To appear in Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012)

PtidyOS: A Lightweight Microkernel for Ptides Real-Time Systems

Jia Zou, Slobodan Matic, Edward A. Lee
University of California, Berkeley
{jiazou, matic, eal}@eecs.berkeley.edu

***Abstract*—Ptides, a programming model for distributed real-time embedded systems, was proposed previously. In this work, we focus on a workflow that applies Ptides in a single-CPU environment using model-based design techniques. Our workflow starts with a programming environment where a real-time application is expressed as a Ptides model. The model captures both the functionality of the system and the desired timing of interactions with the environment. The Ptides simulator supports simulation of both of these aspects. Once the designer is satisfied with the design, a code generator can be used to glue together the application code with a real-time operating system called PtidyOS. To ensure the responsiveness of the real-time program, PtidyOS's scheduler combines Ptides semantics with the earliest-deadline-first policy. To minimize scheduling overhead associated with context switching, PtidyOS uses a single stack for event scheduling and execution, while still enabling event preemptions. We demonstrate the Ptides workflow through a motion control application. The automatically generated code running on PtidyOS is compared with a manual C implementation running on bare silicon. We discuss the tradeoffs in functionality and performance between these two implementations.**

I. Introduction

Most real-time software is structured either as threads with priorities or as tasks with periods or deadlines. Zhao et al. proposed an alternative programming model that they called Ptides [19] (programming temporally integrated distributed embedded systems) that structures real-time software as an interconnection of actors [12] communicating using timestamped events. Ptides leverages network time synchronization [10], [3] to provide a coherent global temporal semantics in distributed systems. Zou et al. give an execution strategy for Ptides and introduce feasibility analysis in [21]. Eidson et al. further describe Ptides in [2], which shows how Ptides supports modal behaviors and describes an application to power plant control. This work builds on the previously developed theory, and describes a workflow for distributed, real-time systems.

Ptides builds on a particular variant of a discrete-event (DE) model of computation (MoC), where software and hardware components called actors send timestamped events to one another [1], [5]. DE specifies that each actor should process events in timestamp order, and thus the order of event processing is independent of the physical times at which events are delivered to the actors. Whereas classically DE is used to construct *simulations*, in Ptides, a DE model is an executable specification. The objective is to compile (or code generate) this specification into a deployable implementation, following the principles of model-based design [11]. This paper presents a design flow that encompasses a simulator, a code generator, and an ultra-lightweight real-time operating system, as shown in Fig. 1. The simulator is built on the Ptolemy II framework [4].

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), #0931843 (ActionWebs), and #1035672 (CSR-CPS Ptides)), 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 Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

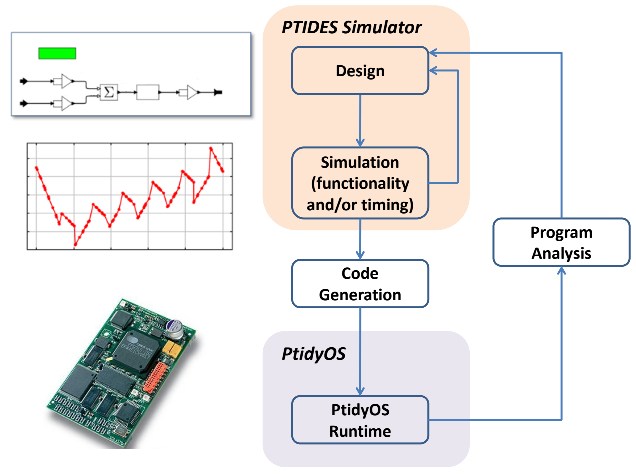

Fig. 1. Ptides Design Flow

By leveraging DE semantics, Ptides is well suited for systems with aperiodic events. Periodic scheduling schemes such as rate-monotonic have been widely adopted in the industry [16]. The research community has worked on extending these schemes to allow aperiodic tasks [18], [13], [17]. These approaches use a special purpose process called a "server" to schedule aperiodic tasks. These servers take a "slack-stealing" approach, where aperiodic tasks can "steal" as much processing power as possible, without causing periodic tasks to miss their deadlines. This approach is based on the assumption that periodic tasks have hard deadlines, while aperiodic tasks have soft or "firm" deadlines. Tasks with firm deadlines are those whose executions can be rejected by the scheduler, but the scheduler will meet deadlines for the accepted tasks, while tasks with hard deadlines are those whose failure to execute results in system failures. However, this assumption is not true in many real-world applications. For example, faults are inherently aperiodic, and they usually require hard real-time processing in order to guarantee safety of the system. In its current form, Ptides takes a purely event-triggered approach, and treats all periodic and aperiodic tasks equally as hard real-time tasks. Processing power is allocated purely based on events' priorities, where the priorities are inferred from the real-time specifications of the system.

An example of a simple Ptides model is shown in Fig. 2, where the model is rendered visually in Ptolemy II. This model counts discrete events, producing outputs that are displayed. It has two sensors, labeled EventSensor and ResetSensor. These sensors abstract the hardware devices that interact with the environment. When data is sensed, EventHandler

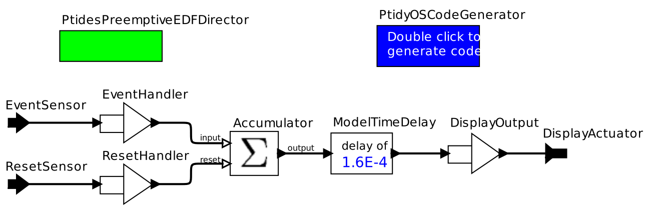

Fig. 2. Simple Ptides Example

and ResetHandler are triggered, and events are produced to the downstream Accumulator. The timestamp on such an event represents the time at which the sensor was triggered by the physical environment. The "handler" actors abstract interrupt handlers that process incoming events. When the Accumulator receives events from EventHandler, it increments its count and issues an output event with the accumulated count and timestamp equal to that of the input event. When the Accumulator actor receives an event from the ResetHandler, it resets its count and outputs an event with value zero and timestamp equal to that of the reset event. The output of the Accumulator goes into a ModelTimeDelay actor, which increments the timestamp by a fixed non-negative amount. The resulting event is delivered to the DisplayOutput, which schedules an actuation at the DisplayActuator. The timestamp of the actuation event is interpreted as a deadline for said delivery, and also as an indication of when in physical time the actuation should occur.

The workflow presented in this paper begins with construction of a model like that in Fig. 2 and provides a simulator that can be used to check functionality of the model. The model can then be elaborated with implementation details such as execution times of the components, enabling simulating execution of the model on an embedded platform. A code generator can then be used to translate the model into embedded code, where a runtime library called PtidyOS provides event handling, scheduling and memory management.

Notice the role of the ModelTimeDelay actor in Fig. 2. Since the timestamp of a sensor event represents the time at which the sensor is triggered by the physical environment, and the timestamp at the actuator is interpreted as a deadline and an indication of when in physical time the actuation should occur, the value of the ModelTimeDelay increment specifies the latency between the sensors and the actuator. Hence, the timing of the program is captured as a part of the Ptides model, and distributed real-time applications can be designed and simulated without the knowledge of execution times of software components. Designers can initially focus on specifying the functionality of the system and the timing of its interactions with the physical environment. A key principle in Ptides is that if execution times of software components are sufficiently small (the design is "feasible"), then the timing of interactions between Ptides software and its physical environment is independent of execution times. If execution times of software components are not sufficiently small (the design is "infeasible"), then deadlines specified by the Ptides model will not be met by the implementation.

As a design is elaborated, the designer will become concerned with whether the design is feasible. Our simulator supports modeling execution times of software components, enabling evaluation of feasibility of the model. This in turn allows the system designer to check for scenarios in which deadline misses occur.

Once the designer is satisfied with the design, our workflow supports target-specific code generation using a real-time operating system that we call PtidyOS. Like its namesake, TinyOS [7], PtidyOS is a C library against which the application code links to run on bare iron, rather than an operating system that supervises the execution of programs that come and go.

In order to ensure the responsiveness of the real-time program, the PtidyOS scheduler combines Ptides semantics with traditional scheduling methods, particularly earliest-deadline-first (EDF). To minimize scheduling overhead associated with context switching, PtidyOS performs all event processing in interrupt service routines, using only a single stack.

The remaining sections are organized as follows. In Sec. II, we review of basic concepts of the Ptides programming model. Sec. III defines a family of execution strategies for Ptides, and relates them to a set of simulator modules in Ptolemy II. Sec. IV then talks about the Ptides design flow, in particular about PtidyOS. Sec. V applies this design flow on a few applications including a physical setup called the tunnelling ball device. We conclude and provide pointers for future work in Sec. VI.

II. Time and Causality

A. Model Time

In DE and Ptides, actors communicate by sending events through the actor connections. An event is a pair of data value and timestamp. Here we assume that the timestamp is a real number, also called the model time of the event. Each actor may have a state associated with it. DE semantics requires that an actor state be accessed in timestamp (or model-time) order. For example, if the processing of an input event results in the reading and/or writing of the state of the actor, then that actor must process input events in timestamp order.

B. Causality Relationships Between Ports

An actor can manipulate the timestamp of an input event, with the constraint that this manipulation must be causal, i.e., the timestamp of an output event must be greater than or equal to that of the triggering input event. Here we review two model time delay functions formally defined in [21]. The first function is $\delta_0(i, o)$, where i and o are input and output port, respectively. δ_0 represents the *minimum* model time delay between this pair of ports. If these ports do not belong to the same actor, or if there is no causality relationship between these ports, then $\delta_0(i, o) = \infty$. Since all actors are assumed to be causal, $\delta_0(i, o)$ takes a minimum value of 0. The second delay function, called $\delta(p_1, p_2)$, takes a pair of arbitrary ports in the system. Intuitively, $\delta(p_1, p_2)$ is the minimum model time delay from p_1 to p_2.

Let I be the set of all input ports in the system, and O be the set of all output ports in the system. $i', i'' \in I$ belong to the same input port group $\mathbb{G}_i$ if and only if there exists an output port $o \in O$ such that $\delta_0(i', o) < \infty$ and $\delta_0(i'', o) < \infty$.

C. Sensor and Actuator Ports

In general, for an event in the Ptides model there is no relation between its model time and the physical time of its occurrence. However, the relations are introduced at points of interaction between the Ptides platform and the physical environment. We call these points *real-time ports*.

Sensor ports are inputs to a platform. When an event from the environment is detected, the sensor timestamps that data with the current physical time, and sends the event to the Ptides scheduler, where it is made visible to the rest of the platform. Thus, if the physical time at which sensing occurs is τ, and the physical time at which the event is made available to the Ptides scheduler is t, then $t \geq \tau$. Also if we assume an upper bound on the worst-case-response-time of a sensor actor, then

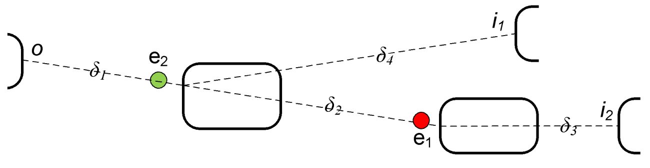

Fig. 3. Deadline Example

$t \leq \tau + d_o$, where $d_o \in \mathbb{R}^+$ is a parameter of the sensor input port called the *real-time delay*.

On the actuator side, we enforce the following rule: an event with timestamp τ must be delivered to the actuator at physical time t, where $t \leq \tau$. This constraint imposes a physical time deadline for each event delivered to an actuator, where the timestamp τ serves as the deadline.

III. Ptides Execution Strategies

In this section, a set of exeuction strategies based on the Ptides programming model is reviewed. These strategies are described in detail in [21]. The goal of these strategies is to ensure all input port groups consume events causally. All strategies take an event as argument, and return a boolean to indicate whether this event can be "safely" processed. An event is "safe to process" if the same input port group will not receive an event with a smaller timestamp at a later physical time. If an event e_2 could potentially render an event e_1 unsafe, then we say e_2 *causally affects* e_1. Take Fig. 3 for example, where e_2 causally affects e_1 if the processing of e_2 produces a future event e_2' such that e_2' and e_1 reside in the same input port group, and e_2' may have a timestamp that is less than or equal to that of e_1. In other words, e_2 *causally affects* e_1 (or e_1 is not "safe to process" if and only if there exists $i_x \in \mathbb{G}(i_1)$ such that $\tau(e_2) + \delta(i_2, i_x) \leq \tau(e_1)$, where i_2 and i_1 are the destination ports of e_2 and e_1, respectively; and $\tau(e)$ denotes the timestamps of event e.

A. Baseline Strategy

Before we give a formal definition of the baseline strategy, we define a *physical-time delay* function $d\colon I \to \mathbb{R}^+ \cup \{-\infty\}$ that maps each port $p \in I$ as follows:

$$d(p) = \begin{cases} d_o & \text{if } p \text{ is a sensor input port,} \\ -\infty & \text{otherwise.} \end{cases}$$

Let $\mathbb{I}_f$ denote all real-time sensor ports in the platform f. Recall that I is the set of all input ports and $\mathbb{G}_i$ denotes the input port group for an input port $i \in I$. A formal definition of the baseline strategy is as follows:

An event at platform f and input port $i \in I$ with timestamp τ is safe to process when:

1) *platform time (also called physical time) exceeds:*
$$\tau + \max_{p \in \mathbb{I}_f, i' \in \mathbb{G}_i} \{d(p) - \delta(p, i')\},$$
and
2) *for each port $p' \in I$ in platform f, each event at the input queue of p' has timestamp*
   a) *greater than $\tau + \max_{i' \in \mathbb{G}_i}\{-\delta(p', i')\}$ if $p' \notin \mathbb{G}_i$.*
   b) *greater than or equal to τ if $p' \in \mathbb{G}_i$.*

This strategy is described in the General Execution Strategy Section of [21]. Note the first part of the strategy can be enforced through a check of the timestamp of the event against the physical time subtracted by some offset which is calculated statically. We call this offset the *delay offset*. If this check is only performed on the event of smallest timestamp in a platform, then the second part of the *Baseline Strategy* can be omitted. We call this form of the *Baseline Strategy* the *Simple Strategy*.

B. Parallel Strategy

One way to avoid an unnecessary stall of event execution in the Simple Strategy is to not constrain the check to the earliest event in the queue. Here we present an approach that achieves this goal in addition to avoiding the second part of the *Baseline Strategy*.

Lemma 1. *If e_1 causally affects e_2, and the first part of the safe-to-process analysis as defined in the baseline strategy fails for e_1, then it must fail for e_2.*

Lemma 1 says that if e_1 causally affects e_2, then at all time instants of physical time, if there are potential events from outside of the platform that can causally affect e_2, then the same events will causally affect e_1; i.e., the first part of the strategy cannot be true for e_2 unless it is true for e_1. Armed with Lemma 1, we can scan the entire event queue, and a later event will not pass the physical time check unless all earlier events that causally affect it do so first. Proof for this lemma can be found in [20]. Based on Lemma 1, we give the operational semantics of the Parallel Strategy.

Step 1. Let e be the event of smallest timestamp in the event queue for platform f.

Step 2. Let i be the destination port of e, and let τ be its timestamp. If physical time exceeds
$$\tau + \max_{p \in \mathbb{I}_f, i' \in \mathbb{G}_i} \{d(p) - \delta(p, i')\},$$
then e is safe to process. Otherwise, let e be the event of next smallest timestamp in the queue, and repeat step 2.

Like the Simple Strategy, the Parallel Strategy is simplified to a simple check against physical time. Moreover, the Parallel Strategy exploits parallelism inherent in the model, and never wastes processor cycles as long as a safe event exists, while the Simple Strategy may block the execution of other events if the smallest one is not safe to process.

Moreover, if the event queue is ordered by timestamps, then the scheduler can simply step through the queue while performing Step 2. This results in an efficient implementation since it avoids analyzing all events in the event queue. Instead, as soon as a safe event is found, it is processed.

C. Deadline-Based Strategy

Both the Simple and Parallel strategies use event ordering by timestamp. In this section we explore other possible priorities. A natural choice is the earliest-deadline-first (EDF) scheme, where the scheduler selects to process the safe event of earliest deadline.

To understand how deadlines are defined, recall that events arriving at an actuator have deadlines associated with them, where the timestamp of the event is the deadline. Thus, for any event e that is separated by model time delay δ away from the nearest actuator, the deadline of e is $\tau(e)+\delta$. Take Fig. 3 as an example, where the deadline for e_1 is $\tau(e_1)+\delta_3$, the closest actuator port is i_2, and the delay between the destination port of e_1 and i_2 is δ_3. To aid the deadline calculation for events, we define a notion of relative deadline for each actor input port. This relative deadline function ($rd()$) captures the minimum model time delay between the input port and its reachable actuator ports. Let the function $port(e)$ denote the destination port of event e, then in the previous example, $rd(port(e_1)) = \delta_3$, $rd(port(e_2)) = \min\{\delta_1, \delta_2 + \delta_3\}$. The relative deadline for each input port can be calculated with a simple shortest path algorithm starting from each input port of actuators, and traversing backwards until the entire graph is visited. With the definition of relative deadlines, the absolute deadline of an event can be defined as $ad(e) = \tau(e) + rd(port(e))$. In this deadline-based strategy, the event queue is sorted by these absolute deadlines. When two events have the same deadlines, they are sorted by timestamps.

Recall that if e_2 causally affects e_1, e_1 will not pass the time check for safe-to-process at an earlier platform time than e_2. Moreover, e_2 is placed at an earlier position than e_1 in an event queue sorted by timestamp order. Finally, we state the following Lemma.

Lemma 2. *If e_2 causally affects e_1, then e_2 will be placed at an earlier position than e_1 in an event queue sorted by absolute deadlines.*

With Lemma 2, both the simple and parallel strategies can be used with a queue that is sorted by deadline order. Proof for this lemma can be found in [20].

D. Strategy Implementations In the Ptides Simulator

All of the above strategies are implemented as directors in Ptolemy II. A director is a software component that defines actor interactions within a model. The most basic of these is the *PtidesBasicDirector*. This director implements the *Simple Strategy* as explained at the end of Sec. III-A. However, preemption is disabled in this director, which means if an event is currently executing, and another event of smaller timestamp arrives at the platform, it will need to wait for the first event to finish before starting its own execution. Another interesting director is the *PtidesPreemptiveEDFDirector*, shown in Fig. 2. This director implements the deadline-based strategy as defined in Sec. III-C. The *PtidesPreemptiveUserEDFDirector* is similar to the *PtidesPreemptiveEDFDirector*, however it allows users to define arbitrary deadlines for system events. Finally, notice the *Parallel Strategy* as defined in Sec. III-B can be simulated using the *PtidesPreemptiveUserEDFDirector* by setting all relative deadlines to positive infinity. This results in all events having the same absolute deadlines, and the event queue will be sorted by timestamps.

These implementations in Ptolemy II allow for the design and simulation of distributed real-time systems. Once satisfied with the design, a code generator can then translate the model into a runtime called PtidyOS, the details of which is examined in the next section.

IV. PtidyOS

A. Related Work

First, we look at some of the similarities and differences between PtidyOS and conventional off-the-shelf RTOS.

The software architecture of the generated PtidyOS is unlike RTOS's such as QNX [6] or VxWorks [15], but instead, is more similar to TinyOS [14]. PtidyOS's microkernel includes a scheduler that performs interrupt handling and context switching and a primitive memory management unit. However, other conventional OS feature such as file system are not provided. When compiled, the PtidyOS kernel links against application code and produces one executable. Since applications are statically linked, dynamic instantiations of applications during runtime are not allowed.

PTIDES is particularly suited to systems where the order of distributed events must be determined to high accuracy and this order preserved in processing and response. To implement this, PtidyOS requires that actors with state process events in timestamp (model-time) order. This is an important difference from typical event-based frameworks, including TinyOS, that process events in the order they are received. In addition, PtidyOS is designed for systems with real-time requrements, and scheduling is mandated through event priorities. As explained in Sec. III-C, PtidyOS scheduler always processes the safe event of earliest deadline, and the processing of an event is preemptible by an occurrence of another safe event of earlier deadline. This is contrary to TinyOS, which uses FIFO scheduling and tasks are preemptible only by interrupts and not by other tasks.

PtidyOS is similar to QNX [6] in that it is a real-time microkernel that implements a message-passing scheme for inter-process communication. QNS provides *send()*, *receive()* and *reply()* primitives for explicity message passing. PtidyOS is based on actor-oriented programming paradigm, where events (messages) are transmitted between actors as defined by the actor graph. Both QNX and PtidyOS allows inter-process messages to be sorted by priorities. However priorties in PtidyOS are specific to each event, and are inferred through model-time based deadlines, while priorities in QNX are statically defined for each process by the user. Associating priorities to events instead of processes makes PtidyOS more applicable for event-triggered systems.

Also unlike traditional RTOS, PtidyOS steps away from thread-based approach, but instead uses a single stack for event execution. This eliminates the need for additional data structures to store outstanding system threads, and reduces context switching to simple *push()* and *pop()* operations. The Ptides formulation and the choice of model-time based deadlines allows for context switching to the earliest deadline event even in a single stacked environment. We elaborate on how this is achieved in Sec. IV-E3.

B. Design Requirements

First, we should note our current implementation of PtidyOS does not yet have networking and time synchronization components installed, and thus cannot be used in a distributed environment. We leave the distributed aspect as future work and focus on single and multi-core platforms where all computation cores use a common clock to reference physical time.

The primary goal of PtidyOS (or any RTOS) is to support the timely execution of applications. To achieve this goal, a deterministic scheduling scheme is required. Our goal for PtidyOS is to meet deadlines “optimally”, where optimality is defined with respect to feasibility. In other words, if there exists a scheduler that is able to meet deadlines for a model, the PtidyOS scheduler should be able to meet these deadlines. At the same time, the scheduler should not be overly complicated, in order to minimize and bound scheduling overhead.

Also depending on the application, hardware resources may be especially constrained. The particular microcontroller platform for our prototyping purposes only allows for implementations (PtidyOS + application) of size 32kB or smaller. Even if more powerful computation platforms are available, again to avoid delays in the operating system layer and minimize scheduling overhead, PtidyOS's application programming interfaces (API) methods should be as lightweight as possible. We discuss our approaches to meet these requirements.

C. Memory Management

As in many conventional RTOSs, to ensure real-time system behavior, and to meet the limited memory requirements, memory management in PtidyOS is kept as simple as possible. Specifically, application code is not allowed to dynamically allocate memory on a heap, but instead must allocate memory either on the stack or as static variables. Moreover, recall that actors in a Ptides model communicate through timestamped events. The memory for these events is allocated as static variables at compile time. The size of the allocated memory must correspond to the maximum number of events running in the system at any time. However, this number is difficult to determine. It depends on factors such as the rate of incoming events in the system, worst-case-execution-times, as well as scheduling decisions. We do not attempt to tackle this problem here. Instead, we throw a runtime exception if the pool of free events is exhausted. A linked list is used to maintain the pool of free events. This data structure is chosen because it supports event allocation and deallocation methods in $O(1)$ time.

D. Ptides Scheduler

The scheduler in PtidyOS implements the deadline based scheduler, as presented in Sec. III-C. Among all the presented schedulers, this scheduler is chosen because it is most applicable as a real-time scheduler. First, it integrates Ptides with EDF, thus allowing us to leverage EDF's optimality with respect to feasibility. Also, it performs the safe-to-process analysis for all events in the event queue; i.e., if there exists an event that is safe, then the CPU will not be idle.

Efficiency of queue accesses also plays an important role in minimizing overhead of the scheduling. For this reason, a doubly linked list is chosen to implement the event queue, and the following API methods are supported: peekNextEvent(Event*), addEvent(Event*), and removeEvent(Event*). addEvent() and removeEvent() simply insert (in deadline order) and remove events from the event queue, respectively. The peekNextEvent() returns the next event in the event queue without removing it. Both peekNextEvent() and removeEvent() take $O(1)$ time, while addEvent() takes $O(n)$ in the worst case, where n is the number of events in the system. A binary search algorithm is planned for addEvent() in future work, which will reduce the bound to $O(logn)$.

Finally, note that all event access functions are made atomic by disabling interrupts during the procedure in order to ensure correct concurrent behavior, which we discuss next.

E. Concurrency Management

Since PtidyOS is developed for systems with critical timing constraints, it is necessary that preemption is supported. In this subsection, we examine the implementation of this preemptive scheduler in PtidyOS. The preemptive scheduler includes two parts: interrupt handling, followed by event processing. We examine these separately.

1) Interrupt Handling: There are two kinds of interrupts that are of interest: safe-to-process interrupts and external interrupts. Safe-to-process interrupts are timer interrupts, which are set up to happen when some event(s) become safe to process. External interrupts on the other hand, are triggered through hardware peripherals such as GPIO or network devices. These interrupts could potentially post new events into the event queue. The stripped down implementation of the interrupt service routines is shown below:

```
// ISR for a sensor interrupt.
externalInterrupt() {
    fireActor(thisSensor);
    permitAllInterrupts();
    if (addedNewEvent()) {
        processEvents(1);
    }
}

// ISR for the safe to process timer interrupt
safeToProcessInterrupt() {
    permitAllInterrupts();
    processEvents(1);
}
```

Here fireAction() processes events for a particular actor, while processEvents() (elaborated in Sec. IV-E2) is the main scheduler routine that picks the safe event of earliest deadline to process. Note we trigger processEvents() at the end of each ISR in order to achieve preemptive behavior. In the case where a safe event of earlier deadline than the preempted event becomes available through the ISR, the the state of preempted event is saved, and the earlier deadline event is processed.

Also note these ISR's are almost identical. For a sensor ISR, since we have an actor that abstracts the sensor hardware device, that actor is fired during the ISR. On the other hand, since the safeToProcessInterrupt() is simply a timer interrupt to indicate an event has become safe to process, no actor is fired during this ISR. Next, we notify the rest of the system that new interrupts can be handled. This is the most important function that ensures the responsiveness of PtidyOS, where we assume the hardware supports a permitAllInterrupts() function, which allows all peripherals (including timer) to preempt the currently executing stack once it is called, even if those interrupts are of *lower* priority than the currently executing interrupt. Another way to interpret permitAllInterrupts() is that it tricks the system into thinking that there are no ISRs running in the system. This function is needed because we start the event processing routine next (in processEvents()) and we want to be able to handle all system interrupts during that routine.

Note in the PtidyOS implementation, the interrupt priorities are not important because these interrupts are only assumed to insert new events into the queue. Instead, the event deadlines dictate how computation resources are allocated. With the permitAllInterrupts() function, all interrupts can be given arbitrary priorities, as long as permitAllInterrupts() allows all other interrupts to preempt the currently executing one.

2) Event Processing: When new events are created, or when a safe-to-process timer interrupt occurs, the ISRs in PtidyOS end with a call to processEvents(). As mentioned before, this is the main scheduling function of the Ptides scheduler, and is the second part of the context switching algorithm. This routine traverses through the entire event queue, and processes all events that are declared safe-to-process. The pseudo-code is shown below:

```
1  processEvents(int restoreStateFlag) {
2      processVersion++;
3      platformTime = getPlatformTime();
4      disableInterrupts();
5      Event* event = peekNextEvent(NULL);
6      while (event && hasEarlierDeadline(event)) {
7          earliestTime = safeToProcessTime(event);
8          if (platformTime >= earliestTime) {
9              pushDeadline(event);
10             enableInterrupts();
11             fireActor(event);
12             platformTime = getPlatformTime();
13             disableInterrupts();
14             popDeadline(event);
15             freeEvents(event);
16             event = NULL;
17         } else {
18             setTimerInterrupt(earliestTime);
19             localProcVer = processVersion;
20             enableInterrupt();
21             disableInterrupt();
22             if (localProcVer != processVersion) {
23                 break;
24             }
25         }
26         event = peekNextEvent(event);
27     }
28     enableInterrupts();
29     if (restoreStateFlag) {
30         restoreState();
31     }
32 }
```

On line 5 and 26, peekNextEvent() is used to traverse through the event queue. This method takes a pointer to Event as an input. If the input argument is null, then the earliest deadline event from the queue is returned. If the input is not null, then the event of next earliest deadline compared to the input event is returned. If the input is already the event with the latest deadline (last event) in the queue, then a null pointer is returned. Using this method, the main while loop performs safe-to-process analysis for all events until a safe one is found (line 8). In that case, the safe event is processed, and we start again from the top of the event queue (line 16 and 26). For each event that is unsafe, a timer interrupt is set up to wake up the system at the earliest platform time when that event becomes safe (line 18), and the safe-to-process analysis is performed for the next earliest deadline event.

Also note a deadline stack is used to decide whether the preempting event should be executed before the preempted event. Before a safe event is processed, and before interrupts are enabled, that event's deadline is pushed onto the stack (line 9). When another interrupt preempts the currently executing event, the deadline of the preempting event is compared to that of the preempted event through hasEarliestDeadline() (line 6). The scheduler returns to process the preempted event if that event has a larger or equal deadline. Mirroring line 9, line 14 pops the deadline once actor firing finishes. When there are no more events to process, or when the preempted event is of earlier deadline, all system interrupts are enabled (line 28), and we restore the machine to its previous executing state.

Finally, lines 2 and 19-24 are added as an optimization for events with earlier deadlines. Imagine the case where the event queue is very large, and none of these events are safe. Then processEvents() would traverse the entire queue, testing each one for safe-to-process before returning. If an interrupt occurs while we are traversing through the queue, that interrupt will not be handled since interrupts are disabled during the traversal. If the requested interrupt posts a event with a very early deadline, then that event might miss its deadline due to the traversal overhead. To avoid this behavior, lines 20 and 21 are added to enable interrupts briefly so that they can be handled. However, if an interrupt indeed occurs between these lines, then the event queue might have been modified, and *this* instance of processEvents() must stop executing; i.e., we break out of the while loop at line 23.

The only other function linked during code generation is the main entry function. The main() function first initializes all actors. During the initialization, actors may post initial events onto the event queue. Following initialization, processEvents() is called with restoreStateFlag set to 0 to process any of the initial events. On line 30, restoreState() in processEvents() is only called if restoreStateFlag is true. Since no interrupt has occurred, there is no need to restore the original state of execution. At the end of processEvents(), the machine can go into hibernation if there is hardware support for such mode. If not, we simply loop forever in while (TRUE) to wait for future interrupts.

```
void main() {
    initializeAllActors();
    processEvents(0);
    while (TRUE) {
        hibernate();
    }
}
```

3) Single Stack for Storing Execution Context: As mentioned in Sec. IV-A, PtidyOS uses a single stack for event execution. To ensure the responsiveness of the scheduler, PtidyOS still allows preemptive context switching even under a single stacked scheme.

There are two important properties with our scheduler that allow us to implement a single-stacked solution:

1) The deadline of an event does not change throughout the lifespan of the event.
2) PtidyOS schedules events according to the monotonic ordering of the deadlines.

These properties ensure that if an event e_1 is preempted by a earlier deadline event e_2, then e_1 will never be processed before e_2 finishes processing; i.e., the saved state for e_1 will not be accessed before the state of e_2 is accessed. This in effect allows the use of a stack to save the state for all processing events. Each time an event e_1 is preempted, its state can be simply pushed onto the stack. The only time this state is accessed again is when e_1 becomes the earliest deadline event in the system. As the stack grows, more events are preempted, and the priority of the events on the stack grows monotonically as well. Due to this property, PtidyOS does not need to keep a separate data structure that stores all the currently preempted events. Instead, the event states are all pushed onto the stack. Thus, the only context switch operations are to push the current state in order to execute a new event, or pop the last preempted processing state. Note these are $O(1)$ operations, which do not depend on the number of preempted events in the system. Thus, it is easy to bound the context switching time in the PtidyOS environment.

4) Stack Manipulation: As we mentioned before, permitAllInterrupts() assumes there exists hardware support (usually in the form of writing a register bit) for the user to signal the end of this ISR, and to allow the CPU to process other interrupts. Unfortunately, not all microcontrollers provide this support. The ARM Cortex-M microcontroller (which was used for prototyping) for example does not to provide such functionality for security reasons. Instead, a stack manipulation scheme is used to achieve the desired behavior. To take advantage of this scheme, the ISRs are modified, and we show the modified sensor ISR below:

```
1 externalInterrupt() {
2     saveState();
3     fireActor(thisSensor);
4     if (addedNewEvent()) {
5         addStack();
6     }
7 }
```

First, a saveState() function is performed when we enter the ISR. This function saves the registers used to execute the preempted event. Normally these registers are saved when we first enter the ISR. However, as we will explain later, addStack() is a stack manipulation scheme which updates the program counter (PC) to point to an instruction which the compiler does not expect, thus the compiler may not push the necessary registers. Instead, we need to manually save the previously executing registers onto the stack. The functions permitAllInterrupts() and processEvents() are replaced by addStack(), which is a stack manipulation procedure written in assembly. On line 5, addStack() modifies the stack as well as the stack pointer. The modification ensures the PC saved on the stack now points to the start of processEvents() instead of the preempted instruction. Once we reach the end of of the ISR (line 7), the updated stack is popped, and the program executes processEvents(). At the end of processEvents(), a restoreStack() function is called to restore the stack to the preempted state. In the next section we discuss the performance, including latency of the context switching algorithm.

F. Meeting the Key Constraints

We now examine the key constraints we set out to satisfy for PtidyOS.

1) Limited Memory: The base size of the generated PtidyOS is 16.18kB. This includes all utility functions in the PtidyOS API. In this case, the pool of events consists of a single Event structure. This means, 16.18kB is the absolute minimal amount of memory needed to run PtidyOS. As we will talk about in more detail in the next section, the tunneling ball device demo was implemented using the Ptides design flow. After code generation, the entire PtidyOS file is of size 22.32kB, which includes PtidyOS, 5 event structures allocated, as well as the application code.

2) Reactive Concurrency: PtidyOS's support for concurrency is evaluated based on the following two metrics: 1) the concurrency exhibited by the application and 2) scheduling overhead.

Exhibited Concurrency: A common metric to test the concurrency exhibited by an application is to look at the percentage of code that is reachable through interrupt contexts [14]. In other words, this is the amount of code that runs between when an interrupt occurs and when we return to the previously executing state. since all interrupts could end with a call to processEvents(), and processEvents() can access all the actor processing code (through fireActor()), this means the entire PtidyOS code base is reachable through interrupts, except for main entry and initialization functions. This constitutes 83.9% of the code base.

This indicates PtidyOS is highly concurrent, however this behavior is achieved by running the scheduler at the end of ISR. Thus the efficiency of the scheduling overhead has a big impact on the concurrency of the system. We analyze this scheduling overhead next.

Context Switching Overhead: Fig. 4 shows the context switch overhead. Measurements are taken on a Luminary controller that runs at 50MHz. The overhead was measured using an oscilloscope. The x-axis of the timing diagram is platform time, and the number annotated on each execution trace indicates the amount of time (in microseconds) it takes to run a number of procedures. The line numbers on top of the traces indicate the procedures that are run during that period of time. These values correspond to the line numbers shown earlier in the pseudo-code for processEvents(). Since the interrupt handling routines are short, they are not annotated with line numbers. At some points of the timeline, there are multiple procedure running. e.g., at platform time $20\mu s$, two procedures are active. This indicates a possible branching condition. We will explain these branching conditions in more detail below.

As mentioned before, the context switching procedure can be broken down into two parts: interrupt handling and scheduling. The interrupt handling overhead takes either $7.14\mu s$ or $7.74\mu s$ to complete. If the interrupt did not insert new events into the event queue, then the system simply restores its original state, and goes back to the preempted event. In this case, the total latency for this interrupt is $7.74\mu s$, which translates to 387 cycles given a 50MHz clock. If however, new events have been inserted into the event queue, then it takes $7.14\mu s$ before addStack() finishes, and processEvents() starts.

It might seem unintuitive that the execution trace where an event is inserted takes longer than the execution trace where no event was inserted. This is because restoring the previous execution state requires writing more registers than addStack(). Also notice, in both cases, we assume a sensor firing time (execution time of fireActor(thisSensor)) of $2.7\mu s$, based on the tunneling ball device example discussed in the next section.

Recall that addStack() leads to the running of the scheduler, which performs processEvents(). There are four possible scenarios: the event queue is empty; the queue is not empty, but the preempted event is of higher priority; the queue is not empty, the preempted event is of lower priority, but the preempting event is not safe to process; or the queue is not empty, the preempted event is of lower priority, the preempting event is safe to process.

The first two cases result in almost the same overhead, since hasEearlierDeadline() is an $O(1)$ operation that takes less than $1\mu s$ to process. These two cases also correspond to the situation when context switching time is the smallest. The scheduler would simply call restoreStack() to return to the previously processing event. In this case processEvents() takes a total of $11.75\mu s$ to complete.

If the event is safe, then interrupts are enabled after $15.01\mu s$. Right after enabling interrupts, that event is processed. In other words, the context switch time to process the earliest deadline event in the queue (assuming that event is safe) takes $7.14 + 16.63 = 23.77\mu s$, or roughly 1100 cycles. Finally, if the event is not safe, processEvents() takes $32.46\mu s$ to finish, which includes setting up a timer interrupt.

To put these numbers into perspective, assume there are n events in the event queue, and none of these events are safe except for the last one. Then, it would take $7.14+6.35+(3.38+17.38)\times(n-1)+3.38+5.28 = 22.15+20.76\times(n-1)\mu s$ until the n^{th} event is processed. Assuming there are no other interrupts occurring during this time, this is the worst case context switching overhead time.

Also just to emphasize, as we have discussed earlier, since interrupts are enabled after the safe-to-process analysis of each event, if earlier deadline events are inserted into the event queue during context switching, we can switch to those events and immediately process them. In other words, the PtidyOS scheduler is optimized for the earliest deadline events (the essence of EDF).

V. Design Flow Experiments

This section discusses two example applications implemented with the Ptides design flow.

A. Execution Time Simulation

The first example is the simple Accumulator model shown in Fig. 2. C code is generated using the Luminary microcontroller as the target platform. The main functionality we wish to illustrate with this example is the ability of the Ptides simulator to simulate the timing of critical time instants in the execution of the model, including end-to-end delays between sensors and actuators. This ability to simulate physical time delays enables checks against deadline misses for Ptides models. We will show that with properly annotated execution times and scheduling overheads, the simulated end-to-end delays are tight bounds of the actual delays.

For the Accumulator model, execution time for each of the actors is measured individually on the target Luminary platform. This information is annotated via the executionTime parameter at each actor's input port. Notice that in this model, the execution time of the SensorHandler includes the context switching time as well as time to produce a sensor event.

Next, we need to annotate the model with scheduler overhead execution time. This overhead can be captured in the schedulerExecutionTime parameter of the Ptides director. As discussed in the previous section, the scheduling overhead depends on whether the events are safe. By default, we take the conservative approach and assume that all events are *not* safe, which yields the largest overhead. As shown in the last

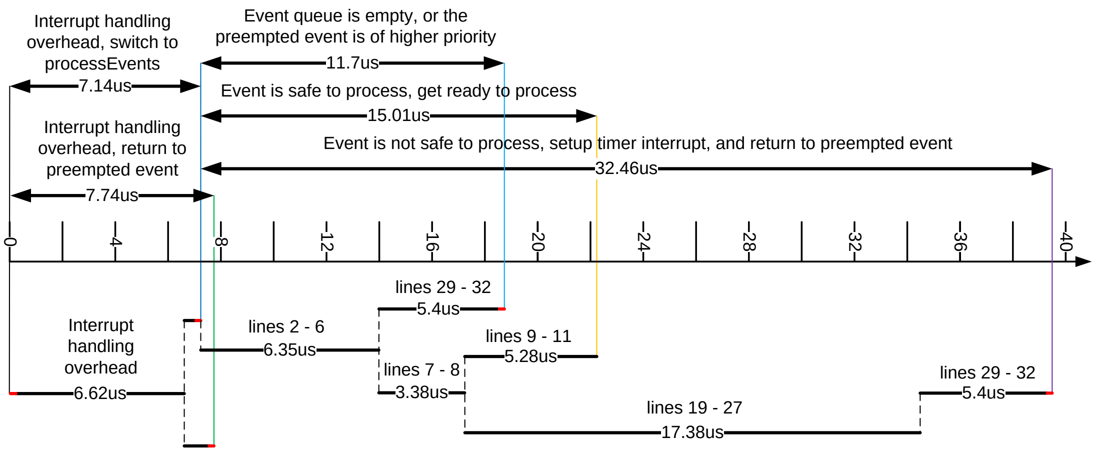
Interrupt handling overhead, switch to processEvents
7.14us
Event queue is empty, or the preempted event is of higher priority
11.7us
Event is safe to process, get ready to process
15.01us
Interrupt handling overhead, return to preempted event
Event is not safe to process, setup timer interrupt, and return to preempted event
32.46us
7.74us
0
4
8
12
16
20
24
28
32
36
40
lines 29 - 32
5.4us
Interrupt handling overhead
lines 2 - 6
6.35us
lines 9 - 11
5.28us
lines 7 - 8
3.38us
lines 29 - 32
5.4us
6.62us
lines 19 - 27
17.38us

Fig. 4. Scheduling Overhead Timing Diagram

section, assuming there is at most one event in the queue, this overhead is measured to be $32.46\mu s$. We will now illustrate how these values can be used to simulate end-to-end delays in different scenarios.

The first scenario we consider is the one where only the EventSensor is connected, while the ResetSensor is idle. Here we also assume $d_o = 0$ at both sensors. By annotating the Ptides model with measured execution times for actors and the scheduler, the Ptides simulator shows an end-to-end delay from EventSensor to DisplayOutput of $175.69\mu s$. However, the actual measured delay is $110\mu s$. The reason our simulated delay does not provide a tight bound is due to the fact we assumed events are always *not* safe-to-process. However, in this case, since $d_o = 0$, input events are actually *always* safe, which leads to over-estimation of the scheduling overhead. If we use the previously measured overhead for the case where events are always safe, which is $16.63\mu s$ (an overhead of $15.01\mu s$ is shown in Fig. 4; however, that does not include functions popDeadline() and freeEvents() as shown in lines 14 and 15 of processEvents()), the end-to-end delay reduces to $112.37\mu s$, which is a tighter bound for the measured delay of $110\mu s$.

Next, we measured the end-to-end delay for the case when both inputs are triggered by the same input signal. Here both sensor interrupts would run one after the other. This adds additional end-to-end latency into the system. The delay from EventSensor to DisplayOutput is simulated to be $162.8\mu s$ in this setup. The actual measured delay is $144\mu s$. Again, the optimistic overhead of $16.63\mu s$ is used. Notice this bound is not as tight as what we obtained the last time. This is due to an imperfection of the current simulator implementation, which assumes a scheduling overhead after each actor firing, including ISR executions. However, in the actual implementation, since EventSensor and ResetSensor are triggered at the same time, the sensor ISRs execute one after another, and only one context switch to processEvents() is needed to find the safe event (the second call to processEvents() is stacked, and will be called after all safe events are processed). Indeed, if the simulator supported this simulation strategy, then a tighter bound of $162.8 - 16.63 = 146.17\mu s$ would be simulated, which is closer to the measured value.

Notice the execution times for Accumulator and TimeDelay are larger in the latter scenario. Recall from Sec. IV-C, addEvent() is an $O(n)$ procedure, where n is the number of events in the queue. Since addEvent() is typically called as a part of the actor firing, to correctly simulate the end-to-end delay, the execution time of the actors must be modified.

In both of the previous cases, we assumed a sensor real-time delay $d_o = 0$. In the last case, we measured the end-to-end delay assuming real-time delay $d_o = 60\mu s$. This delay translates to a non-zero delay offset (defined in Sec. III-A) at the input ports of the Accumulator. Here the overhead to execute the safe-to-process interrupt takes $10\mu s$ (an event becomes safe at the $60\mu s$ mark, but processEvents() does not start until the $70\mu s$ mark). This delay can be attributed to two sources. First, even though we would expect the interrupt to occur at platform time $60\mu s$, it actually happens at $65\mu s$. This delay is related to the hardware timer support of the Luminary microcontroller. The functionality of the Luminary microcontroller's timer interrupt can be summarized as follows: a value is loaded into a timer register. As soon as this timer is enabled, it starts counting down. When it counts down to zero, an interrupt occurs. Thus, this timer is perfect if we want to wakeup after a specific period of time. However, this timer is less optimal when we wish to wake up *at* a specific time. Notice the subtle difference here, where we need to first get the current platform time, and then perform a subtraction in order to find the value that should be loaded into the timer register. Both of these actions together with the interrupt latency contribute to the $5\mu s$ delay before the safeToProcessInterrupt is handled. In addition, interrupt handling of the safeToProcess interrupt plus the time to context switch to processEvents takes another $5\mu s$, thus the total overhead for the safe-to-process interrupt is $10\mu s$.

A more ideal microcontroller platform for the purpose of Ptides would be one which provides hardware support for interrupt triggers at a specific time. However, even then the interrupt handling overhead may not be negligible. Thus the Ptides simulator provides another parameter called safeToProcessTimerHandlingOverhead to capture this latency. In the above scenario $10\mu s$ is entered for this parameter. The simulated delay is predicted to be $161.56\mu s$, while the actual measured delay is $161\mu s$.

B. Application Example

1) Application Setup: A more real-world application called the Tunneling Ball Device (TBD) was developed by Jeff C. Jensen [8]. The detailed explanations of this device can be found in [9]. This setup is chosen because it is an exemplary motion control application. In such an application, the physical plant usually interfaces to the computers through three kinds of signals: periodic, e.g., control loop signals; quasi-periodic, e.g., signals whose rates are proportional to the velocity of the motor; and sporadic. e.g., signals that indicate irregular events.

The physical device consists of a spinning disc with two holes on opposite ends, and the disc is connected to a

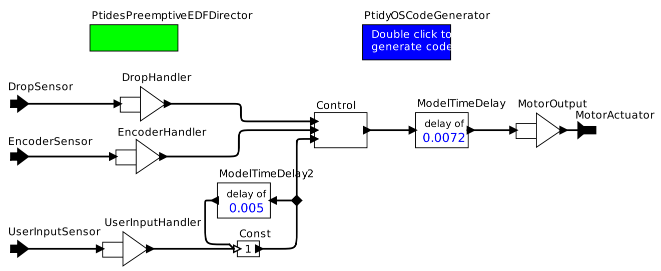

Fig. 5. Ptides Model For Tunneling Ball Device

motor. The motor is controlled by a microcontroller, which interacts with the environment through GPIO and pulse-width-modulator (PWM). Every revolution, the motor generates encoder pulses to a GPIO pin of the microcontroller. These pulses are interpreted to measure the current position of the disc. The controller outputs a periodic pulse through the PWM. The width of the pulse dictates the power output of the motor, and thus the speed of the disc. Aside from the motor-disc setup, there is also a tube placed above the parameter of the disc, with two optical sensors mounted on the top of the tube. Metal balls are dropped through these sensors. When drop events are detected, pulses are sent to the microcontroller through another GPIO pin. According to these pulses, the speed of the ball is calculated. Using this information, along with the vertical distance from the sensors to the disc, and the current position of the disc (which is captured by the encoder ticks), the control algorithm calculates the change in disc speed to ensure one of the holes intersects the trajectory of the ball so the ball can pass through it. To control this application, Jeff Jensen developed a C implementation that runs without an operating system. Another implementation is developed using the Ptides design flow.

The C program consists of four external interrupt handlers: encoder interrupt, which updates the current position of the disc from the motor encoder ticks, planned position interrupt, which updates the planned position of the disc, ball drop interrupt, which indicates that a ball drop event has occurred, and user startup interrupt, which starts rotating the disc at a nominal speed. The main function of the C program runs in an infinite loop. As control theory dictates, the controller must run periodically. Thus the loop starts by busy waiting until the current platform time is equal to a multiple of that period. This is followed by polling for system state information and executing the control routine, which implements a proportional-derivative (PD) controller. The calculated output is sent to the PWM, followed by returning to the start of the loop. This loop runs indefinitely until the system reset signal is received. We will show that this common style of polling for sensor data as control inputs is actually nondeterminate. Although this nondeterminism is tolerable in most cases, the determinism of Ptides has measurable advantages when a large delays is experienced by the sensed data.

The Ptides model for the TBD is shown in Fig. 5. This model abstracts each of the previously mentioned sensors as sensor actors except the planned position interrupt. Instead of having an interrupt that updates the planned position, a local variable in the Control actor keeps track of this position. This variable is updated every time the control loop runs. An ISR is generated for each of the sensors in Fig. 5. The main control logic resides in the Control actor.

2) Application Analysis: The key Ptides feature we will demonstrate with this application focuses on the use of timed semantics to ensure deterministic event execution. In particular, when a drop event occurs, the controller calculates the new

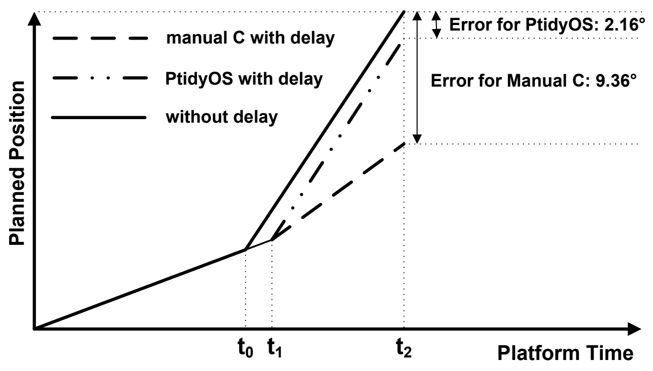

Fig. 6. Position Errors For PtidyOS vs. Manual C Implementation

speed of the disc based on the time-to-impact for the drop ball (calculated from the ball drop event) and the current position of the disc (calculated from the encoder event). Notice that the implicit assumption here is that both of these values refer to the state of the disc at the same instance in physical time. However, this assumption may not be true in general. Our goal is to compare the manual and the Ptides implementations in situations where this assumption does not hold.

In particular, we are interested in the scenario where the controller is physically far away from the drop sensor. In order to reduce the amount of wiring in the physical implementation, a separate platform would be used to parse drop sensor data. This drop sensor controller would then communicate with the controller through a network. This communication delay can range from microseconds to a few milliseconds depending on the communication protocol used. We will examine the effect of this delay on both implementations. Note that we have not implemented a distributed control of the TBD. Instead, an artificial delay of $5ms$ is introduced at the input of drop sensors using timer interrupts, i.e., when a drop event occurs at platform time t, a timer interrupt is set to occur at $t + 5ms$.

As shown in Fig. 6, if there is no delay at the sensor, the new rate of the disc is calculated immediately at t_0. If the drop data is delivered at t_1, the Ptides implementation would still acknowledge the drop occurred at t_0 due to timestamping at the drop sensor platform. Also, Ptides semantics still ensures events from drop and encoder sensors are processed in tag order. This means the calculated output will have both variables referencing the same state of the disc, which allows for the correct calculation of new disc rate (the slope for Ptides implementation with delay is the same as the slope for the case without delay). Thus, even though the Ptides implementation experienced some position error as a consequence of the sensor delay, the rate calculation is not affected, and the position error at impact time is minimized.

For the manual C implementation, however, the drop data still indicates that a ball drop occurred at t_0, but since the manual implementation has been polling encoder inputs during the $5ms$ delay interval, it would use the current position of the disc at t_1 instead of t_0. This leads to a wrong rate calculated for the manual implementation, which leads to a much larger position error.

We performed experiments with both the PtidyOS and manual C implementations to control the TBD, first without and then with the simulated $5ms$ network delay. The nominal speed of the disc was set to 3 revolutions per second (RPS).

For the Ptides implementation, a delayed sensor input results in a planned position error of 3 encoder ticks, which translates to $2.16°$. In other words, at the impact time of the ball, the planned center of the disc is $2.16°$ away from the ideal position, where a ball drop is guaranteed to be successful.

This is compared to a position error of 13 encoder ticks, or 9.36° for the manual C implementation. This difference is large enough to cause ball drop failures. A total of 50 ball drops are performed for each implementation. The success rates for different cases of ball drops are shown below:

Manual Without Delay		PtidyOS Without Delay		Manual With Delay		PtidyOS With Delay	
96%	100%	100%	100%	40%	62%	98%	100%

Two percentages are shown for each case. The one on the left denotes percentages of ball drops that went through the hole without touching the disc, while the one on the right denotes the percentages of drops that touched the disc, but still went through the hole, i.e., a drop that touched the side of the hole but still went through would count as a success in the second case, while failure in the first. As we can see, with the added delay, the success rate of the manual implementation dropped significantly for both cases, while the success rate of the PtidyOS implementation remained virtually unchanged.

It should be noted, however, that the deterministic behavior of the Ptides model is not obtained without a price. One limitation of the Ptides implementation is that each time an encoder interrupt occurs, a corresponding event is created and inserted into the event queue. The safe-to-process analysis is then performed on this event. Since we expect around (500 ticks / revolution $\times$ 3 RPS =) 1500 encoder events per second, this process results in much larger scheduling overhead than the manual C implementation (which does not have overheads such as event queue management and safe-to-process analysis). The side effect is that the PtidyOS implementation can only handle lower disc rates. If the nominal rate is set to more than 8 RPS for example, the large number of encoder interrupt preemptions would result in stack overflow in the PtidyOS implementation, where the manual implementation can handle a disc rate of up to 20 RPS.

Finally, we should note that it is, of course, possible to modify the manual implementation such that the calculated planned position takes into account the delay experienced by the drop sensor. Just like the PtidyOS implementation, this would minimize the position error and ensure successful ball drops. However, Ptides guarantees deterministic functional behavior regardless of the sensor delay. Once the sensor delay d_o's are known, the programmer only needs to update the Ptides model with this value, and the working implementation is generated with the click of a button. This opposes the manual implementation, where the programmer has to change both the program structure and functionality manually in order to take into account this delay.

VI. Conclusion and Future work

A set of execution strategies that implements Ptides semantics is introduced in this work. By assuming bounds on synchronization error and network delay, and by defining a relationship between model time and physical time, our strategies ensure event processing that results in deterministic logical and timing behavior, while allowing out-of-order execution without backtracking. These strategies use different event queue orderings and result in different complexities for their safe-to-process analyses.

The second part of this work focuses on the implementation of these strategies as a part of the Ptides design flow. This design flow includes a simulator, a code generator, and a real-time operating system PtidyOS. The simulator allows application programmer to capture both the logical operation and the timing aspects of the interaction with the environment without the knowledge of execution times, which are dependent on low level hardware details. If execution times and kernel overheads are available, however, these information can be annotated as part of the model to simulate deadline misses. The programmer can then generate a runtime implementation onto a target-specific hardware platform.

A motion control application called the tunneling ball device is implemented using the Ptides design flow. The generated PtidyOS runtime is compared to a manual C implementation. We showed the Ptides semantics ensures the correct order of event execution, which prevents performance degradation even with a large delay introduced at the sensor. This contrasts the manual implementation, whose performance degraded significantly when the same delay is introduced.

One possible future direction is to expand the Ptides programming model and enable its use for soft real-time systems. A possible approach would be to define a utility function on event timestamps and have the scheduler maximaize it. On the implementation side, we are working on a distributed setting, since this is what Ptides strategies were originally formulated for. This requires an integration of a time synchronization protocol such as IEEE1588 into the PtidyOS framework.

References

[1] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[2] J. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou. A time-centric model for cyber-physical applications. In *Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, pages 21–35, 2010.

[3] J. C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer, 2006.

[4] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[5] G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, 2001.

[6] D. Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[7] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, 2000.

[8] J. Jensen. Tunneling ball device. http://chess.eecs.berkeley.edu/tbd/.

[9] J. Jensen. Elements of Model-Based Design. *University of California, Berkeley, Technical Memorandum. UCB/EECS-2010-19, February*, pages 2010–19, 2010.

[10] S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, 2004.

[11] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[12] E. Lee, S. Neuendorffer, and M. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[13] J. Lehoczky, L. Sha, J. Strosnider, et al. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[15] W. River. Vxworks: Embedded rtos with suport for posix and smp. http://www.windriver.com/products/vxworks/.

[16] L. Sha, M. Klein, and J. Goodenough. *Rate monotonic analysis for real-time systems*. Kluwer Academic Publishers, 1991.

[17] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11. IEEE, 1994.

[18] S. Thuel and J. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 22–33. IEEE, 1994.

[19] 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.

[20] J. Zou. *From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems*. PhD thesis, EECS Department, University of California, Berkeley, May 2011.

[21] 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 *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, CA, 2009. IEEE.