Implementing Distributable Real-Time Threads in the Linux Kernel: Programming Interface and Scheduling Support

Sherif F. Fahmy
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
fahmy@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

E. D. Jensen
Time-Critical Technologies
Boston, MA, USA
jensen@real-time.org

ABSTRACT

We present an implementation of Real-Time CORBA's distributable threads (DTs) as a first-class, end-to-end real-time programming and scheduling abstraction in the Linux kernel. We use Ingo Molnar's PREEMPT_RT kernel patch, which enables nearly complete kernel pre-emption, and add local (real-time) scheduling support to the Linux kernel, atop which we build DT scheduling support. We implement DTs using Linux's threading capabilities. Our implementation of a suite of independent and collaborative DT schedulers confirm the effectiveness of our implementation.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed systems*
; C.3.d [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

Keywords

Distributed, Real-time, Linux, Distributable thread, Thread integrity, Distributed scheduling

1. INTRODUCTION

In distributed systems, action and information timeliness is often end-to-end – e.g., a causally dependent, multi-node, sensor to actuator sequential flow of execution. Designers and users of distributed systems, distributed real-time systems in particular, often need to dependably reason about – i.e., specify, manage, and predict – end-to-end timeliness.

Maintaining end-to-end properties (e.g., timeliness, connectivity) of a control or information flow requires a model of the flow's locus in space and time that can be reasoned about. Such a model facilitates reasoning about the contention for resources that occur along the flow's locus and resolving those contentions to optimize system-wide end-to-end timeliness. The *distributable thread* (or DT) programming abstraction which first appeared in the Alpha OS [14] and subsequently in Mach 3.0 [7], and OMG's Real-Time CORBA 1.2 standard [15] provides such a model as its first-class programming and scheduling abstraction. A DT is a single thread of execution with a globally unique identity that transparently extends and retracts through local and remote object instances (via remote method invocations).

The objects in the DT model are passive (as opposed to active objects that encapsulate one or more local threads). An object instance resides on a single computational node. A distributable thread enters an object instance by invoking one of its operations; that portion in the object instance is a *segment* of the DT (typically implemented as a local thread). If a DT invokes a sequence of methods in object instances on the same node, that sequence of segments is called a *section*. Only the "head" segment of a DT is active (executing), all others are suspended (i.e., they have made remote invocations to other nodes, eventually to the head).

Approaches for the real-time of scheduling of DTs can be broadly classified into two categories: *independent* and *collaborative* scheduling. In the independent approach (e.g., [3]), DTs are scheduled at nodes using DT scheduling parameters (e.g., end-to-end deadline, end-to-end utility) that are propagated along with DTs, and without interaction with other nodes (thereby not considering node failures during scheduling). Fault-management is separately addressed by *thread integrity protocols* [8] that run concurrent to thread execution. Thread integrity protocols employ failure detectors (FDs) [1], and use them to detect failures of DT segments or sections [3]. Thus, any uni/multiprocessor real-time scheduling algorithm can be used in this paradigm for DT scheduling, as all their required scheduling parameters are available locally once the DTs become locally active.

In the collaborative approach (e.g., [4, 6, 16]), nodes construct system-wide thread schedules distributively, anticipating and detecting node failures using FDs. Different distributed algorithmic paradigms can be used for such distributed schedule construction – e.g., CUA [16] and ACUA [6] algorithms use consensus; QBUA [4] and DQBUA [5] use quorums; RTG-DS [9] and RTG-L [10] use gossip. Both independent and collaborative scheduling approaches are included in the Real-Time CORBA specification.

We describe implementing DTs as a first-class end-to-end programming abstraction in the Linux kernel and as a first-class kernel scheduling construct, allowing for both independent and collaborative DT scheduling. We first modify the Linux kernel to add local real-time scheduling support (Section 2). We implement DTs using Linux's threading

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

capabilities and export a DT API that is largely similar to Real-Time CORBA's (Section 3). The DT schedulers in the literature assume uniprocessor nodes, and we follow that assumption, but augment with an implementation optimization, pioneered in Alpha [14], to optimize collaborative scheduling: *co-scheduling* – i.e., use multicore nodes, with one core for running schedulers and related functionality, and other cores for application logic (Section 4). As a proof-of-concept, we implement a set of independent and collaborative schedulers and measure effectiveness (Section 5).

Our motivation for this effort is simple: Linux is one of the most mature OSs with wide acceptance and traction across various form factors – from enterprise-level servers to embedded devices. The only publicly available DT platforms are either middleware (e.g., [11, 19]) or OSs that are no longer extant [7, 14]. While middleware has compelling advantages for supporting DTs, the natural home of DT schedulers is the OS kernel because DTs are then scheduled coherently with the other native OS abstractions and mechanisms (synchronizers, etc.). Our Linux implementation – the first such – fills this gap and constitutes our contribution.

2. LOCAL SCHEDULING

2.1 The PREEMPT_RT Patch

The stock Linux kernel provides soft real-time capabilities, such as the POSIX primitives and the ability to set priorities. The kernel provides two real-time scheduling policies – **SCHED_FIFO** and **SCHED_RR** and a "nice"-based policy called **SCHED_NORMAL**. **SCHED_FIFO** executes processes in FIFO order within a priority and will only preempt a task to execute a newly arrived task at a higher priority. This is primarily a POSIX-specified feature. **SCHED_RR**, on the other hand, schedules processes of the same priority in a round robin fashion while favoring higher priority tasks. To introduce real-time semantics to the kernel, a preemptible kernel is required. To achieve this, we use Ingo Molnar's fully preemptible **PREEMPT_RT** kernel patch [13] which enables nearly complete kernel preemption. The patch improves interrupt latencies and allows preemption of most parts of the Linux kernel except for a few regions which are inherently unpreemptible, such as the task scheduler. In-kernel locking primitives have been reimplemented using wait queuing mutexes. As a result, critical sections that are protected with spinlocks or read-write locks can sleep and are therefore preemptible. Priority inheritance has been implemented for in-kernel spinlocks and semaphores. Interrupt handlers in the kernel have been converted into preemptible kernel threads.

2.2 Real-time Scheduling

To cause minimal disruption to the Linux kernel's scheduling framework, we implement real-time schedulers as subtypes of the **SCHED_FIFO** scheduling class. We extended the **task_struct** struct, which is used in the kernel to represent processes and threads (note that Linux implements a 1:1 threading model) by adding a new data member **rt_info**. **rt_info** contains scheduler-specific information, including a member, **rt_sched**, that can be used to specify which scheduling algorithm is to be executed. This member is only examined if the task is in the **SCHED_FIFO** scheduling class. Additionally, the **rt_info** struct contains scheduling parameters such as deadlines, periods, execution times, and utility.

The Linux scheduler calls the **pick_next_task_rt** function to select the next real-time task to run. We exploit this feature by calling our schedulers from within this function and setting the return value of **pick_next_task_rt** to the return value of our scheduling algorithm. We thus keep the Linux dispatching and scheduling code unchanged and only modify the portion that selects the next task to be executed.

Our scheduling API starts a real-time task as a high priority **SCHED_FIFO** task. It then calls **setrtinfo**, a custom system call we added to the kernel. This call fills the **rt_info** struct with the appropriate scheduling parameters, reduces the task priority, and sets the **TIF_NEED_RES-CHED** flag of the current task, causing **schedule** to be called on return from the system call. Note that the task always starts at a higher priority than the rest of the real-time tasks managed by our scheduling algorithm. This allows it to immediately gain access to the processor and invoke the system call that sends its scheduling parameters to the kernel.

Instead of calling **schedule** manually from within **setrtinfo**, we set the **TIF_NEED_RES-CHED** as mentioned above. This flag is checked after a return from a system call. When the kernel finds this flag set in the current process, it reschedules the task, by calling **schedule**, and places it in the same queue as the other real-time tasks managed by our schedulers in the multi-level priority queue used by Linux to represent the processor ready queue. (The **SCHED_FIFO** algorithm simply selects the head of the ready queue with the highest priority.) The call to **schedule** puts the kernel's scheduling process into motion. This eventually calls the **pick_next_task_rt** function, which calls our custom schedulers to select the next task to execute – this ensures that **schedule** is not called in an unsafe state by allowing the kernel to automatically call it when returning from the system call with the **TIF_NEED_RES-CHED** flag set.

3. THE DISTRIBUTABLE THREAD API

We base our DT implementation on the Native POSIX Thread Library (NPTL) in Linux, which provides near complete POSIX compliance, and is fully integrated into the GNU C library, glibc, using features provided by the 2.6 kernel. The NPTL provides a 1:1 implementation of threads in Linux – i.e., each thread is a kernel-schedulable entity. Thus, each thread is represented by a **task_struct** in the kernel and is scheduled using the kernel's scheduling mechanism.

Our DT implementation has two parts: 1) a middleware that allows application programmers to make use of the DT semantics in user-space, and 2) a set of kernel operations that keeps track of the DT as it transits across nodes.

The middleware consists of three different components:

- **Name server**: The component maintains a list of the services hosted by each node. When the middleware starts, it exchanges information with its peers to obtain the global services-to-node mapping. The name server is consulted when a program wishes to make a remote invocation in order to determine the IP address of the remote node.
- **Portable interceptor (PI)**: The PI listens for remote invocations and spawns new threads to accommodate the incoming service requests. The PI also sends information about the spawned sections to the kernel so that it has complete knowledge of the identity of the DTs it hosts and their sections.
- **Service library**: This library implements the func-

tionality of the services hosted locally. The library functions are linked to by the PI (when a service request arrives) using Linux's dynamic linking functions.

When a remote invocation is made, the PI spawns a new thread to contain the new DT section. Information about this section is sent to the kernel via a set of system calls so that the kernel can maintain a global view of the DTs it hosts. In particular, the kernel uses a system-wide ID, which we refer to as the **gtid**, to identify the DT for scheduling purposes. Scheduling parameters are propagated to the PI during remote invocation and are enforced locally by the DT API. Each section returns its exit status to the main application so that it can determine whether or not it has successfully completed.

One important feature of the DT API is that each section is represented by a POSIX thread on the local node hosting it. The DT API spawns a new thread and sets its scheduling parameters using the **setrtinfo** system call as previously described. DTs that make the system unschedulable or cannot accrue timeliness utility are removed from the system by most DT scheduling algorithms [4, 6]. We accomplish this by sending the **SIGUSR1** signal to the affected DTs. The thread then runs any appropriate exception handlers it has registered in order to bring the system to a safe state and returns a status code to indicate that it was abnormally terminated. Whenever a section starts on a node, its PID (retrieved by Linux **gettid**) is sent to the kernel and stored in the DT table (see the next section). This PID is then used to communicate with the section by sending it the appropriate signals.

In order to simplify the implementation, we consider each job of a real-time task as a separate thread. This nicely fits the aperiodic task model and can be used to mimic the periodic model by releasing a new thread representing the next invocation of the task using appropriate timers. We use high precision POSIX timers (based on HPET or TSC) to start invocations of the periodic tasks.

3.1 The API

We added a set of system calls to store and retrieve information about DTs in the kernel. At the kernel-level, a table, implemented using kernel linked lists, is used to store information about the DTs it hosts. The table is protected using a traditional spinlock (i.e., **raw_spinlock_t**).

In the API, a new type, **dt_handle_t**, is defined to identify the DTs. This data type is defined as follows:

```
typedef unsigned long long dt_gtid_t;
typedef struct dt_handle dt_handle_t;
struct dt_handle {
       dt_gtid_t gtid;
       long secid;
       pthread_t ltid;
};
```

The **gtid** member of the struct is used to store a 64-bit unique ID representing the DT, the **secid** is used to keep track of the section currently being executed, and the **ltid** member is the traditional **pthread_t** type of the POSIX thread representing the first section in the DT. It is used to wait for the DT using the standard **pthread_join** NPTL function (discussed in detail later).

There are three main functions that a program using DTs needs to call. All of these functions take a **dt_handle_t** as at least one of their parameters. The functions include:

```
int dt_spawn(dt_handle_t *handle, void (*function)
     (dt_handle_t *, void *), void * args);
int dt_remote_invoke(dt_handle_t *, const char *name,
     void * args, struct rt_info *rt_param);
int dt_join(dt_handle_t *handle);
```

We now describe each of these functions.

3.2 API: dt_spawn

This function is used to spawn a new DT. Once the function returns, a new DT has been created and the **pthread_t** handle of the POSIX thread that represents the first section in the DT is placed inside the **ltid** member of the DT's handle. Subsequently, **ltid** can be used to wait for the entire DT to complete execution using the standard **pthread_join** function as we discuss below.

In our implementation of real-time DTs, we use the first section of the DT as a placeholder for the DT – it does not perform any real-time computing but, instead, is responsible for calling the real-time sections of the DT. The actual real-time sections are started from within this first section using the **dt_remote_invoke** function. We also use this function to start local as well as remote sections of the DT.

Note that **dt_spawn** is used to create the first "dummy" section that then spawns the functional sections of the DT by calling **dt_remote_invoke**. Also, the **dt_remote_invoke** calls are used to start sections regardless of the identity of nodes they are hosted on. The name server ensures that the invocation is sent to the appropriate node, regardless of whether this node is remote or local.

When the **dt_spawn** function is called, it first invokes a system call that generates a unique 64-bit ID for the new DT being created – this involves creating a new entry for the DT in the kernel DT table. It then starts another thread to actually represent the first section of the DT. This new thread invokes a system call to store the information of the section it represents in the DT's kernel DT table entry.

Since each POSIX thread on Linux is represented by a kernel thread, we use the Linux **gettid()** call to retrieve the PID of the kernel entity representing the current thread. This information is stored in the kernel DT table and is used to communicate with the section, by sending it appropriate signals, when necessary.

After successfully storing this information in the kernel, the function passed in to **dt_spawn**, in its parameter list, is called. This function contains the code of the first section of the DT and can contain any application-specific operation.

When this function call returns, a system call is invoked to remove the information of the current section from the kernel DT table, and the DT entry is cleaned up since the termination of this first "dummy" section of the DT implies that the entire DT has finished execution.

3.3 API: dt_remote_invoke

When **dt_remote_invoke** is called, it first sends a query to the name server to check if the target service exists on the (target) remote node. Note that, as stated earlier, we use this function to call services that are hosted on either the local host or a remote node. (The name **dt_remote_invoke** may be a bit of a misnomer in this case.)

If the requested service does not exist, an error code is returned. Otherwise, the name server returns the IP address

of the node that hosts the service. If the service does exist, a system call is invoked to check whether or not an entry for the current DT exists in the kernel. If this is the first remote invocation to a remote node, there will be no entry in the kernel to store information about the DT. It is at this point that the entry is created on such nodes.

This is followed by **dt_remote_invoke** making an invocation to a system call that sets the status of the current DT to "remote". This information can be used by other algorithms, e.g., thread integrity protocols, to determine a DT's status and health. A call is made to a library function that creates a UDP connection to the node hosting the service. Finally, **dt_remote_invoke** sends the remote node the name of the required service and the scheduling parameters to be used to schedule the section that will be spawned to execute the service. Once this information is sent, the thread blocks on a receive from the remote (possibly localhost) node.

When the remote node receives the UDP message, it spawns a new thread to handle it. The spawned thread first checks with the name server to see if the service name exists locally. If it does, a system call is invoked to store information about the DT making the invocation. The scheduling information sent is then extracted from the UDP message received, and a new thread is invoked and sent this scheduling information. The newly invoked thread is responsible for executing the remote section. (We will discuss the reason a separate thread is invoked to execute the local section instead of using the thread that was spawned to handle the UDP message.)

The newly created thread first installs a signal handler for the **SIGUSR1** signal. The thread then raises its priority by calling **sched_setscheduler** with **SCHED_FIFO** as the scheduling class to be used. This is followed by a call to **sched_set-affinity** to change the CPU affinity of the thread to the other processor. (We set the CPU affinity of the **init** process at boot time to force all functionality to move to one processor.) We call this processor the scheduling co-processor. The scheduling co-processor is responsible for running all the DT API code and the kernel modules implementing the collaborative scheduling algorithms.

The real-time sections themselves are executed on the other processor. The call to **sched_set-affinity** implements this by moving the thread that will be hosting the real-time sections to the other processor. Once this is done, a call to **setrtinfo** is made and the scheduling parameters received in the UDP message are passed to this function to set the real-time scheduling parameters of the thread.

Immediately following this, a system call is made to add information about the section to the kernel (this system call also sends information to the collaborative scheduling modules if they are active – this is further elaborated in Section 4). This is followed by the actual real-time code.

Recall that the kernel module implementing the DT schedulers sends the **SIGUSR1** signal to the thread that is to be terminated. The signal handler that is installed when the thread begins is programmed to return a status code indicating that the thread has been terminated. Appropriate cleanup handling code is placed in this handler.

When this thread exits, either by successfully finishing execution or receiving the **SIGUSR1** signal from the scheduler, it returns to the thread that spawned it – namely, the thread that received the UDP message from the invoking node. This thread then checks the return code of the returning thread to see whether it has successfully finished or if it has been terminated by the scheduler. In case of the former, an "OK" message is sent to the invoking node, which is blocked on a receive waiting for a reply from the remote node. In case of the latter, a "TR" message is sent.

The main reason that we spawn a new thread and then install a signal handler on it (instead of installing a signal handler on the thread that receives and sends UDP messages from/to the invoker) is that we want to send a message to the invoker and do not want the thread to be terminated until it communicates with the invoker about its status. It would be possible to carefully construct a mechanism for determining whether a message has been sent to the invoker or not and then sending it from the signal handler using asynchronous-safe system calls, but it presents too large a potential for introducing bugs. Therefore, we spawn a new thread and have it execute the section.

Note that POSIX specifies that a signal sent to a process can be handled by any threads that have been spawned by that process. One of the major POSIX incompatibility issues that was addressed in NPTL was providing this signal handling semantics. However, we wish to send signals to specific threads within the application – the ones hosting the sections of the DTs. We do this by taking advantage of Linux's 1:1 implementation of POSIX threads.

We use the **send_sig_info** system call, with the **task_struct** of the thread executing the section as its parameter, to ensure that the signal is delivered to this thread. The actual **task_struct** is retrieved by calling **find_task_by_pid** with the result of **gettid** on the thread being targeted. Note that we may call **send_sig_info** from within the scheduler.

send_sig_info makes a number of invocations to a set of functions that acquires locks protecting scheduling data structures. Since we may call **send_sig_info** from within the scheduler, this results in a classic deadlock lock acquiring pattern with the function trying to acquire a lock that is already held by its invoker. In order to circumvent this problem, we add a field, **about_to_abort**, to the **task_struct** representing tasks in the kernel. Before calling **send_sig_info** to abort the task, we set this field to one. We then modify the code path taken by **send_sig_info** to avoid acquiring these locks if **about_to_abort** is set. This allows us to avoid the deadlock scenario by not attempting to acquire locks that were already held when **send_sig_info** was called by the scheduler. Mutual exclusion is not sacrificed in this case because the data structures are already protected by the locks acquired from earlier on in the scheduler.

*The **dt_join** function.* This function provides functionality equivalent to **pthread_join** on a DT. It calls **pthread_join** on the **pthread_t** ID of the first section in the DT.

4. THE CO-SCHEDULING APPROACH

We now describe how we use a scheduling co-processor to run the scheduler and the DT API. We divide the scheduler into two logical parts. The first part is responsible for responding to distributed scheduling events (e.g., the arrival of new DTs, the removal of a DT, and node failures), and the second part is responsible for performing local scheduling and dispatching of the sections on the node.

We first focus on the first part. Whenever a new thread arrives, this part of the scheduler is responsible for informing other nodes about the thread arrival, sending them the scheduling parameters of the new thread, and requesting the start of collaborative scheduling. Note that collabora-

tive scheduling involves scheduling DTs and local threads on the nodes that have DTs on them, by sharing information among the affected nodes about all those threads; because of overloads or resource contention, urgency and importance may result in the scheduling algorithm being used terminating some threads (both DTs and local). Based on the result of collaboration between the nodes, a set of threads, possibly empty, is selected for removal, based on a system-wide (often application-specific) scheduling optimality policy.

This set of threads is sent the **SIGUSR1** signal so that they can execute their appropriate exception handlers. This part does not perform any actual thread dispatching. We implement that functionality in a kernel thread on the scheduling co-processor using CPU affinity to bind the thread to the appropriate processor. This module (actually a thread) is now referred to as yet another schedulable entity. The module is notified of the events it should handle by waiting on a wait queue. The events are placed on this wait queue by the system calls issued by the DT API. The details of this module are specific to the scheduling algorithm in use.

The second part of the scheduler is responsible for selecting one of the threads that remain after the kernel module has eliminated threads according to the scheduling policy. In all our current target DT scheduling algorithms [4–6,16], the second part of the scheduler implements EDF (with priority inheritance in the case of DQBUA). This is implemented, as discussed before, by hooking into **pick_next_task_rt**, sorting the ready queue at the appropriate priority in increasing order of deadlines, and dispatching the top of the queue.

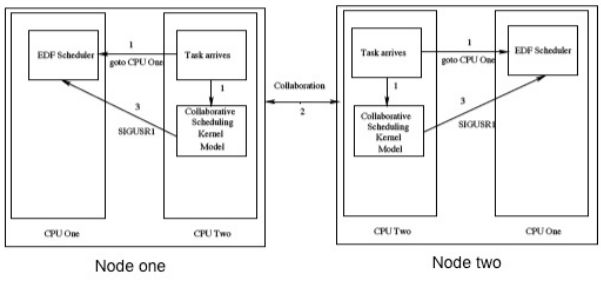


Figure 1: Co-Scheduling approach

Figure 1 depicts our co-scheduling architecture. We only show two nodes for ease of exposition. When a new real-time task (DT) arrives at either of the two nodes, it sends information about itself to the collaborative scheduling kernel module, and then transitions to the other processor in the node by calling **sched_setaffinity**. These steps are labeled "1" in Figure 1 to indicate that they are the first steps.

Once the call to **sched_setaffinity** succeeds and the task moves to processor one, the collaborative scheduling kernel module starts collaboration with the other nodes (step 2). Once collaboration is over, the result of the collaboration is a set of threads that should be removed from the system. This is accomplished when the collaborative scheduling kernel module sends the **SIGUSR1** signal to the threads that need to be terminated (step 3). The threads remaining after this are scheduled using EDF on processor one.

4.1 Collaborative Scheduler Commonalities

The heart of the co-scheduling approach is a high priority kernel module that waits for distributed scheduling events. A wait queue is defined in the kernel to enable communication between the DT API and the kernel module. When the DT API makes system calls involving events of interest to the co-scheduler, a corresponding event is placed on this wait queue for processing by the kernel module. The wait queue is defined as follows:

```
wait_queue_head_t my_wait_queue;
```

In addition to the system calls in the DT API, we add additional system calls to handle events not present in the API. For example, some collaborative algorithms require knowledge about all sections of an arriving DT. To allow this, we add a system call that the application can use to register the list of sections of a DT with the kernel module, before starting the DT. This list is placed on **my_wait_queue** for processing by the kernel module. We also define another system call that can be used at the end of DT invocations to release memory allocated for the DT in the kernel.

These calls allow us to store information about the created DT locally, and also update other nodes so that they can take part in collaborative scheduling. This is accomplished by placing a newly arrived DT's section list into a UDP packet and sending it to the remote nodes, which have a high priority kernel module listening for such messages. Once this module receives such a message, it makes a system call with the section list so that the information about the newly arrived DT is included in the kernel at its end.

This also serves another purpose. Recall that a new DT's arrival is a distributed scheduling event. Thus, the arrival of a message to the kernel module listening for DT creation messages is a trigger for collaborative scheduling. As soon as the information about the newly arrived DT is included in the kernel data structures, the co-scheduling kernel module begins executing the collaborative scheduling algorithm.

As previously mentioned, when a section arrives into the system, a new POSIX thread is spawned to accommodate it. The PID of this thread is sent to the co-scheduling kernel module via **my_wait_queue**. The main purpose for this operation is the ability to subsequently communicate with the section by sending signals to its PID.

A common feature of most collaborative scheduling algorithms [4,6,16] is the ability to create an EDF schedule and check its feasibility. We provide this functionality in a set of functions that can be called by the kernel module (e.g., **create_schedule** and **isFeasible**).

When collaborative scheduling is triggered, each node sorts its list of sections according to its algorithm-specific criteria. It then inserts these sections into the ready queue. This is followed by a call to **create_schedule**, which effectively sorts the threads into EDF order, considering the possibly different arrival times. Once this is done, a call to **isFeasible** is made. If the schedule is feasible, the next thread is added to the schedule and the process is repeated. Otherwise, the thread is added to the list of threads to reject.

Depending on the scheduling algorithm in use, the threads to reject are either used as inputs to a round of consensus so that all nodes agree on the set of threads to reject (as in CUA [16] or ACUA [6]), or are unicast to nodes that host the sections so that they can be removed (as in QBUA [4]).

5. EXPERIMENTAL EVALUATION

5.1 Co-Scheduling

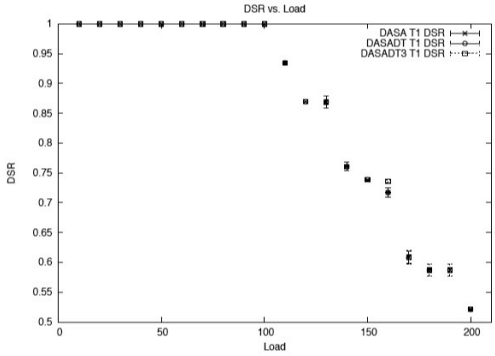
We first evaluate co-scheduling's effectiveness. We implemented Clark's DASA scheduler [2] in the co-scheduling approach. We selected DASA, as it is an excellent example of a uniprocessor scheduling algorithm that allows thread dependencies, and allows thread urgency to be orthogonal to thread importance, which is specified using time/utility functions [12]. Also, DASA defaults to EDF during underloads and no dependencies, and performs extremely well during overloads. Specifically, the kernel module on the co-scheduling processor selects a set of threads that need to be eliminated according to DASA and sends them the **SIGUSR1** signal. The threads that are not terminated are scheduled using EDF.

Table 1: Task set parameters

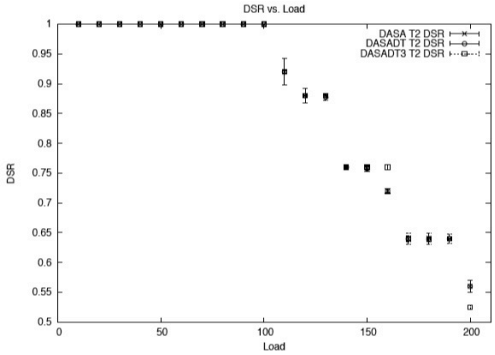
	Period (μsecs)	Utility
T1	500000	17
T2	1000000	4
T3	1500000	24
T4	3000000	39
T5	5000000	18

	Period (μsecs)	Utility
T1	1000000	17
T2	2000000	4
T3	3000000	24
T4	6000000	39
T5	10000000	18

We performed this experiment for two task sets (Table 1) with periods equal to deadlines, and measured the average deadline satisfaction ratio (DSR) for fifty runs each and the 95% confidence interval. Task execution times were varied to produce utilizations between 10 and 200. Our platform is an Intel E5300 dual-core CPU, 2.60GHz/1GB, running kernel 2.6.24.7/rt27 PREEMPT patch plus our custom extensions.



(a) DSR vs. Utilization: Task set I



(b) DSR vs. Utilization: Task set II

Figure 2: Effectiveness of co-scheduling: DASA's performance without DT API; with DT API; and with DT API and co-scheduling

We performed three different experiments: 1) DASA without using the DT API; 2) DASA using the DT API, which incurs the overhead of the DT API; and 3) DASA using the co-scheduling kernel module approach, which incurs the overhead of the DT API and the co-scheduler approach. Figures 2(a) and 2(b) show the results.

We observe that the third approach ("DASA DT 3") closely follows the other two, illustrating co-scheduling's effectiveness. Note that there are slight differences between DASA DT 3 and the other two approaches just after 100% utilization and at the high end of utilizations close to 200%. This occurs due to three reasons: First, DASA DT 3's feasibility analysis ignores overheads inherent in the scheduling mechanism. Second, since DASA DT 3 does not stop the entire system when making its decision, it is possible for some tasks that have been selected for elimination by the co-scheduler to actually complete execution before they are signaled for termination. Finally, the co-scheduling approach incurs lower overhead than the bare-bones scheduler in certain cases since it does not stop tasks for the duration of DASA (which is more expensive than EDF).

The third point is responsible for the DSR gains in Figure 2(a) and in some of the data points in Figure 2(b). As can be seen, DASA DT3 outperforms the other two approaches for most data points. This is because DASA is run concurrently with other threads in the system on the co-processor and only affects the system when it sends a termination signal to the relevant threads. Otherwise, the system is only stopped for the duration of EDF, which selects a task to run from those that have not been signaled for termination.

Note, however, that at the 200% data point in Figure 2(b), the co-scheduler approach performs worse than the two others. This is due to the first two reasons mentioned earlier.

5.2 Independent Scheduling

We now evaluate independent scheduling algorithms for DT scheduling. Recall that any uniprocessor scheduling algorithm can be used in this paradigm for DT scheduling. We thus consider independent scheduling with RMS (same as the approach used in [18]), EDF, DASA, and LBESA [12].

Table 2: DT set parameters

	Period (μsecs)	Utility
T1	1524000	11
T2	977000	12
T3	2135000	17
T4	1337000	25
T5	2533000	35

Our testbed includes two machines (Intel E5300 dual-core CPU, 2.60GHz, 1GB; Intel T3400 dual-core CPU, 2.16GHz, 2GB), connected via 100Mb/s Ethernet, and both running kernel 2.6.24.7/rt27 PREEMPT patch plus our extensions.

We consider five DTs, with each thread consisting of two real-time sections. Table 2 shows the DT set (deadlines equal periods). Figures 3(a)–3(d) show the DSR and accrued utility ratio (AUR), which is the ratio of the total accrued utility to the total number of DT releases. The values are plotted against the *transactional load*, which is the ratio of the sum of the execution times of all sections in the DT to its end-to-end period (aggregated for all DTs). As Figures 3(c) and 3(d) indicate, independent scheduling with DASA consistently outperforms others, with LBESA [12] a

close second.

Independent scheduling with EDF is optimal during underloads, while that with RMS is not. This is evident in Figure 3(c), where RMS begins to miss deadlines at 140% load, while EDF only begins to miss deadlines after 180% load. However, after 180% load, EDF's DSR quickly degrades (DSR is just above 0.1 at 240% load). In contrast, RMS's DSR is competitive with DASA and LBESA.

Figure 3(d) shows that, despite RMS's competitive DSR with DASA and LBESA (and sometimes better DSR), DASA and LBESA consistently yield higher AUR. This is because DASA and LBESA (heuristically) select the "right" set of tasks to increase the accrued utility.

5.3 Collaborative Scheduling

We now evaluate collaborative scheduling algorithms including CUA [16], ACUA [6], and QBUA [4]. As a baseline, we also include independent scheduling with HUA [17], which extends DASA with failure-handler scheduling (HUA defaults to DASA when handler overheads are negligible).

We considered the DT set in Table 2, and measured the average and 95% confidence interval of the DSR and AUR for 100 runs of each setting for each experiment and plotted them against the transactional load. Failure handlers were configured to consume 10% of processing time. Figures 4(a)–4(d) show the results. The difference between HUA and QBUA increases from 13% to 18%, that between CUA and QBUA expands from 8% to 13%, and that between ACUA and QBUA increases from 2% to 3%. Note that ACUA tracks QBUA quite closely – i.e., it is minimally affected by the increased opportunity to show-case its ability to better increase AUR against QBUA due to the increased deadline misses introduced by the failure handlers. This is due to the fact that the main difference between QBUA and ACUA is their overhead terms, not their AUR properties. (QBUA reduces the overhead of the consensus-based ACUA algorithm by a quorum-based approach, which reduces the communication patterns from broadcasts to multicasts and is invariant with respect to the number of failures.)

6. CONCLUSIONS

Our results confirm that DTs can indeed be supported as a first-class, end-to-end programming and scheduling abstraction in the Linux kernel with acceptable overhead. Co-scheduling is effective. Independent and collaborative scheduling show their respective merits: uniprocessor scheduling algorithms' (uniprocessor) behaviors carry over to distributed systems in the independent DT scheduling approach; collaborative scheduling algorithm design behaviors are preserved in the implementation.

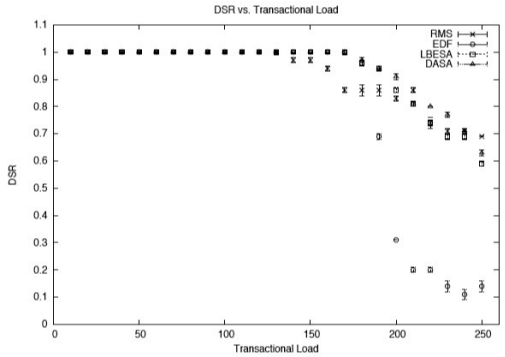
Our Linux implementation is open-sourced under GPL, and is publicly available at `chronoslinux.org`.

7. REFERENCES

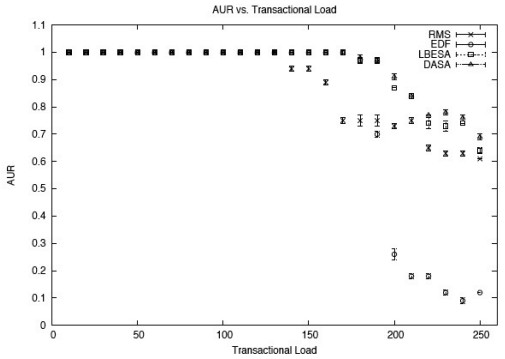
[1] M. K. Aguilera, G. L. Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC*, pages 354–370. Springer-Verlag, 2002.

[2] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.

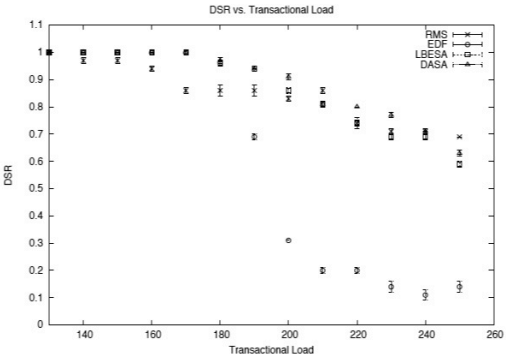
[3] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time



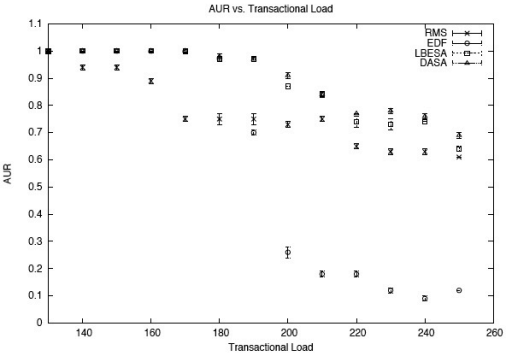
(a) DSR vs. Utilization



(b) AUR vs. Utilization

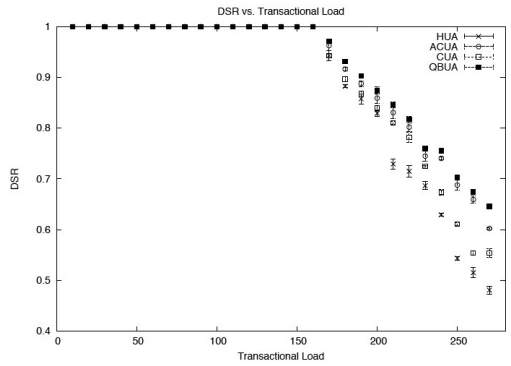


(c) DSR vs. Utilization (overloads)

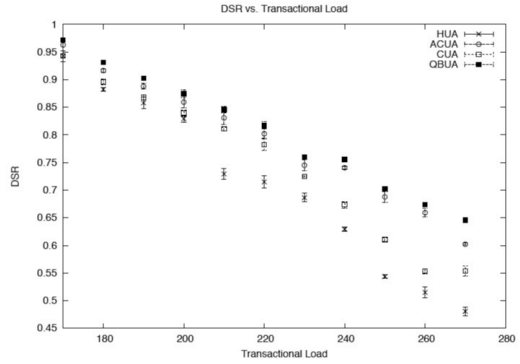


(d) AUR vs. Utilization (overloads)

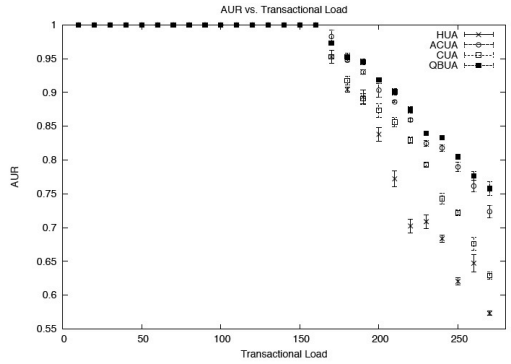
Figure 3: Effectiveness of independent scheduling: RMS, EDF, DASA, and LBESA



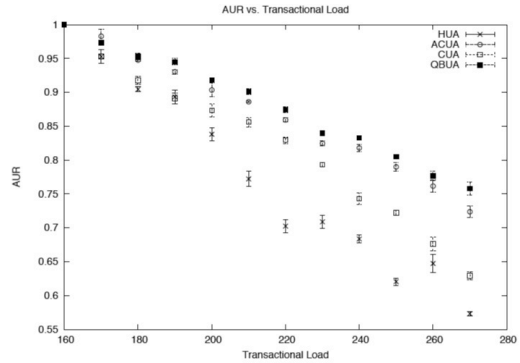
(a) DSR vs. Transactional Load



(b) DSR vs. Transactional Load (overload)



(c) AUR vs. Transactional Load



(d) AUR vs. Transactional Load (overload)

Figure 4: Effectiveness of collaborative scheduling: HUA, ACUA, CUA, and QBUA

distributed kernel. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, 1992.

[4] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Fast scheduling of distributable real-time threads with assured end-to-end timeliness. In *Ada-Europe*, pages 211–225, 2008.

[5] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Scheduling dependent distributable real-time threads in dynamic networked embedded systems. In *IFIP DIPES*, pages 171–180, 2008.

[6] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Scheduling distributable real-time threads in the presence of crash failures and message losses. In *ACM SAC*, pages 294–301, 2008.

[7] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *USENIX Technical Conf.*, pages 97–114, 1994.

[8] J. Goldberg, I. Greenberg, et al. Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, SRI International, January 1995. `http://www.csl.sri.com/papers/sri-csl-95-02/`.

[9] K. Han, B. Ravindran, and E. D. Jensen. Exploiting slack for scheduling dependent, distributable real-time threads in mobile ad hoc networks. In *RTNS*, pages 225–234, 2007.

[10] K. Han, B. Ravindran, and E. D. Jensen. RTG-L: Dependably scheduling real-time distributable threads in large-scale, unreliable networks. In *PRDC*, pages 314–321, 2007.

[11] P. Li, B. Ravindran, H. Cho, and E. D. Jensen. Scheduling distributable real-time threads in Tempus middleware. In *IEEE ICPADS*, pages 187 – 194, 2004.

[12] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, CMU, 1986. CMU-CS-86-134.

[13] I. Molnar. CONFIG_PREEMPT_REALTIME, 'Fully Preemptible Kernel', VP-2.6.9- rc4-mm1-T4. Available `http://lwn.net/Articles/105948/`.

[14] J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel*. Academic Press, 1987.

[15] OMG. Real-time CORBA 1.2: Dynamic scheduling specification. Technical report, Object Management Group, September 2001.

[16] B. Ravindran, J. S. Anderson, and E. D. Jensen. On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In *IFIP SEUS*, pages 67–81, 2007.

[17] B. Ravindran, E. Curley, et al. On best-effort real-time assurances for recovering from distributable thread failures in distributed real-time systems. In *IEEE ISORC*, pages 344–353, 2007.

[18] J. Sun. *Fixed Priority Scheduling to Meet End-to-End Deadlines in Distributed Real-Time Systems*. PhD thesis, CS Dept., UIUC, 1997.

[19] Y. Zhang, B. Thrall, et al. A real-time performance comparison of distributable threads and event channels. In *RTAS*, pages 497–506, 2005.