the one used in process scheduling explained in Section 4.2. This requires another priority level called *BlockCatcher*, between the *Active* and *Inactive* levels. This does not however eliminate the need for the *Blocked* priority level, but rather bracketing the *Active* priority level with a higher priority *Blocked* level and a lower priority *BlockCatcher* levels ensures high fidelity of the enforcement of the SDF to the computation semantics as threads block and unblock.

As soon as an *Active* level thread blocks, the block catcher thread wakes up. This thread is in a continuous service loop with a single purpose - wake up the scheduler thread by signaling the condition variable on which the scheduler thread is blocked. The scheduler thread wakes up and schedules the most eligible thread from among the inactive threads and bumps its priority up to *Active*. The block catcher thread won't run until the active thread blocks, since its priority is less than that of the *Active* level thread, but when that thread does block the block catcher thread will run and signal to the executive thread that the active thread has blocked.

4.4. Condition Variable Based Dispatching

Mechanism: In this approach, which was prototyped in the DSRTC2 scheduler implementation in TAO [6], the most eligible thread runs and all other threads wait on a condition variable until they become most eligible. This approach assumes a cooperative threading model, where threads yield on a regular basis giving a chance for newly arrived threads to be sched-

uled. When a thread is ready to be scheduled, it is inserted into a reordering queue, which is a SDF-managed queue with queue elements sorted according to a particular scheduling discipline. Enqueued threads wait on a condition variable as shown in Figure 5.

Only one thread is running at any point in time. When the currently running thread yields, it causes the condition variable to be signaled. The most eligible thread from the reordering queue needs to run next. Therefore, each thread in the reordering queue, after waking up, checks whether it is the most eligible thread. Only the most eligible thread runs and all other threads block again on the condition variable as shown in Figure 5.

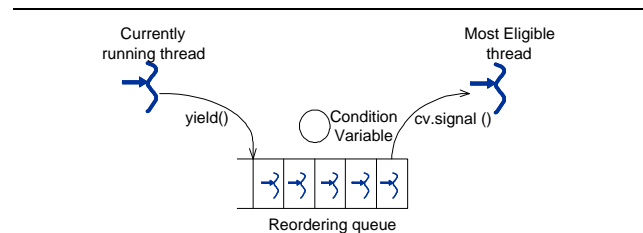


Figure 5. Most eligible thread runs while others wait on condition variable

Drawbacks: This approach has the drawback that it requires a cooperative threading model, where threads yield voluntarily on a regular basis. The application threads are responsible for doing this voluntarily.

CV per thread: A variation of the above approach is to associate a condition variable with each thread. When a scheduling event occurs (e.g., the currently running thread yields or blocks), the condition variable associated with the most eligible thread at that point is signaled. This is in contrast with the single condition variable approach where the signal is *broadcast* so that each thread will wake up and check whether it is the most eligible. If it is, it will run and all other threads go back to wait on the condition variable. The downside of this approach is the potentially higher cost to select the most eligible thread each time, when compared to the original approach. However, this additional cost is at least mitigated by the fact that the number of unnecessary context switches incurred by this approach is often less than that of the single CV approach.

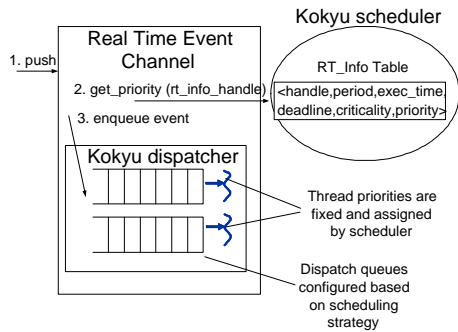


Figure 6. Interaction between the TAO RTEC, Kokyu scheduler and Kokyu dispatcher

4.5. Group Scheduling in the TAO Real Time Event Channel

In this section, we extend the concept of group scheduling to scheduling events in the TAO Real Time Event Channel (RTEC) for the Boeing Bold-stroke [4] avionics mission computing framework. Figure 6 shows the interaction between the various modules when an event is pushed into the RTEC. The Kokyu scheduler maintains a table of operations, each operation represented by a unique *handle*. The operation attributes like period, deadline, execution time, etc are aggregated together in a structure called *RTInfo*. For a detailed discussion of the scheduler architecture and internals, the reader is referred to [5]. The scheduler then assigns priorities or other ordering parameters to operations based on plug-gable scheduling algorithms.

The Kokyu dispatcher, which is used within the RTEC dispatching module contains a set of SDF-ordered queues. The configuration of the dispatcher is done using configuration information obtained from the scheduler which determines the number and type (FIFO, laxity ordered, etc.) of re-ordering queues.

On an event push, the RTEC consults the Kokyu scheduler for the dispatching lane within which the event should be delivered. The *RTInfo* handle for the supplier of the event is passed in as part of the request. The dispatcher then enqueues the event into the appropriate SDF-ordered queue for delivery to the event consumer.

Runtime Reconfiguration: To deal with runtime mode changes of varying granularity, the scheduler offers interfaces to update quality of service (QoS) parameters associated with an operation. The scheduler then

recomputes the schedule and assigns operation parameters. This approach has been highly successful in rate-based systems like avionics mission-computing systems [4]. However, this approach in isolation does not provide a computation-based reconfiguration interface that is intuitive to the application developer. We now discuss two approaches which incorporate group scheduling into the original architecture.

4.5.1. Group scheduling outside Kokyu In this approach, collections of *RTInfos* form groups as illustrated in Figure 7. Instead of the application changing the individual operation attributes during a mode change, the group scheduler takes the responsibility of changing the QoS attributes of appropriate operations. For example, a set of event suppliers can change the rates at which they push events. The set of operations changed and the nature of change is determined by group membership and the scheduling policy for that group. Note that the original dispatching architecture is still present and the notion of groups is formed on top of that original architecture.

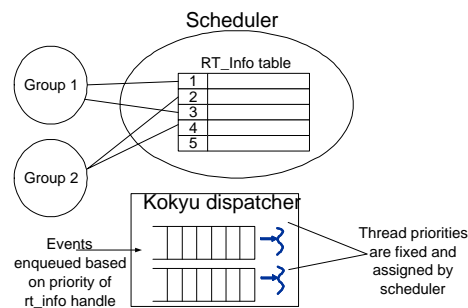


Figure 7. Groups atop the Kokyu dispatcher

4.5.2. Group scheduling inside Kokyu In this approach, collections of *RTInfos* still form groups as is illustrated in Figure 8. There is a fundamental difference between this and the previous approach, however, in that we bring the notion of groups into the Kokyu dispatcher and scheduler mechanisms. Each *RTInfo* carries one or more numbers corresponding to the groups to which it belongs. The dispatcher is configured with an SDF queue per group. The thread priorities are no longer static as in the previous approach, but are themselves managed by a higher-level SDF. They thus can be manipulated according to changes in computations in arbitrary groups. This architecture allows the isolation of events belonging to different groups and also allows increased control of

event ordering and processing based on computation progress.

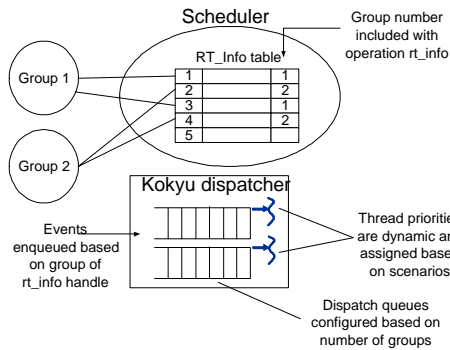


Figure 8. Groups within the Kokyu dispatcher

5. Evaluation

The performance experiment results presented here are the first level evaluation of the thread management overhead under the kernel and middleware implementations. There are several performance metrics of interest, but this section only presents the one we consider most fundamental: the controlled thread to controlled thread switching delay, as illustrated in Figure 9 for the CPU bound tasks where the *Block Catching* thread does not run. This is defined as the delay between stopping a controlled task and starting another controlled task. As measured, it begins with the context switch away from the current controlled task (Context Switch A to SSDT) and ends with the context switch to the selected controlled task (Context Switch SSDT to B).

It is interesting to note that for the middleware implementation this delay could only be calculated by using data gathered using the Data Streams [8] method from both the application and kernel levels. The reason for this is that the context switch events come from the kernel, but the event announcing which thread was selected (Selection) by the SSDT comes from the SSDT thread at the user level. The post processing of the event stream thus had to look for the event announcing which task had been selected in order to look for the proper context switch event.

In the kernel implementation measuring the time between one context switch and another would not be fair, because it would not include the SSDT execution time. So, in the a kernel implementation we mea-

sured the interval between when the SSDT is invoked and when the system switches context to the selected task. It is worth noting that the invocation of the of the SSDT follows the blocking of the first controlled task by only a few machine cycles.

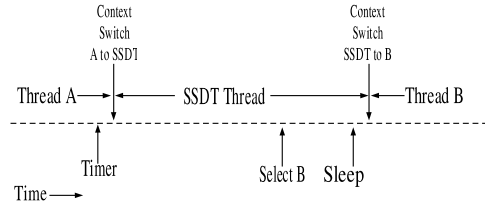


Figure 9. Middleware Task Control Event Time Line

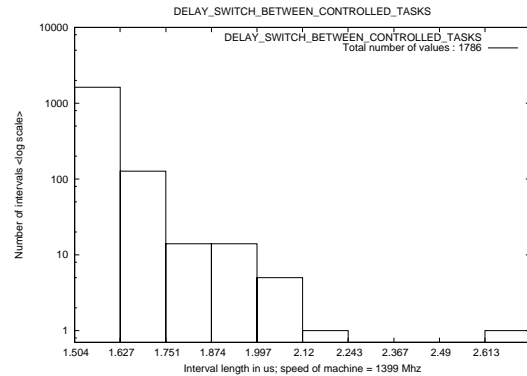


Figure 10. Kernel Group Scheduling Task Switch Delay Distribution - Blocking Threads

Figure 10 shows the distribution of delay values for the kernel implementation group scheduling implementation controlling a set of blocking tasks. Note that the task to task switch delays vary between roughly 1.5 microseconds and slightly more than 2.6 microseconds. Also note that the vertical scale of all the histograms is exponential, so more than 1000 of the thread switch delays are classified in the first equivalence class of the histogram. The test was run on a 1.4 GHz Pentium machine, and shows a set of slightly less than 1800 context switch delay values. Clearly the simple RR SSDT evaluation imposes very little overhead, and the ef-

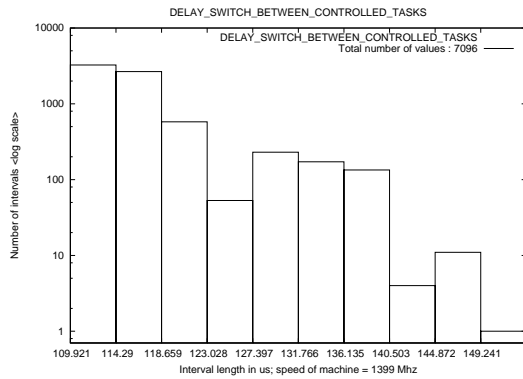


Figure 11. Middleware Group Scheduling Task Switch Delay Distribution - Blocking Threads

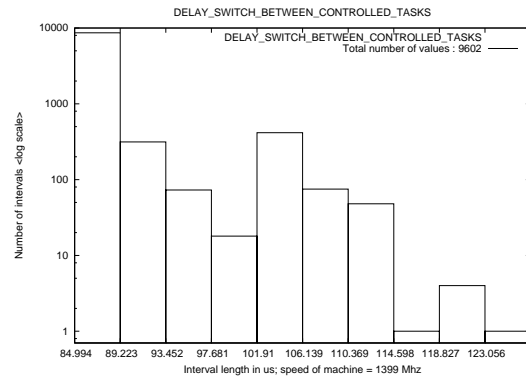


Figure 12. Middleware Group Scheduling Task Switch Delay Distribution - CPU Bound Threads

efficient context switch of the Linux platform is not compromised. It is important to note that the distribution for delay between non-blocking CPU bound tasks is essentially identical for the kernel implementation because the patch taken through the kernel code is essentially identical.

In significant contrast to this, the middleware implementation controlling the same scenario with the same SSDT code experienced task switch delays, illustrated in Figure 11, ranging from slightly more than 110 microseconds to roughly 150 microseconds. It is interesting to note that there are a significantly larger number of larger values in this distribution. This is due, at least in part, to interrupts occurring during essentially all of the longer task switch intervals, which we have been able to detect with fairly straightforward post-processing of the Data Stream output from the kernel.

Figure 12 shows the delay distribution for the middleware group scheduling implementation when controlling a set of CPU bound tasks. The distribution of thread switch delays covers roughly the same range as those for the set of blocking tasks; the lowest and highest values differ by roughly 40 microseconds. The range is, however, shifted lower by roughly 25 microseconds because with CPU bound threads context switching between controlled tasks does not require the *Block Catching* task to run, saving a significant amount of time.

6. Conclusions and Future Work

This paper has presented our group scheduling model and has shown how it can be used in a wide va-

riety of situations. We have demonstrated that a middleware implementation is possible, with significant but not unsupportable task switching delay overhead. We have also shown that a kernel based implementation enjoys significant advantages over the middleware method, and is thus clearly desirable in systems where it is possible.

We plan to continue development and use of the group scheduling model at both the kernel and middleware levels in several current projects. As part of this effort, we plan to investigate how scheduler activations can be used to increase the efficiency of the implementation for blocking tasks.

References

- [1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992.
- [2] A. Bavier, L. Peterson, and D. 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] Bryan S. Doerr and David C. Sharp. Freeing Product Line Architectures from Execution Dependencies. In *Proceedings of the 11th Annual Software Technology Conference*, April 1999.
- [5] Chris Gill, Douglas C. Schmidt, and Ron 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.

- [6] Y. Krishnamurthy, I. Pyarali, C. Gill, and V. Wolfe. Design and Implementation of the Dynamic Scheduling Real-Time CORBA 2.0 Specification in TAO. In *OMG Workshop on Real-time and Embedded Distributed Object Computing*, Alexandria, VA., July 2003. Object Management Group.
- [7] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*. USENIX Association, October 1996.
- [8] 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.
- [9] Object Management Group. *Interceptors FTF Final Published Draft*, OMG Document ptc/00-04-05 edition, April 2000.
- [10] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *The 24th IEEE Real-Time Systems Symposium (RTSS '03)*, Cancun, Mexico, December 2003.
- [11] J. Regehr and J. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. In *The 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, London, UK, December 2001.
- [12] Douglas C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/~schmidt/ACE.html, 1997.
- [13] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the shelf hardware and free software. In *Proceedings of the Real-Time Technology and Applications Symposium*, Denver, June 1998.
- [14] David B. Stewart and Pradeep K. Khosla. Real-Time Scheduling of Sensor-Based Control Systems. In W. Hahng and K. Ramamritham, editors, *Real-Time Programming*. Pergamon Press, Tarrytown, NY, 1992.
- [15] Tarr, Harrison, Osher, Finkelstein, Nuseibeh, and Perry. Workshop on multi-dimensional separation of concerns in software engineering. In *Proceedings of ICSE 2000*, Limerick, Ireland, June 2000.
- [16] Wind River Systems. VxWorks 5.3. www.wrs.com/products/html/vxworks.html.