

PTIDES on Flexible Task Graph: Real-Time Embedded System Building from Theory to Practice *

Jia Zou

UC Berkeley, IBM Research
jjazou@eecs.berkeley.edu

Joshua Auerbach

IBM Research
josh@us.ibm.com

David F. Bacon

IBM Research
dfb@watson.ibm.com

Edward A. Lee

UC Berkeley
eal@eecs.berkeley.edu

Abstract

The Flexotask system claims to enable implementation of both real-time applications and real-time schedulers in a Java Virtual Machine using an actors-like model. The PTIDES model is an actors-like model that claims to deliver precise control over end-to-end latencies in a complex real-time system. The present work jointly investigates both claims by (1) implementing several PTIDES-based schedulers as Flexotask scheduler plugins, and (2) using the resulting system to implement a new reactive control program for a simulation of the JAviator. We present results from the realistic JAviator control application and also from synthetic benchmarks designed to shed light on the differences between the several PTIDES schedulers we implemented.

Categories and Subject Descriptors D.0 [Computer Applications]: Computers In Other Systems—real time, industrial control

General Terms Algorithm, Design, Measurement

Keywords Flexible Task Graphs, PTIDES, jitter elimination, real-time scheduling, real-time systems

1. Introduction

This work explores the interaction of two possibly contradictory trends in cyber-physical systems. First, as such systems become more complex, the use of high level languages like Java to write them becomes more attractive. Second, the entire programming

stack at all levels of abstraction requires re-examination to ensure timing predictability and repeatability. Of course, the interposition of a Java VM in the programming stack might compromise the second goal.

Previous work falling within the first trend has led to the publication of the Flexible Task Graphs (or Flexotask) system [3], which unifies a number of previous efforts [21, 4, 22, 23]. Fortunately for our purposes, Flexotask is now available as an open-source offering [18]. Flexotask provides two interesting capabilities. First, it allows a real-time application to be written in a restricted subset of Java and executed in a Java virtual machine with high timing precision. Second, Flexotask provides a plugin interface to add new schedulers and the associated time annotations that those schedulers need to do their work.

A noteworthy example of the second trend is provided by the PTIDES model [29]. The focus of PTIDES is on distributed systems, although the insights can be applied to any complex system with bounded sources of timing variability. PTIDES makes time semantics an integral part of the system scheduler by basing itself on the discrete-event (DE) [13] model of computation. PTIDES enables the scheduling of output events in terms of the observed time of input events with provable bounds on variability. This requires only that there be bounds on various sources of variability such as communication delay, clock drift, and computation time.

To evaluate how a model aimed at increasing predictability and repeatability interacts with a system for real-time Java programming, we conducted a case study in which we first implemented PTIDES as a plugin for Flexotask and then evaluated the resulting system on a realistic application. In implementing PTIDES as a Flexotask plugin, we were guided by the analysis in [9]. That work provides a general execution strategy encompassing all feasible execution policies, but leaves numerous practical details open. Consequently, we implemented not just one but three PTIDES schedulers that conform to the general execution strategy but that have different pragmatic design points.

The ideal application for our purposes would be one that benefited from the PTIDES model and had some previous history. The JAviator application [25] has been used to evaluate Exotasks [4], which is a precursor to the Flexotask model. Previous controllers written for the JAviator focused on closed loop feedback between sensors and motors. Time-based controllers are adequate for this problem because the information from the sensors arrives at regular intervals. However, there is a second aspect to JAviator control, involving a human operator moving a joystick at a remote machine

* This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, and Toyota.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCES'09, June 19–20, 2009, Dublin, Ireland.

Copyright © 2009 ACM 978-1-60558-356-3/09/06...\$5.00

and observing the JAviator’s responses. Variations in the end-to-end delay of this process (due to imperfect communication) was not subject to any regulation in earlier JAviator software, and existing controls cannot dampen such variability because the arrival pattern is irregular.

A PTIDES-based control was written with the goal of reducing the variability in the time between when the human pilot moves a joystick at the operator console and the time when the JAviator notices the new information and responds. We do not claim that regulation of this variability is necessarily important to the flight experience; indeed, the observed variability with existing controls was in the tens of milliseconds, probably within human tolerance. However, existing literature [28, 20] establishes that all forms of jitter in real-time systems are potentially problematic, and the ability to control end-to-end jitter in the responses to non-periodic events is therefore an important addition to the overall repertoire of techniques. The joystick/response sequence in the JAviator application demonstrates exactly that.

Since JAviator hardware is hard to come by, we used a simulation in this study. Only the physical JAviator itself was simulated. The control program and the human operator console operated in real-time, and the simulation’s responses were delivered in real-time to make the evaluation realistic.

Since our three differing realizations of the PTIDES model as Flexotask schedulers had implications that were not likely to be visible in the JAviator control, we also wrote some artificial benchmarks to showcase these differences.

In the next section we briefly summarize related work in real-time scheduling. We follow by summarizing PTIDES semantics, its uses and its implications for this experiment. In the subsequent section, we describe how we implemented our PTIDES schedulers for Flexotask, touching on characteristics of Flexotask, some of the limitations we found, and how these were overcome. Next, we describe and evaluate synthetic applications that were used to compare and contrast the PTIDES schedulers with each other. Then we describe the new JAviator control and compare its behavior using PTIDES and using two different non-PTIDES schedulers, one time-based, the other event-based.

2. Related Work

Timing problems in real-time control systems have been extensively studied [24, 15]. Elimination of jitter within a control system has been addressed mainly through two different approaches. Some recent research has tried to solve the jitter problem using specific scheduling-based solutions [6, 1], while others have stated it is impossible to eliminate jitter, and have tried to compensate for it at runtime in the controller design [14]. In the latter case, the overhead associated with the compensation may be too big when the system runs on embedded platforms. Also, we believe that by building a jitter-eliminating scheduler on a real-time supported platform, jitter can be reduced until the system control is no longer affected. Thus, we complement the works such as [6, 1] by adding timed semantics to the system model, and use this addition to reduce the jitter as observed in the environment.

We are not the first to incorporate timed semantics in system models. For example, Giotto does it by introducing “logical execution time” or LET [10]. Both LET and PTIDES limit the places where non-physical time is related to physical time, however, PTIDES takes an event triggered approach instead of only focusing on periodic tasks.

PTIDES scheduling semantics differ from schemes such as those described in [16] in that scheduling decisions are made regardless of the properties of the execution platform (although such properties may be considered in judging feasibility). In [16], the decision to choose the next event to execute depends on the re-

maining execution time, which depends on the platform it runs on. This causes the execution of the model to vary across different computation platforms. On the contrary, PTIDES will always make the same decisions regarding which event to process, decoupling the decision from the choice of platform.

Finally, in Sec. 3, we introduce a scheduler that is a composition of a PTIDES scheduler and earliest-deadline-first(EDF) scheduler. This work differs from hierarchical real-time scheduling as described in [8]. Work such as [8] addresses the problem of running multiple real-time applications on a shared computation platform. In hierarchical real-time scheduling, there usually exists a top level scheduler in the OS kernel which distributes computation resources to different real-time applications, and each application is scheduled separately, allowing different real-time scheduling schemes for different applications. The PTIDES EDF scheduler only addresses the scheduling of a single application but combines concepts from two different scheduling schemes.

3. PTIDES

3.1 Review of PTIDES Model

PTIDES, or Programming Temporally Integrated Distributed Embedded System, is a programming model based on the discrete-event (DE) model of computation. DE is used for hardware simulation in languages such as VHDL, for network simulation, and for complex system simulation [27]. It has the advantages of simplicity, time-awareness, and determinism. PTIDES emphasizes data and timing determinacy, meaning that the same set of time-stamped inputs results in the same set of time-stamped outputs [29, 9].

Within a PTIDES model, just as in DE, each event can be thought of as a pair of data value and timestamp. DE specifies that each actor in the system must *process* events in timestamp order. As a simplifying assumption in the present work, we also assume that the actors in this framework *produce* events in timestamp order, although this is not a necessary assumption of PTIDES semantics.

Unlike in DE, where a simulation runs entirely in “model-time”, a PTIDES scheduler relates model time to real time at points where inputs are consumed from the environment or outputs must be produced to the environment. While model time is abstracted by the timestamp of each event executed by the scheduler, real time is the time in the physical world. Thus while real time has a total order, model time does not have to observe that total order. The scheduler may schedule an event of bigger timestamp before processing one of smaller timestamp, as long as no actor observes any violation of causality. The model time of an actor always has to be causal, but this imposes no ordering on events at actors that are not connected to each other.

To relate model time to real time, *sensors* and *actuators* as defined in [9] are used. A sensor senses information from the environment. When data is received, the sensor time-stamps the data with the current real time, then releases the event to the system. Thus, at a sensor, real time and model time are bounded by the relationship $t > \tau$. In English, this means the real time t when an event is released by a sensor is always greater than the timestamp τ assigned to the event. PTIDES also requires a real time delay upper bound d_o for each sensor. This statically determined value is used to specify another relationship at the output port of the sensor, namely $t < \tau + d_o$. Intuitively, this means an event of time stamp τ will be sent to the system at a real time no later than $\tau + d_o$.

Actuators, on the other hand, act upon time-stamped events at a real time exactly equal to the timestamp. Thus PTIDES imposes the requirement $t < \tau$ at the input ports of actuators, where the timestamp of an event at the actuator can be interpreted as a deadline for event delivery. As Fig. 1 shows, since the output port of the sensor has the relationship $t > \tau$, and the input port of the

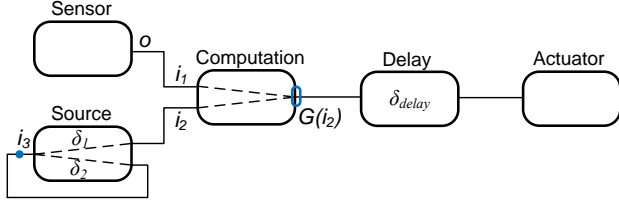


Figure 1. Simple PTIDES Example

actuator has the relationship $t < \tau$, the model would never be schedulable, unless some model time delay is specified for some actors. In the example of Fig. 1, a *Delay* actors is used. This actor consumes an event at its input, and produces another at its output consisting of the same event with the timestamp increased by the value δ_{delay} . All actors in PTIDES models are assumed to be causal, i.e., $\delta_{delay} \geq 0$.

Though these delays are part of the semantics of the model, when the model is run on a execution platform, it is important to determine whether the delays result in achievable deadlines at the actuators. This involves determination of worst-case-execution-time (WCET), issues of resource contention, communication delay, etc. We do not consider these issues in this study, but use what we judged to be reasonable heuristic values. Although it may be difficult to obtain tight bounds on the model time delay values, in realistic systems with reasonable amounts of slack, conservative estimates are possible that allow real time goals to be met.

Finally, PTIDES assumes a global notion of time when applied to distributed systems. This is usually achieved with precision time protocols such as NTP, IEEE1588, or the GPS system. With a bound on the clock error between different platforms, as well as a bound on inter-platform communication delay, the practice of using "null messages" [7] across computing platforms is no longer needed.

3.2 Schedulers that Implement the PTIDES Model

This section describes the assumptions that were made in implementing Flexotask schedulers based on PTIDES semantics, and contrasts the three different implementations. These are compared empirically in Section 5.

Ref. [30] defines the general semantics of PTIDES. The implementations described in this paper conform to the general semantics, but make some limiting assumptions about the system. First, we assume all network communications are one-to-one, and network packets are received in the order they are sent. Second, we assume that all actors produce events in increasing timestamp order. Together, these assumptions imply that all events in the system are sent and received in timestamp order. To complete the realization of this assumption, we introduce super-dense time, where the timestamp of an event also includes a microstep, in order to differentiate multiple events of the same timestamp. These assumptions, which are not requirements of PTIDES semantics, were adopted in the interest of providing more optimal scheduler implementations.

Since DE and PTIDES require actors to process events in timestamp order, the scheduler must determine when events can be safely processed before presenting them to actors for processing. Consider the example in Fig. 1. For an actor such as the *Delay* actor, which only has one input port, the answer is simple: since all actors produce events in timestamp order, any actor with only one input port will be able to process events as they appear on that input port. However the question is more complex for actors such as the *Computation* actor, which has multiple input ports. The scheduler could potentially wait for events to appear at both of its input ports before running it. However, in this case one of the inputs is connected to

a *sensor* actor, which only produces an event when something interesting happens in the environment. If nothing happens for a long time, then whatever inputs that are available from the *source* actor are blocked and cannot be safely processed. PTIDES solves this in [9, 30], by relating model time to real time at sensors and actuators, and using a global notion of time across different platforms to try to achieve better scheduling schemes as well as better utilization of the processor. The cited PTIDES work defined a *dependency cut*, or simply a *cut*, that defines which upstream actors the scheduler should examine to determine whether a given event e is safe to process. A scheduler implementation defines how the dependency cut is determined, and the cut allows an implementor to break up the safe-to-process analysis into two parts, as follows.

1. For all events outside of the cut, as [30] defines, the timestamp of the e is checked against the current real time of the system, as well as a statically determined offset value based on where the cut was made.
2. For all events residing inside of the cut, a dynamic check is made whether there can be an event of timestamp smaller than e in the future. The cut allows this check to be limited to local data structures, such as an event queue.

Since sharing event queue information across different computation platforms incurs communication expense, as [9] suggested, an intuitive cut is made at the boundary of the platform, such that each platform would only need to hold its own event queue.

We constructed three Flexotask schedulers that implement the PTIDES programming model. The schedulers differ in their implementation of the second part of the safe-to-process analysis, as well as the set of events that are examined in the analysis. The Flexotask-specific details about the implementation are discussed in the next section. Here we discuss key algorithm differences.

Our first scheduler (SimplePTIDES) is directly inspired by distributed DE simulators. A single event queue is used to sort all events in a given platform. The scheduler only examines the event of smallest timestamp for safe to process. Since we assume all actors to be causal, it follows that no events inside of the platform could render the event of smallest timestamp unsafe to process, so the scheduler merely waits for a point in real time at which no more events can come from outside the platform, and the event is safe to process at that instance. The advantage of such a scheme is that it is extremely simple and therefore can be implemented very efficiently. However, SimplePTIDES does not exploit parallelism either in the system model or the compute platform, since at one point of time, only one event is examined for safety to process.

The ParallelPTIDES scheduler resembles SimplePTIDES but avoids a key limitation: when only the event with the smallest timestamp is examined, other safe-to-process events may be "blocked" and overlooked. Instead, the ParallelPTIDES considers all events for safe-to-process. The ordered queue is just an optimization, since events with smaller timestamps are usually more likely to be safe to process. If the event of the smallest timestamp is safe-to-process, it is immediately processed. If not, the next event in the queue is analyzed. The ParallelPTIDES scheduler also augments the second part of the safe to process analysis by keeping track of model time at input ports of each actor. When an actor executes and produces a number of events, the model time frontiers on all reachable input ports downstream from this actor and within the same platform as this actor are updated. The model time is updated to the timestamp of current event plus the minimum model time delay between the output port of the actor that has produced the events and the input port where the model time resides. Now when an event is checked for safe to process, it only needs to check the model time of all input ports in the same actor, and if the model times are larger than the timestamp of the event of interest, then the

event is safe to process. These propagated model times are similar to “null messages” described in [7]. However, ParallelPTIDES only uses these messages within a platform. The use of these in-platform null messages is possible because all actors are required to not only process, but also produce events in timestamp order. Our implementation of ParallelPTIDES exploits parallelism in the system model, but does not yet exploit hardware parallelism, though its design enables such exploitation.

The third scheduler (EDFPTIDES) combines PTIDES semantics with previous work in real-time scheduling methods. Note that, at a particular instant, there might be multiple events in the event queue that are safe to process. Since the ParallelPTIDES scheduler uses an event queue sorted by the timestamps of the events, it always selects the event of smallest timestamp to process first among those that are safe. This, however, may not be optimal with respect to schedulability. In addition to PTIDES semantics, it is possible to consider other factors in the scheduling, imposing a “priority” (of whatever kind needed) that is different from timestamp order. This inspired the implementation of the EDFPTIDES scheduler, based on previous work on earliest-deadline-first (EDF) scheduling [5]. In this case the “priority” is the deadline of each event. The deadline of an event is generated by summing the timestamp of the event and the minimum model time delay from the current position of the event to its “nearest” actuator, where the smaller the delay, the “nearer” it is. EDFPTIDES uses the same logic as ParallelPTIDES to check whether an event is safe to process, only its event queue is ordered by deadline. If more than one event exists with the same deadline, these events are then ordered by the timestamps. An alternative way to understand the EDFPTIDES scheduler is that PTIDES semantics dictate a set of events that are safe to process, while EDF semantics guides the choice from among those events, dictating the one with the earliest deadline. Note even though EDF is proved to be optimal, we have not yet proved that the EDFPTIDES scheduler is optimal.

4. Flexotask

As described in [3], Flexible Task Graphs unify a number of earlier models for doing real-time programming in Java that its developers call “restricted thread programming models.” These models impose Java language restrictions that are checked by the system. Threads that will provably execute only the restricted code are then exempted from interference from garbage collection, a major source of non-determinism in Java. Like its precursor Extotask [4] and StreamFlex [23] models, Flexotask organizes the restricted computation as a graph of actors. Like the precursor Extotask system, a Flexotask system has pluggable scheduling. The GC-exempt threads are provided by the system, but exactly how they are used to execute individual actors is up to the scheduler.

Rehashing the Flexotask model is beyond the scope of this paper but for understanding we summarize some critical aspects here.

1. Each actor (or “task” or Flexotask) is written in a restricted subset of Java (most notably restricting how static fields may be used). It has an execute method that does some computation, during which it reads inputs from input ports and writes outputs to output ports.
2. Each actor has a private heap memory area.
3. The output ports of each actor are connected to the input ports of other actors. There are few constraints on the topology, but schedulers may impose their own constraints. Connections are not buffered, and the values sent across them are Java objects. Flexotask offers both deep-copy semantics and by-reference semantics on connections, but the latter imposes constraints on



Figure 2. Programming Stack

where objects can reside. We only used the deep-copy semantics in our experiments.

4. Pluggable schedulers have great flexibility but they must observe some restrictions unique to them.
 - (a) They must preallocate all data structures on the private scheduler heap, and they may not do further allocations while running tasks. This ensures the scheduler heaps will never need to be garbage collected.
 - (b) They have limited access to the private memory areas of actors. They cannot copy objects found there freely but can only cause them to be copied by executing methods on the connections.

Fig. 2 shows the programming stack of an application implemented on the Flexotask framework using a PTIDES scheduler. The many layers of abstraction between the hardware and the application provide many possible sources of non-determinism, and the purpose of this study is to see whether we can nevertheless solve real-time problems with bounds on variability. The kernel in this case is the real-time version of Linux, the Java VM is IBM’s WebSphere Real Time product [11], and the Flexotask system further constrains unwanted variation by eliminating the effects of garbage collection entirely (even the relatively low-impact collector in WebSphere Real Time). The Flexotask system’s pluggable scheduling capability was used to implement our PTIDES schedulers.

4.1 PTIDES Scheduler Implementations on Flexotask

A graphical presentation of the various memory spaces in a Flexotask system and their interrelationships is shown in Fig. 3. We use this picture to help explain how schedulers are implemented in Flexotask.

As shown, each actor has its own private heap as well as the scheduler, in addition to the public heap created when Java starts. However, these actors and their heaps must be initialized before the real-time Java application starts running, which implies the scheduler may not dynamically create actors while processing. Also, the scheduler (or Runnable as shown in Fig. 3) may allocate multiple threads to run itself, however this allocation, along with allocation of scheduler data structures must also be done before the graph starts running. Note restrictions on memory space usage is common in programming real-time systems. Dynamic allocation of memory adds complexity to the analysis of bounding system execution time, thus we consider this restriction to be reasonable. The actors themselves are allowed to allocate, and their heaps are garbage collected only at the scheduler’s request. The scheduler ensures that the actor is not running while being collected.

The “connection drivers” in the scheduler heap are used to interconnect the actors. each connection driver has one source (the output port of an actor) and one sink (the input port of an actor).

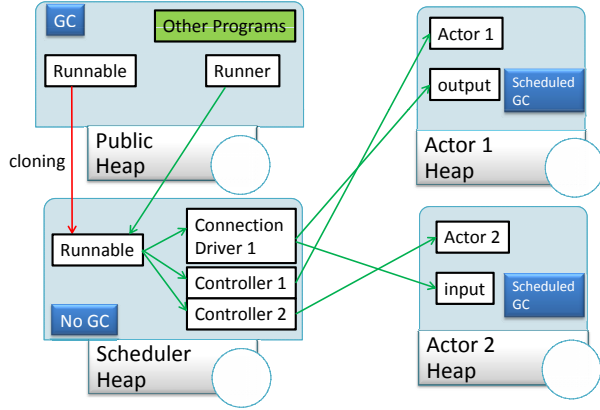


Figure 3. Flexotask Memory Space

The Flexotask model permits both multiple outgoing connections from an output port and multiple incoming connections to an input port. For our PTIDES implementations we assumed the former were allowed but we prohibited multiple connections terminating on the same input port. With the exception of sensors, an actor can only be executed (or “fired”) when there is at least one event at one of its inputs. But an executing actor is not obligated to produce events on any of its outputs, much less on all of them.

We now highlight two design points in implementing PTIDES schedulers on Flexotask. One is associated with how events are stored and referenced by the scheduler, while the other discusses the need for multiple scheduling threads.

4.2 Event Storage and Reference

Since PTIDES require events to be held until they are safe to process, each connection may logically hold more than one event at a time, i.e., buffering is required between tasks. This was an area characterized as “future work” in the published Flexotask paper, and it turns out that the Flexotask system as provided in open source [18] solves this problem differently than the paper predicted. There are no buffer tasks as described in the paper. Rather, buffering is an optional property of both input ports and output ports, and the connection has a destructive move operation and a non-destructive copy operation, both available to the scheduler. While the PTIDES scheduler makes use of this facility, it is also necessary for it to peek at values on port queues in order to observe the timestamps recorded in the events. A small modification of the Flexotask base code was needed to support this. In the case of PTIDES schedulers, buffering is required only on output port buffers, as will be explained. Finally, to make sure the scheduler behavior is application-independent, PTIDES programming on a Flexotask base requires that every event sent between tasks must inherit from a `TimedData` class (basically a timestamp-value pair as called for by the DE model). The PTIDES schedulers only examines the timestamp stored in `TimedData`, but does not consider the data value or the application purpose associated with it.

The inability of the scheduler to freely move objects across different memory area boundaries, essential for the Flexotask model’s correct operation, requires all data to be left at the output port of the task that created it. It must remain until all tasks immediately downstream that will need this data have executed. The scheduler transfers data to a task’s input port immediately before executing the task.

However, to ensure the scheduler can keep track of where the available data resides, creativity is required in the scheduler’s representation of event queues. The scheduler can neither copy the ap-

plication’s `TimedData` to its own memory nor maintain a pointer from its memory to the `TimedData` object in application memory. The restriction against pointers is due to the fact that task memory areas are subject to a moving garbage collection. Thus, events in the event queue must consist of pairs each of which contains a timestamp and a connection driver, which consist of the source port where the events reside, and the sink port where the event needs to be sent. Before and after each task executes, the scheduler records the number of values on each of the task’s output ports. If this count increases due to the execution, that means data has been written, and a corresponding number of events is then created in the event queue with the appropriate timestamp. Before the execution of a task, to ensure that the correct data values are selected for processing, given the event only gives the driver and not the specific data, we rely on the assumption all events are produced in timestamp order, and the fact that the buffer is in FIFO order. Thus when downstream tasks are ready to execute, the scheduler only needs to take out the first data at an output port and move it to the downstream input port, and it is guaranteed that this is the data linked to the particular event of interest.

Storing all events in the output port buffer, and only moving the event of smallest timestamp to downstream input ports also provides the guard against another situation: if multiple events of different timestamps are available for processing at an actor, PTIDES or DE semantics in general says we should only process the event of smallest timestamp, instead of processing all events within one firing of the actor. Thus only the event that is ready to process is moved from the output port buffer to the input buffer of the downstream actor, and the actor will only consume the event at its input when firing.

The limitation on the scheduler’s ability to allocate in its own heap is not very difficult to overcome in general. However, since the PTIDES scheduler make use of an event queue to manage all events in the system, the size of the event queue needs to be fixed at compile time. It is a rather difficult problem to bound the size of the queue, which is conceivably related to the schedulability analysis of the system, and the algorithm is not considered in this paper. At present we are simply making a highly conservative heuristic guess at initialization time, which works when resources are not very constrained.

4.3 Scheduler Threads and Synchronization

For all PTIDES schedulers, each of the sensors and actuators in the system needs to run on a separate thread to avoid blocking scheduler threads in I/O waits. This is made possible by Flexotask system allowing the scheduler to have multiple threads and to give those threads specialized roles. However, previous Flexotask schedulers were based on periodic execution and so the plugin interface provided to schedulers enabled clock-based waiting (or “sleeping”) but not the ability to react to external I/O events. Meaningful exploitation of PTIDES requires a reactive scheduler, rendering clock-based waiting inadequate. The problem was solved without making a fundamental change to the Flexotask system by adding a `WaitSet` utility that is of general use to other scheduler writers. The `WaitSet` allows generalized waiting by schedulers for events of interest, both the passage of time and the arrival of inputs. With this utility, sensors can communicate with the environment by listening on some file descriptor. After a sensor produces events to the system, the sensor thread notifies the main scheduling thread. If the main thread is executing, it would finish the current event execution, after which the main thread polls all sensor threads to see if any of them has updated its output ports. The `WaitSet` facility is not used for actuators, but a similar notification logic is employed; in that case, the actuator would consume its input(s) only when the main scheduler thread wakes up the actuator thread. All modifica-

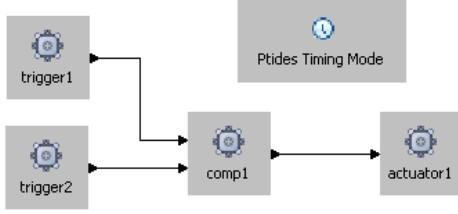


Figure 4. First PTIDES Example

tions of the Flexotask system, including the addition of the WaitSet have been or will be contributed to the open source effort.

To ensure actuation and trigger actions happen as soon as possible, we give all actuator threads the highest priority, the trigger threads the second highest priority, and the main scheduler thread has the lowest priority. It should be noted all the threads in Flexotask should have higher priority than any other thread in the system to avoid preemption of the real-time application.

In order to correctly synchronize the scheduler threads, monitor locks are used. A single monitor lock can be shared by all sensor threads, since when the main thread is waiting, it needs to wake up upon any action from any sensor thread. On the other hand, one lock is needed for *each* actuator thread, since when the main thread decides one actuator should execute, it needs to notify only that actuator. Monitor locks between scheduler threads are specially managed by the Flexotask system but schedulers are permitted to separately request the number of threads and the number of locks they need.

Finally, it should be noted that the EDFPTIDES scheduler cannot achieve true EDF semantics due to the Flexotask system's thread management being limited to what the underlying OS provides. The combination of Flexotask and RT Linux provides no way to programmatically preempt a running task due to a deadline recalculation. Priority manipulation is an inadequate mechanism, especially on a uniprocessor, since any thread the scheduler might dedicate to observing execution and manipulating priorities would itself not be guaranteed to run promptly and non-disruptively. Only the kernel can deliver the precise semantics required, and the RT Linux base on which the Flexotask system runs does not deliver them. The direct implication is that other than sensors or actuators, all of our other actors are fired within the main scheduler thread, and the upper bound of context switch time from executing one actor to another is essentially the longest worst-case-execution-time among all actors.

5. PTIDES Scheduler Comparisons

To compare our schedulers and jointly validate the Flexotask system's ability to support real time scheduling, we implemented two illustrative benchmarks. These benchmarks permit a comparison of the three PTIDES schedulers and were specifically designed to highlight their differences. We also implemented a control application for a simulation of the JAviator aircraft [25], which is used to compare the PTIDES scheduler with two non-PTIDES schedulers. We first present the comparison of the three PTIDES schedulers in this section. We present the control application implementation, as well as results obtained in the next two sections.

The results from this section were collected using a 8-way Intel machine (two quadcore CPUs) with 8GB of memory. The Flexotask framework requires a cooperating VM to achieve real-time behavior. We used IBM's WebSphere Real Time VM [11]. This VM, in turn, requires an RT Linux kernel. We used the RHEL5.0 kernel, version 2.6.24.7-65.el5rt.

5.1 Comparison between SimplePTIDES and ParallelPTIDES Schedulers

Our first benchmark closely resembles the example in Fig. 1. It is shown in Fig. 4. We use it to compare SimplePTIDES and ParallelPTIDES schedulers.

In this example, two trigger actors output events to the *comp1* actor downstream. The trigger actors each has a $d_{o1} = .5ms$ and $d_{o2} = 3ms$ associated with them. Recall from Sec. 3 d_o 's are the real time delay upper bounds. Both of the trigger actors have a model time delay of 0. Employing the analysis in Sec. 3 and [9], we see that events at the each input port of *comp1* will only become safe to process after real time exceeds $\tau + d_x$, where d_x is the d_o on the other input port of the *comp1*.

Now consider the case when two events e_1, e_2 arrive at inputs ports of *comp1* at close intervals. These events are of timestamps τ_1 and τ_2 , respectively. if $\tau_1 < \tau_2$ then the SimplePTIDES would process them at real time $\tau_1 + d_{x1}$, but if at the same time $\tau_1 + d_{x1} > \tau_2 + d_{x2}$, then the ParallelPTIDES scheduler would execute event no later than $\tau_2 + d_{x2}$. In addition, the ParallelPTIDES provides another mechanism to execute events at a earlier time than $\tau_2 + d_{x2}$. Recall when all inputs of an actor have events present, it is safe to process the event of the smallest timestamp, instead of waiting for a specific real time.

Comparing Fig. 5 and 6, this is exactly the behavior we observe. Fig. 5 shows the model run with the SimplePTIDES scheduler. The upper red (lower green) bars indicate the real time interval between when *trigger1* (*trigger2*) senses an event and when *actuator1* receives a corresponding event. Notice there is an important distinction between when actuators *receive* an event, and when it actuates. An actuator always actuates at real time equal to the timestamp of its input event, but may receive such a event at much earlier times. As we said, an event from *trigger1* would be safe to process at real time $\tau_1 + 3ms$, where $d_{x1} = d_{o2} = 3ms$, while an event from *trigger2* would be safe to process at real time $\tau_2 + .5ms$. From Fig. 5, we see time delays from *trigger1* to the output (red bars) is deterministically $3ms + \epsilon$, where $\epsilon \sim .12ms$ is the time to fire the *comp1* actor. On the other hand, we see the time delay from *trigger1* to the output (green bars) is around $.5ms + \epsilon$, except at some points it becomes even larger than that. The zoomed in picture shows these points happen when the *trigger1* and *trigger2* sense events at close intervals (indicated by the blue and yellow dots just above the bars). Particularly, when *trigger1* senses first, we are forced to wait until real time is $\tau_1 + 3ms$. But since the event from *trigger2* arrived just a bit later than *trigger1*, that event is stalled because only the event of smallest timestamp in the queue is checked for safe to process.

The results from the ParallelPTIDES scheduler are shown in Fig. 6. Here, we see that the green bars are deterministically $.5ms + \epsilon$, meaning that an event from *trigger2* is always safe to process at real time $\tau_2 + .5ms$. But the red bars are no longer deterministically $3ms + \epsilon$. In fact, at some points, it is much much smaller than that. These points again occur when the two trigger events occur very closely together, which is when both inputs have events available at the input of *comp1* at the same time, and in which case we can safely process the event of smaller timestamp without having to wait until some real time.

Using these figures we can also determine the scheduling overhead. The figures show the same application running on different schedulers. If the trigger events do *not* arrive close to each other, then the safety analysis of both schedulers will return the same result. The data collected shows an average delay of $3.145ms$ for the ParallelPTIDES scheduler, and $3.117ms$ for the SimplePTIDES scheduler. This means the difference in scheduling time between the two schedulers for a single event is $(3.145ms - 3.117ms)/2 = 14\mu s$. We divide by 2 because both the *computation1* and *actuator1* actors are fired within one iteration. Depending on the application,

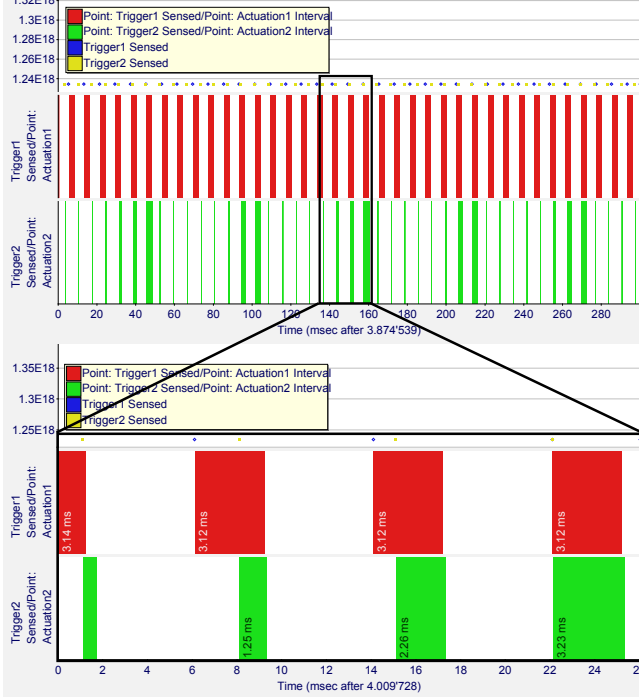


Figure 5. Execution Trace of the SimplePTIDES Scheduler for Example 1.

$14\mu s$ of overhead may or may not be small enough, but for our purposes, it is negligible. Note however we are running on workstations with powerful processing cores. If the scheduler is deployed on some embedded system, generally not as much computational power is available, and would result in a larger overhead.

5.2 Comparison between ParallelPTIDES and EDFPTIDES Schedulers

The second benchmark is shown in Fig. 7. It is used to compare the ParallelPTIDES and the EDFPTIDES schedulers. This example is aimed at showing the superiority of the EDF scheduler in choosing the event of earliest deadline to process over the Parallel scheduler, which chooses the event of smallest timestamp to process. In this example, two independent paths of actors process and produce events. Here, the *computation2* actor is much more computationally intensive than *computation1*, and the model time delay for *computation2* is set to $7ms$, and $1ms$ for *computation1*. Recall the deadline calculation from Sec. 3 would give us a deadline of $\tau_1 + 1ms$ for events produced by *trigger1*, and a deadline of $\tau_2 + 7ms$ for events produced by *trigger2*, where τ_1 and τ_2 are the timestamps of the events, respectively. Thus, if *trigger1* and *trigger2* were to produce events almost simultaneously, with *trigger2* producing its output first, we would have a case of $\tau_2 < \tau_1$. When the ParallelPTIDES scheduler is used, τ_2 is processed before τ_1 , which means the event produced by *trigger1* has to wait until *computation2* to finish firing before it can be processed. Since *computation2* is computationally intensive, this means when the event arrives at *actuator1*, it is very likely to have missed the deadline. When the EDFPTIDES scheduler is used however, we would pick the event of smaller deadline to process first. Since there are no event dependency between the event at *computation1* and the one at *computation2*, *computation1* and *actuator1* will be fired first. Given that the model time delay associated with *computation2* is much larger, both actuators should be able to actuate before the event deadlines expire.

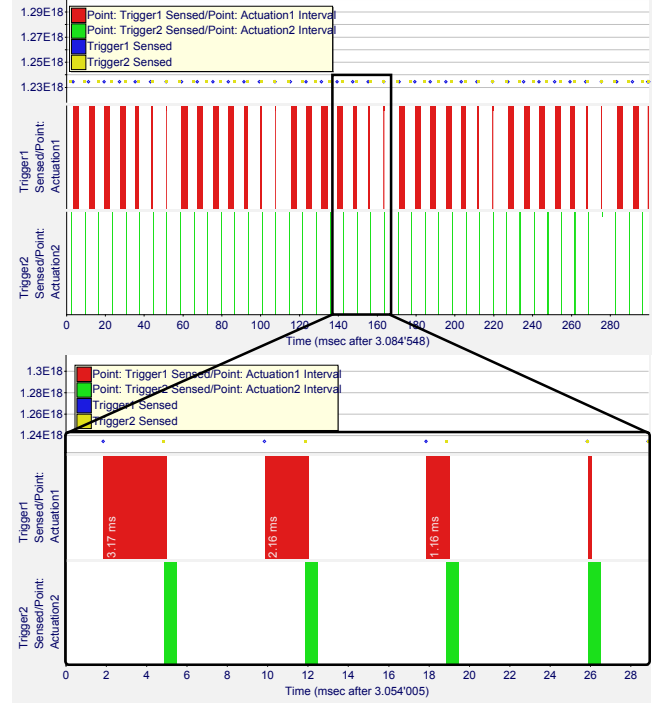


Figure 6. Execution Trace of the ParallelPTIDES Scheduler for Example 1.

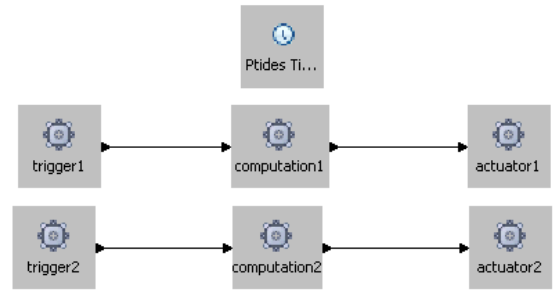


Figure 7. Second PTIDES Example

Comparing Fig. 8 and Fig. 9, this is exactly the behavior we observe. Both of these figures show the real time delay between the event sensed from *trigger1* and the actuation at *actuator1*, while using different schedulers. We display these delays in histograms. In Fig. 8, where the ParallelPTIDES scheduler is used, there are two peaks for delay. One centers around $1ms$, while the other around $2.7ms$. Recall since the model time delay for *computation1* is $1ms$, the delay should only be around $1ms$. The reason we would miss the deadline is explained above. Notice the value $2.7ms$ is merely a reflexion of how long it took for *computation2* and *computation1* to execute.

We see this is not the case in Fig. 9, where the delay is always around $1ms$. We note this example also depends on the scheduler implementation, where threads that run the actuator actors higher priority than the thread that runs the main scheduler. If the actuator thread was not able to preempt the main scheduler thread, we would still get a histogram like the one in Fig. 8, because the firing of *computation2* by the scheduler thread has to finish before the actuator thread can assert the actuation signal.

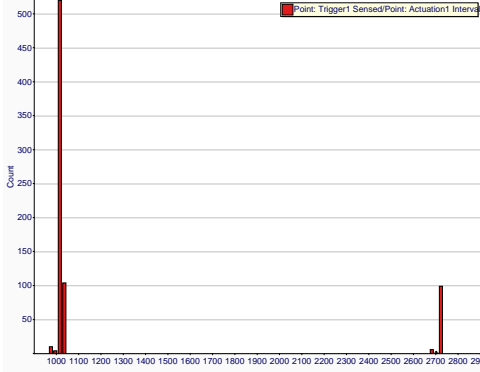


Figure 8. Distribution of End-To-End Delay (in milliseconds) for Example 2 with ParallelPTIDES Scheduler.

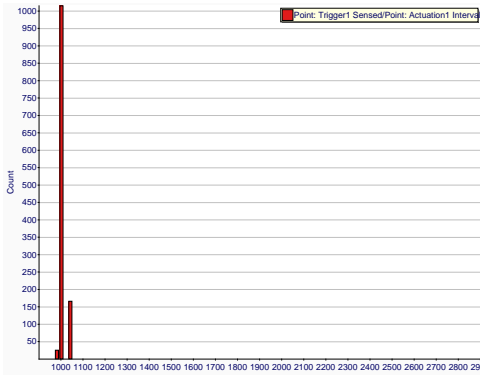


Figure 9. Distribution of End-To-End Delay (in milliseconds) for Example 2 with EDFPTIDES Scheduler.

6. Application

The results of the previous section suggest that the EDFPTIDES scheduler is generally superior and the added complexity is not a serious factor, at least for our chosen hardware. Therefore, we use the EDFPTIDES scheduler in this section to implement the JAviator control, and we will compare it with non-PTIDES schedulers on the same application.

Ideally, we would have collected measurements from the real JAviator, measuring actual human experience or objective end-to-end time delays from the standpoint of an external observer. However, real JAviator hardware is hard to come by and was not available to us during the time of the study. This caused us to employ a simulation of the JAviator, which interacted with the control program via TCP sockets instead of the RS-232 links that exist within the real JAviator. Creating enough realism in this simulation to allow us to place human subjects in the loop was beyond our means, hence, the joystick movements were scripted and the measurements were done within the computer systems by a technique to be described.

The actor graph of the JAviator application is shown in Fig 10. The joystick process runs on a different machine than the control program and is not a Flexotask program. However, it generates PTIDES timestamps, which assign model times to every joystick event. The joystick process polls the joystick every 10 ms but only sends a packet when the value differs from the previous one.

The remaining actors run in a Flexotask graph on a single machine. The *Joystick Receiver* actor receives joystick events. The

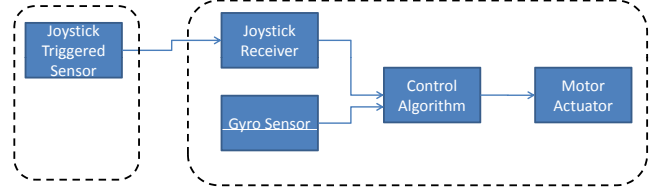


Figure 10. JAviator Control Model

Gyro Sensor actor reads attitude information (roll, pitch and yaw). The *Control Algorithm* is a PID controller which uses the joystick input as a target and the attitude information from the gyro sensor to calculate appropriate force values for each of the four motors.

In the real application, communication with the JAviator is via wireless, which, depending on local conditions, can introduce communication jitter. We know that effective clock synchronization techniques exist that work across wireless [26] and can achieve precisions in the microseconds, and we estimate the practical communications jitter to be in the milliseconds. However, for reasons purely related to the real-time Linux and Java frameworks we used, we were unable to test using wireless communication and so the tests were done with LAN connected machines. The switched ethernet connection introduced almost no jitter, and so we simulated that jitter by introducing a gamma-distributed delay roughly approximating what we expect to see in the field [2]. Both the mean delay and its standard deviation are 1ms, but a gamma distribution has no absolute maximum. However, since the application discards packets that are more than 4ms old, 4ms becomes the practical maximum. As previously noted, this jitter is not likely to really bother a human operator. The purpose here is to establish the ability of PTIDES to tightly control end-to-end latencies, and we do not claim that the JAviator control that we developed is truly superior to existing ones.

To coordinate time across the two machines used for the experiment, an open source implementation of the IEEE 1588 standard [12] for Linux [19] was employed. We encountered some problems with the `adjtimex` system call employed by the software when running on the kernel and hardware we employed, and so the software was modified to use `settimeofday`. The uncorrected drift between the clocks on the two machines was systematic and unidirectional at about $22\mu\text{s}$ per second. To avoid clock non-monotonicity, we ran the IEEE 1588 protocol such that the machine with the faster clock was always the master. With the correction running, the maximum drift during the experiment was well under $30\mu\text{s}$, since the correction occurred once a second. To minimize perturbation due to the clock synchronization daemon itself, we executed the JAviator control on the master machine, the joystick program on the slave machine.

As previously mentioned, all of our PTIDES schedulers dedicate a thread to each sensor, and the *Joystick Receiver* functions as a sensor for this purpose. But, since the purpose is to employ PTIDES on an end-to-end distributed basis, the incoming event does not have a new model time assigned by this sensor but retains its original model time assigned in the joystick process. In contrast, the *Gyro Sensor* is a true sensor, which reads data coming from the JAviator simulator. Upon receiving data from the simulator, it timestamps the data with the current real time, then does some simple scaling of that data so that it is more meaningful to the control algorithm. Both the *Gyro Sensor* and the *Joystick Receiver* feed data to the *Control Algorithm* actor, which calculates the speed of each of the four motors and forwards the result to the *Motor Actuator* actor, which actuates the simulated JAviator. The

Control Algorithm will calculate new motor speeds and forward a new motor packet if it has *either* new joystick data or new gyro data.

The simulated Javiator sends new gyro data every 8ms, which is the maximum data rate of the gyro on the actual JAViator. Existing JAViator controls use periodic rates between 8ms and 20ms. This frequent arrival of new gyro data ensures stability. The much slower arrival of new joystick data is, in a sense, less critical, but in these experiments we were interested in controlling the end-to-end responsiveness of re-calculations stimulated by the arrival of human-originated data.

For purposes of evaluation, we used TuningFork [17] to record events in the *Joystick Receiver* and in the *Motor Actuator*. Recording in the *Joystick Receiver* enabled a measurement of the actual delay between the joystick process and the *Joystick Receiver*, obtained by subtracting the model timestamp in the event (which was the time at origination) from the real time of receipt. The events in the *Motor Actuator* distinguished whether the event included a new joystick receipt. In our evaluation we used these events only to measure end-to-end delay by subtracting the model time recorded in the *Joystick Receiver* from the real time of actuation recorded in the *Motor Actuator*.

Now as PTIDES requires, and similar to the model in Fig. 1, a model delay needed to be added to keep it schedulable. Thus the *Control Algorithm* actor is parameterized to produce a model time delay of $\delta = 7ms$. Experimentally, we determined that 7ms was enough as an upper bound to ensure events will arrive at the *Motor Actuator* before its deadline. Also the receiver and the sensor also need to have a real time delay bound d_o associated with them as PTIDES requires. The *Gyro Sensor*'s delay is relatively easy to analyze, which is basically the maximum real time delay of context switching to the thread that runs this actor, plus the time to decode and produce the event, assuming the event queue is not being accessed by any other thread. Though analytically simple, this value is difficult to determine due to the number of software layers present. Thus we over-estimate this value to be 1ms. The same estimation is done for the *Joystick Receiver*. However, in addition to the real time delays for a sensor, it also needs to take into account of the communication delay as well as the clock error between the two platforms. These delays can be found through extensive testing of the communication system, and we over-estimate this value to be 6ms.

We compared the behavior of the EDFPTIDES scheduler against two alternatives. One was the time-triggered (TT) scheduler provided with the Flexotask open source implementation. The TT scheduler, as the name suggests, is time triggered, instead of event-triggered as in the case of the PTIDES schedulers, and is thus capable of supporting only periodic behavior. Because of this, a comparison between PTIDES and TT is not entirely illuminating, since it confounds the general issue of reactive versus time-based scheduling with the unique contributions of PTIDES. Consequently, we created a third (REACT) scheduler to compare with. REACT simply forwards information from either sensor as soon as it is received, and schedules the remaining actors on a data-dependent basis. Our basic observation is that, in the absence of model-based reasoning as in the PTIDES model, the scheduler has no good basis for deciding when to do the actuation, other than "as soon as possible" (as in REACT) or at a fixed time (as in TT). Thus, the two alternatives more or less bracket the possibilities. The results from comparing these three schedulers are shown in the next section.

7. Performance and Results

The results from this section were collected on two machines similar to that of Section 5. We used 4-way AMD machines (two dual

	Count	Min	Max	Mean	Std.Dev
EDFPTIDES	1721	.135	4.07	0.95	0.744
REACT	2357	.134	4.09	0.94	0.741
TT	1684	0.18	11.62	4.91	2.435

Figure 11. Receipt Delays (in millisecs from origination of joystick event to Joystick Receiver Processing)

	Count	Min	Max	Mean	Std.Dev
EDFPTIDES	1721	7.01	7.04	7.01	0.002
REACT	2352	0.21	4.17	1.01	0.741
TT	1681	8.15	19.58	12.86	2.436

Figure 12. End-to-End Delays (in millisecs from origination of joystick event to Related Actuator Processing)

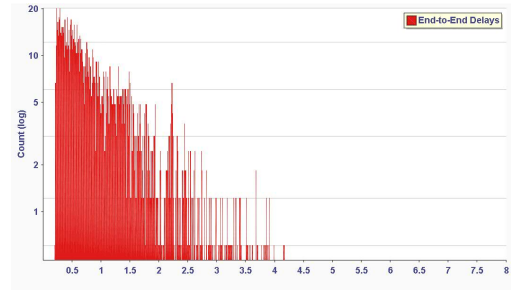


Figure 13. Distribution of End-to-End Delays when REACT is used. The X axis is delay in milliseconds. The Y axis is logarithmic

core CPUs) with 4GB of memory each. IBM's WebSphere Real Time VM [11] was used to run all programs. The joystick program and the JAViator simulator ran on one machine, the JAViator control program ran on the other. The operating system is version 2.6.24.7-95.el5rt of RedHat Linux 5.0 RT.

Fig. 11 are the differences between the time of joystick event origination and the processing of that event by the joystick receiver. Recall that the observed jitter was artificially induced, but was chosen to be realistic in terms of what might be expected in the field. It can be clearly seen that these delays are basically the same for EDFPTIDES and REACT but that TT induces an additional absolute delay plus additional jitter due to the reading of this information at a fixed time offset within an 8ms period.

Figure 12 present the differences between the time of joystick event origination and the actuation of the motors dependent on the information in the event. The EDFPTIDES controller, by virtue of using fixed model time delays, induces a longer average delay than REACT, but shrinks the variance to a negligible amount, whereas the variance for REACT tracks the original jitter quite closely. The TT scheduler has a longer average than either (due to the need to wait for nearly two periods in the worst case for data to be both read and processed) but this does not buy anything in terms of reduced jitter; indeed, its jitter is the worst of the three. A sense of the degree of reduction in variance by using PTIDES can also be obtained by comparing Figure 13, showing a histogram of end-to-end delays using the REACT scheduler, with Figure 14 which shows the corresponding information when EDFPTIDES was used.

These results clearly indicate that using model time to schedule output events can be a useful technique when minimizing end-to-end variance in reactive systems. Neither of the other schedulers had the necessary information to do as well.

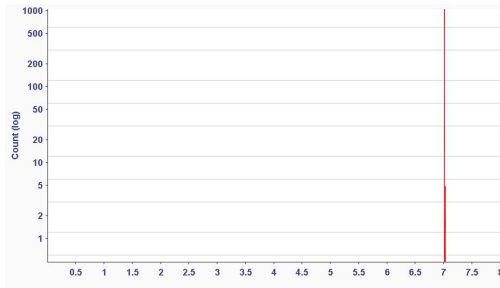


Figure 14. Distribution of End-to-End Delays when EDFPTIDES is used. The X axis is delay in millisecs. The Y axis is logarithmic

8. Summary

We have implemented and evaluated three PTIDES schedulers using the Flexotask framework. The results show that the Flexotask framework is a suitable environment for achieving the goals of PTIDES and that the PTIDES model is amenable to practical implementation on this platform. We demonstrated that the intended strong point of PTIDES (the ability to minimize variation in end-to-end delays in a reactive system) holds in a realistic setting. The work establishes that the use of an EDF concept in conjunction with PTIDES provides good scheduling characteristics without adding excessive overhead compared to the simplest implementation. Compared to both time-based and purely reactive scheduling, PTIDES provides the most precise control over variance in end-to-end delay.

Some issues remain as future work. A better exploitation of hardware parallelism should be possible. A strong demonstration of real application utility requires an application in which the end-to-end delay variance is a more serious obstacle to correct behavior than it is in the JAviator application. In the future, we may explore applications of PTIDES in computer music or other domains where variance in end-to-end delay has a more serious impact.

References

- [1] P. Albertos, A. Crespo, I. Ripoll, M. Valles, and P. Balbastre. RT control scheduling to reduce control performance degrading. In *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*, volume 5, 2000.
- [2] N. Ali, E. Ekram, A. Eljasmy, and K. Shuaib. Measured delay distribution in a wireless mesh network test-bed. In *AICCSA 2008*, pages 236 – 240.
- [3] J. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek. Flexible task graphs: a unified restricted thread programming model for java. *SIGPLAN Not.*, 43(7):1–11, 2008.
- [4] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer. Java takes flight: time-portable real-time programming with exotasks. In *Proceedings of LCTES '07*, pages 51–62, New York, NY, USA, 2007. ACM Press.
- [5] G. Buttazzo and J. Stankovic. Red: A Robust Earliest Deadline Scheduling Algorithm. In *Proceedings of Third International Workshop on Responsive Computing Systems*, 1993.
- [6] A. Cervin. Improved scheduling of control tasks. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, volume 10. IEEE Computer Society Press, 1999.
- [7] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transaction on Software Engineering*, 5(5), 1979.
- [8] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 308–319, 1997.
- [9] T. H. Feng and E. A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *Proceedings of RTAS*, St. Louis, MO, USA, April 2008.
- [10] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, pages 166–184. Springer-Verlag, 2001.
- [11] IBM Corp. *WebSphere Real-Time User's Guide*, first edition, 2006.
- [12] IEEE. A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. <http://ieee1588.nist.gov/>.
- [13] E. A. Lee. Discrete event models: Getting the semantics right. In *WSC '06: Proceedings*, pages 1–1. Winter Simulation Conference, 2006.
- [14] P. Marti, J. Fuertes, G. Fohler, and K. Ramamritham. Jitter compensation for real-time control systems. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 39–48, 2001.
- [15] J. Nilsson. Real-Time Control Systems with Delays. *Lund, Sweden: Lund Institute of Technology*, 1998.
- [16] S. Oh and S. Yang. A Modified Least-Laxity-First scheduling algorithm for real-timetasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings*, pages 31–36, 1998.
- [17] Open Source. The TuningFork Visualization Platform. <http://tuningforkvp.sourceforge.net>.
- [18] Open Source. Flexible task graphs. <http://flexotask.sourceforge.net>, 2008.
- [19] Open Source. The ptp daemon. <http://ptpd.sourceforge.net>, 2008.
- [20] D. Seto, J. Lehoczy, L. Sha, and K. Shin. On task schedulability in real-time control systems. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 13–21, 1996.
- [21] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proc. PLDI*, pages 283–294, Ottawa, Ontario, Canada, 2006.
- [22] J. H. Spring, F. Pizlo, R. Guerraoui, and J. Vitek. Programming abstractions for highly responsive systems. In *Proc. VEE*, 2007.
- [23] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Oct. 2007.
- [24] M. Törngren. Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems. *Real-Time Systems*, 14(3):219–250, 1998.
- [25] University of Salzburg. The JAviator Project. <http://javiator.cs.uni-salzburg.at>, 2008.
- [26] H. Wang, L. Yip, D. Maniezzo, J. C. Chen, R. E. Hudson, J. Elson, and K. Yao. A wireless time-synchronized cots sensor platform part ii applications to beamforming. In *In Proceedings of IEEE CAS Workshop on Wireless Communications and Networking*, 2002.
- [27] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.
- [28] W. Zhang, M. Branicky, and S. Phillips. Stability of networked control systems. *Control Systems Magazine, IEEE*, 21(1):84–99, 2001.
- [29] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proceedings of RTAS 07*, pages 259–268, 2007.
- [30] J. Zou, S. Matic, E. A. Lee, T. H. Feng, and P. Derler. Execution strategies for ptides, a programming model for distributed embedded systems. In *to appear in RTAS*, 2009.