Java Takes Flight: Time-portable Real-time Programming with Exotasks

Joshua Auerbach
IBM Research
josh@us.ibm.com

David F. Bacon
IBM Research
dfb@watson.ibm.com

Daniel T. Iercan
University of Timisoara
diercan@gmail.com

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

V.T. Rajan
IBM Research
vtrajan@us.ibm.com

Harald Röck
University of Salzburg
hroeck@cs.uni-salzburg.at

Rainer Trummer
University of Salzburg
rtrummer@cs.uni-salzburg.at

Abstract

Existing programming methodologies for real-time systems suffer from a low level of abstraction and non-determinism in both the timing and the functional domains. As a result, real-time systems are difficult to test and must be re-certified every time changes are made to either the software or hardware environment. *Exotasks* are a novel Java programming construct that achieve deterministic timing, even in the presence of other Java threads, and across changes of hardware and software platform. They are deterministic functional data-flow tasks written in Java, combined with an orthogonal scheduling policy based on the logical execution time (LET) model. We have built a quad-rotor model helicopter, the JAviator, which we use as a testbed for this work. We evaluate our implementation of exotasks in IBM's J9 real-time virtual machine using actual flights of the helicopter. Our experiments show that we are able to maintain deterministic behavior in the face of variations in both software load and hardware platform.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.3.2 [*Programming Languages*]: Java; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Languages, Measurement, Performance

Keywords Real-time scheduling, UAVs, time-portability, virtual machine

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.

LCTES'07, June 13–16, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-632-5/07/0006…$5.00

1. Introduction

Real-time applications need low and predictable scheduling latencies. This has caused them to be written with specialized methodologies in low-level languages. However, this practice is changing. Recent innovations in real-time garbage collection for Java [3] have made Java suitable for writing real-time applications whose latency requirements are in the millisecond range. Restricted subsets of Java, used in portions of a larger application, are used to achieve still lower latencies in the microsecond range. These include the NoHeapRealtimeThread (NHRT) construct of RTSJ [5], eventrons [20], and reflexes [21]. However, each of these restricted programming models entails problems for the application writer.

Once real-time applications are written in a language such as Java that provides functional portability across platforms, the requirement emerges to make the scheduling behavior portable as well. That is, observable real-time behavior should be the same on all platforms. Of course, this requires adequate resources on each platform. However, if sufficient resources exist, it should not require rewriting or re-tuning the application.

Exotasks are an attempt to solve the timing portability problem and, at the same time, provide a less restrictive capability for programming at low latencies, compared with what is currently offered by NHRTs, eventrons, or reflexes. In practice, restrictiveness is a complex matter, so we do not claim that the exotask model is strictly a superset of the others. Nonetheless, exotasks come the closest to providing standard Java memory management semantics, in which objects are freely allocated and garbage collected only when unreachable. As a result, exotasks are also compatible with a larger (although still reduced) set of Java libraries than other approaches.

Exotasks comprise a programming model defined entirely within the Java language, a supporting tool suite built on the Eclipse framework [7], and runtime support in a cooperating Java virtual machine. Like NHRTs, eventrons, and reflexes, exotasks run on special threads that are exempted from preemption by system threads such as those used to accomplish garbage collection. In addition, exotasks employ some unique features to make that possible.

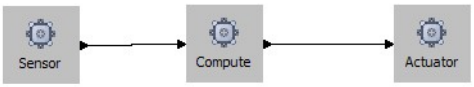


Figure 1. A simple exotask program

Exotasks achieve highly repeatable behavior by enforcing a computational model in which tasks (called exotasks) communicate via explicitly declared channels and are otherwise isolated (logically in time and physically in space) from each other and from the rest of the Java application. The system achieves portable timing characteristics by scheduling exotasks using *logical execution time* (LET) [11]. In the LET model, events involving I/O are executed at precise points in real time while other events are executed based on data dependencies between tasks. The exotask system is inspired by Giotto [11] and its successor HTL [8], but represents a complete rethinking of the syntax of programs and the development cycle so as to fit naturally into the Java programming model.

Exotasks may be written as Java classes that follow certain design patterns, or created with an Eclipse-based graphical editor that generates classes in the required style. All interconnections, and optionally some exotasks, have system-provided behavior. Most exotasks are populated with user-written Java code.

The Eclipse-based programming environment for exotasks is partly graphical in that the exotasks and their interconnections are first declared by dragging and dropping elements from a palette. Navigation between the graphical view and the standard Eclipse Java editors for user-written code is accomplished via the graphical user interface using familiar paradigms such as double-clicking. Exotasks also define a stored format and a programmatic API that permits the graphical phase to be bypassed or a different graphical tool to be substituted.

The exotask system requires a modified Java virtual machine (JVM) that is capable of enforcing exotask memory isolation. Each exotask has a private heap that supports the standard Java memory model. Thus, unlike eventrons [20], exotasks can create objects and modify reachable objects at will. They are restricted only from observing or changing mutable state in (or reachable from) static fields (which would break memory isolation) and from creating new threads (which would prevent accurate scheduling). Unlike reflexes [21], they do not need to pre-declare a distinction between stable and transient objects: standard garbage collection techniques handle all object lifetimes. To support the full development cycle, the characteristics of the exotask JVM can be simulated by the development framework using any JVM; this allows errors to be found early in the development cycle.

The exotask system also supports pluggable system behavior. It is possible to provide new *timing grammars* that define the rules for specifying timing constraints, new *schedulers* that interpret those timing constraints (along with WCET information) to produce correct runtime behavior, and new *distributers* that allow an exotask program to span multiple machines.

Exotasks are *validated* before instantiation to make sure that all of the programming model semantics and restrictions are obeyed. A validated exotask does not require any additional run-time checks, which both improves efficiency and simplifies the run-time system.

The combination of an expressive programming model which allows the use of almost all of Java's features, an annotation-free input language, and a validator which eliminates the need for run-time checks makes exotasks a compelling real-time programming model.

```
public class Compute implements Runnable
{
  // Code generated from the specification:
  private ExotaskInputPort<double[]> in0;
  private ExotaskOutputPort<Double> out0;

  public Compute(ExotaskInputPort<double[]> in0,
                 ExotaskOutputPort<Double> out0) {
    this.in0 = in0;
    this.out0 = out0;
  }

  // Code written by the user:
  public void run() {
    double[] sensorData = in0.getValue();
    double actuation = control(sensorData);
    out0.setValue(actuation);
  }

  private double control(double[] sensors)
  {
    // control algorithm goes here
  }
}
```

Figure 2. Exotask code for the Compute node in Figure 1

2. Exotask Basics

In this section, we introduce the basic features of the exotask programming model using a simple example of a controller for a simulated inverted pendulum with one degree of freedom [16]. Additional features are presented in Section 3 using the actual system for controlling the JAviator helicopter.

An exotask program consists of a *specification graph*, *user code* for some graph nodes (in the form of Java classes), and *timing annotations* applied to the nodes and directed edges of the graph. The nodes of the graph specify *exotasks* and the edges specify *connections* between the *ports* of those exotasks. The user code implements the functional behavior of exotasks. The timing annotations specify the timing behavior of the system.

The exotask specification graph for the inverted pendulum controller is shown in Figure 1, which depicts a screenshot from our Eclipse-based exotask programming environment (timing annotations and user code for the tasks are not shown but can be revealed and edited in the Eclipse environment).

The exotask programming model distinguishes the specification graph (and the Java *classes* that support it) from the *instantiated graph*, which consists of Runnable *objects* corresponding to the nodes and edges of the specification graph. The instantiated graph is what is actually executed. Only the exotask run-time system can create the instantiated graph, and only after verifying the exotask restrictions on data flow, isolation, and timing. The restrictions include type compatibility of connected ports, memory isolation of user code across exotasks, and causality of timing annotations. Once the specification graph has been verified, the run-time system creates the instantiated graph from it by creating the private heaps, instantiating the exotasks in those heaps, and constructing additional objects to represent the ports and connections.

Non-determinism due to just-in-time (JIT) compilation can be avoided either by using an ahead-of-time compiler, or by having JIT compilation forced for all of the methods in the exotask call graph (since the complete call graph is known at instantiation time).

Exotasks are used to implement sensors, actuators, and compute tasks, as well as some advanced features explained in Sections 3 and 4.

Most exotasks have user-written code bodies. Memory isolation is enforced by the verifier by preventing the code body of the task from reading or writing any global variables other than final immutable data, and by disallowing thread creation and reflection. This requires the verifier to construct a summary of the call graphs of all the exotasks (i.e., to find all the reachable methods), which it does conservatively using Rapid Type Analysis [4] on the exotask code bodies and any global immutable data they access (see Section 4.1 for details). Classes are considered live by the analysis if they are members of the instantiated graph, named in new instructions executed by reachable methods, or *admitted* to the graph by native code or intermachine data transfer (admission is described in Sections 2.2 and 3.2).

An exotask has zero or more *input ports* and zero or more *output ports* (but at least one port of some kind). Each port has a data type, specified as a Java class. In Figure 1, Sensor's output port and Compute's input port are of type double[], while Compute's output port and Actuator's input port are of type Double. Arbitrary Java classes (with instance fields that include other classes by reference) may be used freely, as long as the verifier does not find that their methods are being used in a way that violates isolation.

Exotasks are *garbage collected* individually, and garbage collection may be either scheduled or *on-demand*. Scheduled collections occur between task executions at times when the CPU would otherwise be idle. They are the most conservative way of ensuring that garbage collections do not cause non-determinism. As it is generally not necessary to collect each task on every period, overprovisioning is avoided by performing scheduled collections of different tasks in different periods. On-demand collections occur during task execution when the memory region of the task is filled, something that only occurs when scheduled collections are omitted or are done too infrequently. The possible benefits of allowing on-demand collections are discussed in Section 5.2. Non-determinism in memory consumption due to fragmentation is avoided by using the sliding compacting collector of Bacon et al [2].

2.1 Connections and Ports

All communication in the graph is via directed edges (*connections*), which connect an output port of a source exotask to an input port of a target exotask. Connections are constrained to connect ports of identical type. Sharing between tasks is precluded because the operation of moving an object across a connection is semantically a deep copy of the referenced data structure.

Connections are themselves stateless but each port (output or input) can store a value. Thus, the connection and the ports at each end act as a one-stage buffer, allowing the value written by the sending task to its output port to differ from the value available to the receiving task via its input port. The deep copy is made at some time after the execution of the sending task in a given period and before the execution of the receiving task. Consequently, the scheduler is responsible for preventing these executions overlapping.

Exotasks do not share objects with the global Java heap, and may therefore run at a higher priority than the global heap's garbage collector.

2.2 Sensors and Actuators

Sensors are exotasks that only have output ports. Actuators are exotasks that only have input ports. Sensors and actuators generally invoke native (JNI) code to interface with device drivers.

Because exotask verification is based on knowing the set of classes that can be instantiated, sensor specifications must explicitly specify the set of classes they may admit to the graph in native code. Due to subclassing and incorporation by reference, this set may include not only the declared data type of the sensor's output port(s), but also any other classes that the native code may link to the output value via reference fields in the object.

In the inverted pendulum example, the Sensor exotask provides its output port with a vector of doubles, representing the position and velocity of the pendulum's cart and the pendulum's angular position and velocity. In a fully realized version of this simulation, this information would be computed from hardware sensors accessible via native JNI code. Because double[] has no subclasses and incorporates only primitive values, it is also the only admitted class.

The Actuator exotask reads a single java.lang.Double motor control value, rescales it to the appropriate integer range, and writes that to the PWM controller for the motor. Note that for generality, ports take Java objects rather than primitive types, so that scalars like double must be passed in their boxed form.

2.3 Compute Tasks

Exotasks used for computation will have both input and output ports and user-written code bodies. In Figure 1, the task labelled Compute is a task implementing the pendulum control algorithm. Its code is shown in Figure 2.

If the exotask is developed using the Eclipse environment, an enhanced Eclipse class creation wizard generates port instance variables and a constructor based on the graph, and an empty run method. The rest of the code is written by the user.

2.4 Timing Annotations

Timing annotations specify externally visible timing behavior that must be preserved when the program is moved to a different platform. These annotations attach to exotasks, connections, or the graph as a whole. To achieve time portability, annotations should conform to the spirit of the logical execution time (LET) model introduced in Giotto [11]. LET specifies time for external events, but does not specify any timings for internal tasks.

In the inverted pendulum example, the graph as a whole has a period (30 ms), and the sensor and actuator have timing offsets within that period (0 ms and 10 ms, respectively). The internal events are not assigned execution times, in keeping with the LET model.

There are a number of different possible ways of specifying timing constraints. For example, one can specify offsets within a period as was the case in this example. Alternatively, one can use multiple harmonizing periods (as in HTL) or some other notation not yet conceived. Since there are many different approaches to task scheduling, we made this aspect of the system pluggable.

Timing annotations conform to *timing grammars*, which can be added to the exotask system, along with *pluggable schedulers* that expect annotations conforming to those grammars. Intuitively, a timing grammar is a set of syntactic well-formedness rules for attaching timing annotations to elements of the specification graph.

The grammar manifests itself as additional attributes and elements in the specification graph, which may be created programmatically or generated by the Eclipse environment and included in the stored XML representation of the graph. At runtime, the grammar's identity is used to select an appropriate scheduler that understands that grammar. A single grammar may be supported by more than one scheduler. Thus, the exotask programming model is parameterized by timing grammars, and consequently not limited to a particular timing semantics (although we have focused on LET so far).

In this paper, we will discuss two different timing grammars and their supporting schedulers: the time-triggered (TT) grammar and the hierarchical timing language (HTL) grammar. The TT grammar supports annotations that specify timing offsets of exotasks and connections within periods, as well as *modes* (discussed further in Section 3.3). The HTL grammar supports full HTL [8] and is

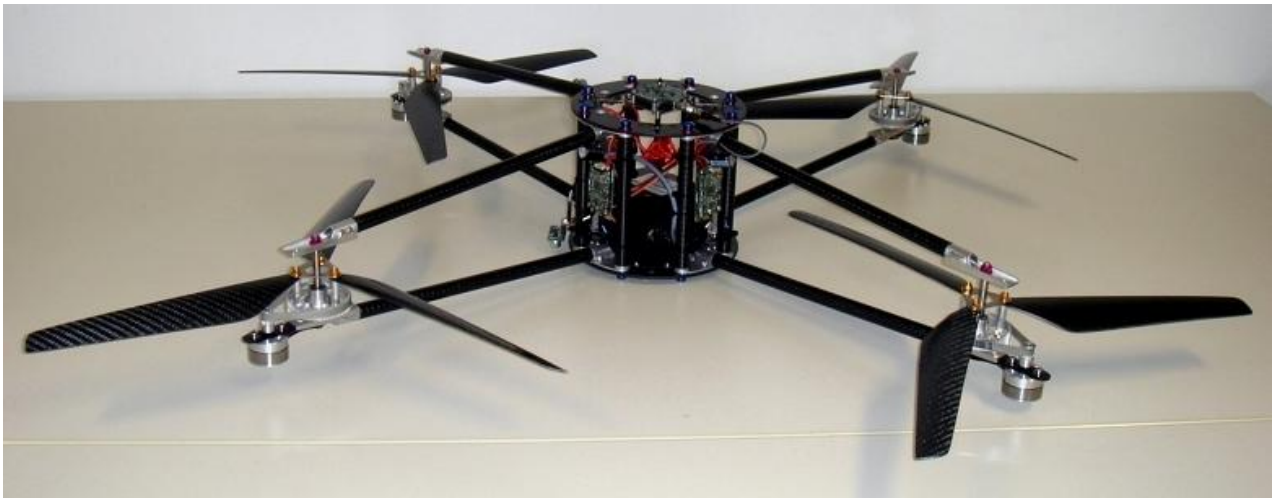


Figure 3. The JAviator: a custom-built quad-rotor helicopter used for experiments in this paper (`javiator.cs.uni-salzburg.at`)

therefore more complex (see Section 5.1.2). For simplicity, we use the TT grammar in both examples, the inverted pendulum and the JAviator, although we also implemented the JAviator system using the HTL grammar.

The combination of deterministic computation enforced by the isolation of exotasks, and the deterministic timing provided by logical execution time, means that a real-time system stimulated with the same external inputs (sensor values) will produce the same actuator values regardless of other running tasks or variations in the hardware platform (assuming only adequate resources to support all the necessary computations).

3. Advanced Exotasks: JAviator

In this section, we will introduce some of the more advanced features of exotasks using our custom-built JAviator quad-rotor helicopter and the associated exotask-based control. The JAviator, shown in Figure 3, is a battery-powered helicopter built from custom-machined carbon fiber, aircraft aluminum, and titanium.

Sensor data comes from a gyroscope providing pitch, roll, and yaw data, and a sonar range-finder for altitude measurements. These are read by a microcontroller, which forwards the data values to the processor on which the exotask-enhanced Java virtual machine is running. All computation is performed there and, upon completion, the values are sent to the microcontroller, which uses them to produce the PWM signals for the motors. The exotask-based control also communicates via a socket to a ground station running a Java program that relays high-level joystick controls to the JAviator and displays an instrument cluster of data from the JAviator.

Figure 4(a) shows the exotask editor opened to the exotask specification graph for the JAviator example. As in the first example, nodes with gear icons represent sensors, actuators, or compute tasks (depending on their ports). As can be seen from the connections, all four such nodes are compute tasks. The meanings of the clock, double arrow, and question mark icons are explained below.

Figure 4(b) shows the properties of the hoverTask compute task: the input and output port names and types, the Java class that implements the exotask, the admitted classes, and the timing annotations appropriate for the selected timing grammar. Properties are changed in a property sheet dialog which contains the same information as the flyover texts.

3.1 Global Timing

The unconnected box at the left with the clock icon is the only node that does not represent an exotask. It represents the selection of a timing grammar (in this case, the TT grammar) and any global properties associated with the chosen grammar. The timing grammar properties in Figure 4(c) show that there are four modes, each with a period of 20 milliseconds.

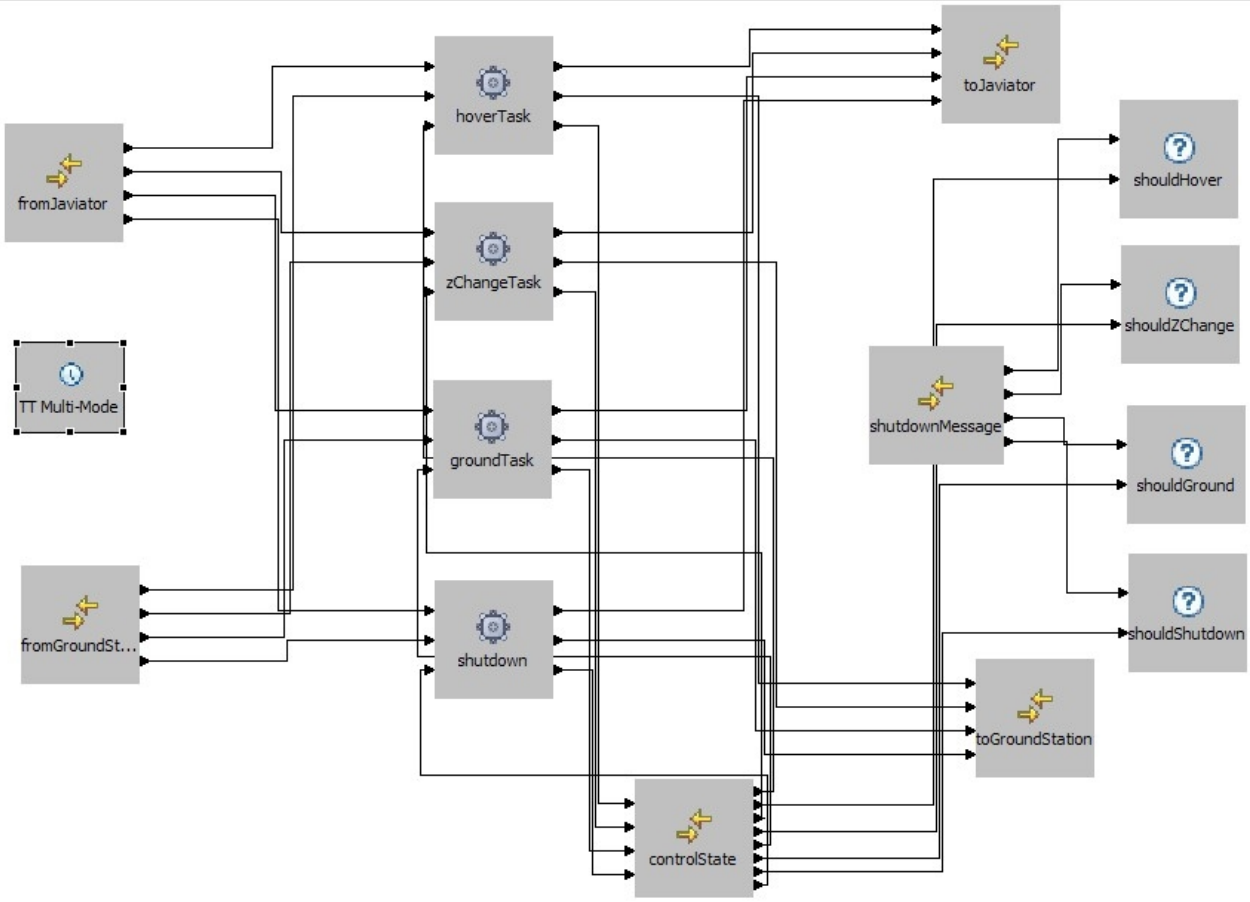
The selection of a timing grammar also augments the property dialogs and flyover texts for other elements of the graph. As can be seen in Figure 4(b), the hoverTask exotask has a "Modes and Time Offsets" property, in which the task is assigned to the Hover mode with no timing offset (meaning it will be scheduled only according to its data dependencies).

Although the exotask model could, in theory, support non-periodic executions, all of the timing grammars and schedulers explored so far have used the concept of a period.

3.2 Communicators

The boxes with double-arrow icons represent *communicators*, which are system-supplied exotasks with one input port and one output port of the identical type, providing several capabilities:

1. Communicators can simply provide an extra buffering stage. For example, the JAviator example uses the controlState communicator to hold controller history across periods of execution.
2. Communicators are attachment points for *distributers*, which extend exotask graphs across machine boundaries. These are discussed further in Section 5.3. For instance, the JAviator example uses the fromJAviator and toJAviator communicators to exchange data with the JAviator's microcontroller, and the fromGroundStation, toGroundStation, and shutdownMessage communicators to exchange data with the ground station. In the example, communicators used for inter-machine distribution are "directional" (either the input or output port is used but not both). This is not an essential requirement, it is simply a choice based on programming convenience and the properties of the distributer.
3. Timing grammars may attach special characteristics to communicators to guarantee scheduling properties. The specialized role of communicators when using the HTL grammar is discussed in Section 5.1.2.



(a) JAviator Example in the Exotask Editor

Implementation=javiator.JControl.exotasks.DoHover
Input ports: ControllerState stateIn, NavigationData terminalIn, SensorData javiatorIn
Output ports: ControllerState stateOut, ReportToGround terminalOut, ActuatorData javiatorOut
No admitted classes
Modes and time offsets: Hover=""

(b) Properties of hoverTask

Modes:
OnGround=20,
Hover=20,
AltitudeChange=20,
Shutdown=20

(c) Properties of TT Multi-Mode

Figure 4. The Exotask System for the JAviator Helicopter

Because communicators can exchange data with other machines, they can admit new classes to the graph. Thus, communicators whose output ports connect into the graph will have an admitted-classes declaration in their specification, like sensors (see Section 2.2).

3.3 Modes, Conditions, and Mode Switching

Modes allow a program to be partitioned into portions that execute under different circumstances. In the JAviator example, there are four modes, corresponding to hovering, altitude change, resting on the ground, and emergency shutdown. Each of the four compute tasks is assigned to a mode, while the six communicators and the four conditions are active in all of the modes.

In the exotask model, mode switching is supported by *conditions*, which give the programmer the ability to control the conditions under which a switch occurs. Conditions are user-written exotasks with only input ports. They compute a boolean value and make it available to the scheduler via a special interface. Conditions may be used whenever the scheduler defines a feedback from the exotask programmer to the scheduler. In Figure 4(a), conditions are represented by the boxes with question mark icons.

The execution-time semantics of a mode is the province of the chosen timing grammar. In the TT grammar selected for this example, conditions are annotated by naming the mode they switch to when true, and the other exotasks and connections in the graph may be assigned to execute at specific times in any subset of the modes. In the HTL grammar (see Section 5.1.2), modes are used in a more stylized fashion. In both timing grammars, each mode has a period of execution and the conditions are only examined at the ends of periods to simplify the scheduling semantics.

The graph in the figure shows the exotask program after all of the modes have been composed. In practice, modes (or any coherent subset of the graph) may be programmed separately, with certain exotasks appearing as common elements. The exotask system recognizes common exotasks (by name) and composes the separately developed graphs prior to instantiation.

3.4 Timing Annotations

Timing annotations on the four compute tasks of the graph assign them to modes, while other exotask belong to all modes. Connections belong to the appropriate modes, depending on what they interconnect. All modes have a period of 20ms. The toJAviator and toGroundStation exotasks are assigned specific timing offsets early in the period so as to create a timely response to the latest sensor data. By convention, the TT scheduler executes nodes with no timing offsets before executing any nodes that are data-dependent upon them.

The controlState communicator is also given a timing offset because it is part of a cyclic data dependency. The TT scheduler only executes untimed nodes to satisfy the data dependencies of timed nodes, so, without this annotation, parts of the graph would not execute. Other schedulers use other conventions for scheduling cyclic graphs.

Everything else in the graph (other than the conditions) will execute before those communicators that are given timing offsets. This includes the fromJAviator and fromGroundStation communicators, which execute at the start of every period due to the rest of the graph being data-dependent upon them.

4. Additional Exotask Features

We have explained much of the exotask programming model through the two examples. What remains are some semantic clarifications and explanation of a few features that were not used in the examples.

4.1 Exotask Memory Isolation

While we do not offer a proof that exotasks are isolated in memory, we can informally establish the property by reasoning inductively that they start out in isolation and thereafter nothing happens that can break isolation:

- The ports of an exotask, which are system-generated, are passed to its constructor with the expectation that they will be stored in instance variables. The exotask gets no other constructor arguments, thus it begins with no references at all, except to its ports and to any objects it creates in its constructor.
- The exotask has a private heap: all objects that it creates go there. It itself resides there, as do its ports.
- When an exotask reads a value from an input port, it gets a deep copy of the value that was placed there by code outside the exotask. When an exotask writes a value to an output port, any code outside the exotask that reads this value will get a deep copy thereof. The values that are visible to the exotask reside only in its private heap.
- No application code outside the exotask retains any reference to the exotask or to any object in the exotask heap. This property is ensured by the exotask system when the specification graph is turned into an instantiated graph. Every item that makes up the instantiated graph is created by the exotask system and no references are leaked to any non-exotask code.
- An exotask may not break isolation through accessing static fields, except for final fields of primitive type, and other objects that the analyzer determines to be immutable. This is enforced by the verifier as was briefly discussed in Section 2; details on immutability analysis are provided below. Once an exotask is verified, it is allowed to execute without dynamic checks.

4.2 Use of Immutable Global Data

One of the trickiest aspects of designing exotasks, or indeed any of the modified Java programming models for real-time (Eventrons, Reflexes, NHRTs) is how to balance the desire for expressiveness and the ability to reuse a maximal amount of pre-existing library code, against the desire to make the model simple, efficient, and exception-free.

In our initial implementation, exotasks were only allowed to read static final fields of primitive types. This made it possible to use a reasonable amount of library code, but there were a number of gaps. Without modification to our JDK, we could use ArrayList and Iterator, but not Integer and HashSet.

We have greatly expanded the amount of usable library code with a two-pronged solution: first, a much more powerful analysis which is able to detect that many objects are either immutable or not accessible to mutating code. The "Ref-immutable" analysis performed by Reflexes [21] is similar but we have increased its power by inferring fields to be "effectively final" and making the analysis data-sensitive (see next section). Second, in the case of objects that are in fact never mutated but for which the analysis is still not sufficiently powerful, the exotask system (but not the user!) can specify some classes as "known to be immutable".

Once an object in the global heap is allowed to be accessed by an exotask, the runtime system must do two additional things: first, the object must be pinned so that the global garbage collector (if it performs compaction) does not move the object, since this would create a race condition between field access by the exotask and object relocation by the collector.

Secondly, isolation must be preserved in the face of synchronization operations on the object, as if the exotask had a private copy. The runtime system simply ignores locking operations by the exotask. This is safe because the exotask is isolated and single-threaded, and exotask code which invokes wait or notify is rejected by the validator.

4.2.1 Immutability Analysis

When the exotask validator runs, it builds a call graph for each exotask in the graph. This call graph may include functions that access static fields. The analysis proceeds recursively as follows: any field that is either final or effectively final because it is private and never mutated outside of the call tree of its constructor, may be accessed by the exotask.

Access of objects reachable from static final fields takes advantage of the fact that by the time the validator is running, all of the class objects referred to by methods in the exotask call graph will have already been instantiated in the global heap (or forced to be so by a call by the validator to Class.forName with the initialize parameter set to true). Therefore, we can use a *data-sensitive* analysis [20] that takes advantage of the dynamic types and field values of the objects, rather than just using information about the static types (which would include subclasses, some of which might not yet be loaded). Data-sensitive analysis is much more accurate, which means that more code may be safely accepted by the validator.

In particular, the analysis examines the code of the exotask call graph in conjunction with its static final referents in the global heap. If the call graph contains an access to a field of such an object, then that field is added to the set of potentially accessed fields, and validation proceeds recursively with that field.

As immutable objects are added to the set of accessed global data, they may cause additional edges to be added to the exotask call graph. Thus the data-sensitive rapid type analysis proceeds in an iterative fashion until a fixpoint is reached.

This analysis is powerful enough for most classes in java.util. For example, it successfully validates the HashSet class, whose only static object is a dummy value of type Object called PRESENT. This object is used because the HashSet is actually implemented with HashMap, with all objects in the set mapping to the value PRESENT.

4.2.2 Immutability Declaration

There are some cases where the data-sensitive analysis is still not powerful enough to determine that a static object is in fact immutable. This occurs with the class Integer, which caches boxed versions of integers smaller than 256 in an array. The analysis is unable to prove that the data structure is immutable. However, simple manual inspection shows that in fact it is immutable and therefore safe to use from an exotask.

To handle this case, the exotask runtime system includes an internal method called CodeValidator.leakproof, which takes a class name and a field name. That field is then considered by the validator to be readable by exotask code. Note that a field may not be declared "leakproof" unless it really is immutable: this is an augmentation for analysis, but not for cases where a runtime mutation is "unlikely".

So far, we have only 13 such declarations, primarily for the classes Integer and FloatingDecimal.

4.3 Impact of Exotask Restrictions

Other than the restrictions above, an exotask can use the entire Java language. The restrictions on using static fields still inhibits the use of library code, but that effect is greatly reduced by the data-sensitive analysis, and in practice we find we are able to use a sufficiently large set of library classes such that the restrictions are not burdensome.

Furthermore, for control programming the code will generally comprise new Java classes, which will naturally avoid the use of static since they are developed as exotask code to begin with. This is what was done for the JAviator control, in which the data types flowing between exotasks were designed for the project using primitive types, incorporation by reference, and whatever methods were needed for convenience.

Most of the classes in java.util that do not pass the validator could be rewritten quite easily in such a way that they did. Generally, the price paid for this is a more restricted use of caching techniques, which results in some additional space overhead. However, the benefits of locality and isolation are useful not only for exotasks but for other aspects of the JVM implementation, and such rewriting is in fact being contemplated for the IBM J9 class libraries.

Exotask programming will still be subject to some limitations, which will probably be most onerous in the case of third-party libraries. However, we believe we have reached a level of permissiveness where the limitations are minor, and well offset by the increase in functionality and real-time behavior.

4.4 Permitted Graph Topologies

There are few universal constraints on the topology of graphs, although there may be constraints imposed by timing grammars in order to ensure that graphs can be scheduled. An output port may be the source of any number of connections to different input ports. An input port may, in general, be the target of any number of connections from different output ports, but schedulers must ensure that there are no data races. Consequently, timing grammars will typically constrain multiple incoming connections to cases that make sense (e.g., the rule that each incoming connection must belong to a different mode, which is the rule in both grammars that we implemented).

As was seen in Section 3, cycles are permitted, but each timing grammar will impose rules on how cycles must be annotated to ensure schedulability.

4.5 Exotask Application Development Cycle

As indicated by the two examples, when using the Eclipse-based tools, an exotask program is developed by developing the specification graph visually, then developing the code for the exotasks. At runtime, however, the exotask program will exist within a conventional Java application that uses the exotask system's runtime API to activate, start, pause, and terminate the exotask program.

A specification graph is represented within this surrounding application by an ExotaskGraphSpecification object, which is a container for a set of exotask specifications and a set of connection specifications, with timing annotations attached. A programmatic API can construct any well-formed ExotaskGraphSpecification from scratch. Alternatively, a parser will produce an ExotaskGraphSpecification from the stored XML format produced by the exotask graphical editor. Typically, the programmer will use a tool in the development environment that creates a Java class with a static method which, when executed, will produce the ExotaskGraphSpecification. However, the parser may also be invoked directly at runtime, in which case the XML representation of the specification graph becomes part of the application source code.

To activate the graph, the surrounding application calls instantiate on the ExotaskGraphSpecification. This runs the verifier and also invokes the appropriate scheduler (depending on the timing grammar) to produce an ExotaskGraph object representing the instantiated graph. If the specification graph fails verification or scheduling, an exception is thrown instead. The ExotaskGraph object resides on the public heap but will be managed by the exotask system to provide the needed start, pause, and terminate behavior.

During development, the ExotaskGraphSpecification resides in the memory of the Eclipse development platform. A tool allows the developer to invoke instantiate on this object at whatever point he believes it to be valid. Therefore, by the time instantiate is called at runtime it is unlikely to fail.

This approach requires the verifier to operate in two modes. At development time, it reads Java class files. At runtime, it processes the actual bytecodes of the loaded classes in the VM. The verification library is built on top of a portability layer that abstracts away details of how the class information is obtained.

5. The Exotask System Design

In the exotask system, computation is a cooperative endeavor involving the exotask program and system, as well as a *scheduler*, and an optional *distributer*.

While the exotask programmer views the scheduler and distributer as just part of the system, these are actually pluggable components. Schedulers assume that the timing annotations of the specification graph conform to particular timing grammars. These grammars are also pluggable components.

In this section, we describe the scheduler and distributer plug-points and briefly discuss the two actual schedulers for the TT and HTL grammars, and one distributer we have developed so far.

5.1 Exotask Schedulers

An exotask scheduler is responsible for deciding when every exotask and every connection should execute. Executing a connection means deep-copying a value from an output port of one exotask to an input port of another. The scheduler also decides when every exotask should be garbage collected (between executions) to prevent on-demand collections that could perturb execution timings. The scheduler is obligated to obey both timing annotations and data dependencies, and to observe the rule that adjacent entities (e.g., a connection and the two exotasks that it connects) may not execute concurrently. This requirement guarantees race-freedom in the access to ports without requiring synchronizations that could make the timing of executions less predictable.

Exotasks and connections are presented to the scheduler as Runnable objects, designed to be called repeatedly on threads belonging to the scheduler (as opposed to normal Runnables, which are permanently inhabited by a thread). Exotasks are presented as

implementations of ExotaskController, which extends Runnable to add a garbageCollect method. The run method of ExotaskController wraps the call to the user-written run method with logic to switch from the scheduler's heap to the private heap of the exotask. Thus, all allocations done in the exotask will allocate to the private heap and not be directly visible, not even to the scheduler.

The scheduler creates its own metadata from the specification graph and its timing annotations after verification and instantiation of the graph but before the surrounding application is given access to the instantiated graph. To guide the creation of a schedule, the scheduler receives an additional object that encodes the WCET information for all exotasks on the current platform. WCETs may be provided for connections as well, if the deep-cloning for those connections may consume non-trivial time.

The scheduler is given its own heap, distinct from the private heap of any exotask and also distinct from the global heap, so that scheduler threads are not subject to interference from global allocation and garbage collection behavior, even while manipulating scheduler metadata. All scheduler threads share the same heap, however, since otherwise, coordination of scheduling across multiple threads would be difficult. To avoid non-determinism due to scheduler heap garbage collections, existing schedulers do no allocations into their heaps once the graph begins execution. System extenders writing new schedulers are required to do the same. If the scheduler heap required garbage collection, it would be difficult to schedule that collection in a non-disruptive fashion, especially if there are multiple scheduler threads.

A scheduler can have as many or as few threads as it requires, all such threads being supplied by the exotask system. The TT scheduler used to collect results for this paper is single-threaded. The HTL scheduler uses as many threads as is required by the concurrency level of the program.

How the scheduler deals with tasks that violate their WCETs is up to the scheduler. A scheduler may have an "overseer" thread that is not used to run tasks but that observes execution and cancels tasks that have exceeded their limit. Exploring this option is future work, as we have made no special provision for restoring invariants after task cancellation.

We have implemented two schedulers that both use the LET model inspired by Giotto.

5.1.1 The TT Scheduler

The TT scheduler uses periods and modes but adopts a conceptually simpler and more explicit approach to assigning logical times. In the TT grammar, events are simply assigned timing offsets within their periods. The time instants when communicators execute are either given explicitly or derived from data dependencies.

Because of the simplicity of their specification, a TT-based graph may have over-specified timing constraints, inhibiting portability. This can be avoided by a more powerful scheduler, such as the HTL scheduler described below. However, simple programs are often easier to express, and trading precision for portability may have practical value.

5.1.2 The HTL Scheduler

The *HTL scheduler* implements the semantics of the hierarchical timing language HTL [8] and requires the use of the HTL grammar. An important difference from the TT scheduler described above is that communicators are periodic in HTL. The time instant when a communicator executes in HTL is derived from its period relative to the periods of the components connected to it. Furthermore, the HTL scheduler is multi-threaded.

The HTL grammar enables an injection from HTL programs to exotask specification graphs: for every HTL program, there is an equivalent exotask specification graph. The HTL scheduler's metadata is a Java version of the E code [10] generated by the original HTL compiler [8].

Because of the injective property, we are able to ensure that all of the desirable properties that have been shown for HTL programs (compositionality, schedulability, refinement, etc.) are inherited by exotask programs that use this grammar and scheduler.

5.2 Exotask Garbage Collection

Garbage collection of the private heaps belonging to exotasks may either be *scheduled* or *on-demand*. Scheduled garbage collections are considered as additional tasks by the scheduler, with their own WCET and period. The period of the task garbage collection need not be the same as the period of the task whose heap it collects.

Each exotask, in addition to its WCET, also has a worst-case allocation (WCA), which specifies how much memory it consumes in a single execution. Like WCET, WCA may be subject to platform-specific variation, due to changes in object representation, pointer sizes, and alignment. However, these variations will be generally smaller than for WCETs, and for a given JVM and broad system architecture (e.g., x86-32) may not vary at all.

Exotasks allow a time/space trade-off since a larger exotask heap will reduce the required garbage collection frequency. When there are multiple exotasks and sufficient memory, heaps that are multiples of the WCA can be used and the garbage collections of different exotasks can be scheduled in a staggered (pipelined) fashion at the corresponding multiple of the underlying period. We use the term *slop* for such extra memory available to the scheduler.

In a system without slop but where slack is available, an exotask can be run in a heap smaller than its WCA. It may then be collected one or more times by *on-demand* collections during its execution, thereby increasing its WCET but reducing its memory consumption. This time/space trade-off is analogous to the time/power trade-offs employed in real-time systems that use dynamic voltage scaling [17].

While on-demand collections may seem risky to programmers used to older methodologies based on static memory allocation, from a scheduling perspective they are no different from scheduled garbage collections: the WCA of the exotask and its heap size must be used to determine the overall WCETs for the scheduler. The sliding compacting collector used for exotasks is highly predictable both in its execution time and in its effects on memory consumption (since it incurs no fragmentation), and instrumentation provided by the exotask runtime system makes it easy to obtain practical WCET bounds.

5.3 Exotask Distributers

Distributers are used to connect exotasks across machine boundaries. The two duties of a distributer are (1) to "replicate" the contents of communicators and (2) optionally, to provide a distributed clock. Replication is the most powerful model since it allows an arbitration based on coordinated time. However, simple message passing can be used as a degenerate form of this model as was done in the JAviator example.

As we expect most programmers to treat distributers as part of the system rather than part of the program, we expect most distributers to be generic, supporting a particular communication fabric or protocol rather than a particular application. The one we have written embodies specific knowledge of the JAviator application, for which we have hand-coded protocols to communicate with the ground station and a specialized RS-232 fabric to communicate with the microcontroller. No exotask code actually runs on the microcontroller, which has just a small amount of C code to communicate with the exotask program and to read or write low-level hardware registers. While we do not expect such application-specific distributers to be typical, neither are they illegitimate. We

	Pentium M	AMD64
Verification	112	118
Instantiation	1641	660
Scheduling	19	14

(a) One-time Exotask Initialization Costs (milliseconds)

	Pentium M			AMD64		
	Min	Max	Avg	Min	Max	Avg
Exotask Run	12	711	25	11	98	16
Exotask Heap GC	0	1043	11	0	647	6
Deep Clone	71	802	95	51	193	67

(b) Per-period Exotask Execution Costs (microseconds)

Figure 5. Execution times in the JAviator program

believe the exotask model to be useful even when actual exotasks are not used everywhere.

In future work we hope to provide support for automated distribution of an exotask graph, with a runtime tool deriving the subgraphs for each machine. The communicators that are to be replicated are discovered in the process of doing the partitioning. We currently do this partitioning manually. The graph in Figure 4 represents the result of such manual partitioning for a single machine.

6. JVM Implementation

Our implementation is based on an experimental variant of IBM's J9 Java virtual machine similar to the WebSphere Real Time product [13], which includes both RTSJ [5], the Metronome real-time garbage collector [3], and an ahead-of-time compiler which can be used to eliminate non-determinism due to JIT compilation.

This variant of J9 includes a thread library enhancement to support nanosecond timing on real-time kernels, along with some native methods that allow a Java program to examine bytecodes and constant pool entries of already-loaded classes (used to create a very lean initialization time verifier).

The internal JVM data structure for a thread includes a set of flag bits, one of which is used to indicate that it is an exotask thread. This flag is used to exempt the thread from being preempted by the global garbage collector (in fact, this is done for threads that have either their `EXOTASK` or `NHRT` bit set). In addition, when the `EXOTASK` bit is set the locking subsystem ignores lock and unlock operations due to synchronized methods and blocks, as described in Section 4.2.

High-precision scheduling is enabled via a high-performance native method we added to the JVM (one that does not use JNI) that invokes the Linux nanosleep function as required by the exotask schedulers.

6.1 Exotask Garbage Collection

In the J9 variant we used, heaps are constructed from multiple *memory spaces*, which are collections of physical areas along with lower-level logic for managing them. The VM also has support for identifying the active memory space of a thread and for switching memory spaces. It supports multiple garbage collectors that can vary across memory spaces. Among the collectors available for this role was one based on the sliding compacting collector described in [2]. This was modified to become the exotask private heap collector.

The collector's root scan was modified to use the exotask's ExotaskController as the sole root in a scheduled collection (because no thread is running in the exotask space). In an on-demand collection, the thread that caused the collection is also scanned to find roots. Only the stack frame representing the exotask run method and any newer stack frames are scanned, because only those frames can have pointers into the private heap.

	Pentium M			AMD64		
	Min	Max	Avg	Min	Max	Avg
GC Quantum	.043	2.21	.548	.041	1.34	.509
GC Duration	384	403	390	172	197	185
GC Interval	8530	8580	8570	8560	8620	8610

Figure 6. Global Garbage Collection Times (milliseconds)

To implement the deep cloning required when objects are sent across ports, the implementation uses "object shape" information that the VM stores on a per-class basis to aid the garbage collector in marking. This information identifies every reference field of every object type. The implementation was aided by the memory space switching support. By establishing the target heap temporarily as the active memory space for the thread, it is possible to reuse standard VM allocation and cloning methods to accomplish the deep-clone, since all copies automatically go into the right target space.

7. Flying the Helicopter

In this section, we present measurements collected during two actual JAviator flights. In Section 7.4, we augment these real-world results with a stress test in which the exotask program was run at four times normal frequency, using a software simulation of the JAviator hardware. All runs were done using the exotask program described in Section 3. Data was collected and analyzed using our TuningFork [12] data collection and visualization system, which is designed to provide very high time resolution without perturbing execution.

The on-board processor for the JAviator is a 400MHz XScale uniprocessor with 64MB of memory running Linux with real time extensions. This processor runs the exotask VM, but there are problems with the real time extensions that make it impossible to collect accurate data from this machine. For the measurements of this section, we approximated the behavior of XScale using a Dell laptop with a 1.4GHz Intel Pentium M CPU and 512MB of memory running a similar Linux real time kernel. Results in Sections 7.1 and 7.4 show that the JAviator application is far from being CPU bound, so running on the XScale should eventually be feasible.

In both real flights, the machine running the exotask control was connected to the JAviator by a thin flexible RS-232 wire. The JAviator was confined in a tower that restricted its motion (for safety) but allowed it to hover freely. The helicopter was airborne for approximately 10 minutes, during which it was monitored visually for anomalies in behavior and was judged to be operating smoothly.

For comparison, data were also collected with the exotask program running on a Dell Poweredge desktop with two 2GHz dual core AMD64 CPUs and 4GB of memory. We judged these machines to be sufficiently different to shed some light on time portability.

7.1 Exotask Run-time Costs

Figure 5(a) shows the times taken by one-time activities that occurred during initialization of the JAviator exotask program. These are (1) the time taken to verify the specification graph and the program's Java code, (2) the time taken to create the instantiated graph from the specification graph, and (3) the time taken by the scheduler to compute its metadata and create its threads. As can be seen from the figure, these times are small relative to the typical duration of program execution. Of course, verification time will vary depending on code complexity. Instantiation time is dominated by

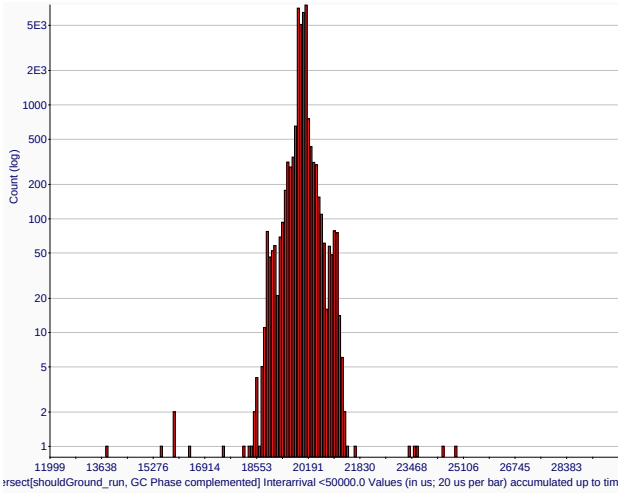


Figure 7. Interval between shouldGround executions, no garbage collection in progress (1.4 GHz Pentium M platform)

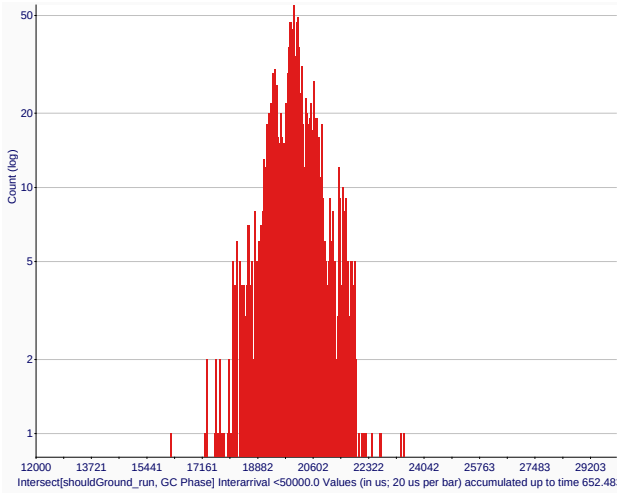


Figure 8. Interval between shouldGround executions, garbage collection in progress (1.4 GHz Pentium M platform)

the time taken to construct private heaps. Scheduling time will vary according to the scheduling algorithm chosen.

Figure 5(b) shows the times for various activities that take place in each 20ms period. These are (1) the total time in which some exotask is running, (2) the total time in which some exotask private heap is being garbage collected, and (3) the total time in which some value is being deep-cloned across a connection. The remaining time in each period is wait time, during which the VM is free to run non-exotask activity. These times are dependent on the specifics of the program but are shown in order to illustrate that exotask execution times and overheads are not excessive. Indeed, the sum of the worst case times on the weaker processor is less than 3ms, suggesting that there will be little problem porting to an even slower CPU. While we are not able to collect precise measurements from the XScale processor, we can tell from system summary measures that its CPU is no more than 25% occupied running the JAviator control program.

Note that on-demand garbage collection was not used in this experiment, so all the GCs were scheduled to occur during otherwise idle time. Thus, the variance in GC time is not a concern for accurate scheduling.

7.2 Interference from Global Heap Garbage Collection

During each flight, a concurrent, unrelated thread was present to simulate the effect of significant other activity within the VM, specifically activity that would cause global heap garbage collections. This thread allocated at a rate of 2MB per second using 48-byte objects and keeping the most recent 40,000 of those objects live. The global GC duration and interval between GCs are shown in Figure 6. The Metronome [3] collector, used by the exotask VM as its global heap collector, divides GC periods into quanta to minimize interference with the application. These are also reported.

Figure 7 shows the distribution of times between executions of the shouldGround exotask (see Section 3) when there is no garbage collection in progress on the global heap (i.e., excluding all intervals during which the global heap garbage collector was running). The shouldGround exotask was chosen for this measurement since it is the first to run in each period. Figure 8 shows the same data for the intervals that fell during global heap garbage collections. Both figures use a logarithmic scale and show the range from 12ms to 30ms. The near-symmetry of both figures is due to the fact that, when an execution happens later than it should, the interval before it is lengthened and the interval after it is shortened by a similar amount.

Because of the logarithmic scale and the differing number of intervals measured (1381 during GC versus 30814 not during GC), the differences between these distributions are amplified visually. The standard deviation in Figure 8 is actually somewhat smaller (.902 versus 1.17) due to fewer outliers, even though there is somewhat greater spread immediately around the mean. Also, when the same analysis was done with the faster AMD multi-processor machine, the GC results were better across the board than the non-GC results (perhaps because the GC locks out other application threads that might interfere).

The results show that the exotask scheduler is indeed free from being paused by the global heap GC (otherwise, the distribution would be substantially worse). On the other hand, the Pentium machine, being a uniprocessor, is more subject to context-switching and caching effects from running concurrently with the GC. On the AMD machine, the GC and the exotask scheduler are likely to migrate to different processors and run more truly in parallel.

Both graphs show a spread of interarrival times of approximately ± 1.5ms. These deviations are rare in absolute terms but are visible in the logarithmic scale employed in the graphs. They appear to be caused by sporadic kernel activity. We have been unable to account for the specific source, but we have ruled out sources within the VM. This jitter has no appreciable effect on the JAviator flight, as it is small at the 20ms scale at which the JAviator control operates.

On the other hand, at the $45\mu s$ time scale explored in the work on eventrons [20] and reflexes [21], a 1.5ms jitter would be unacceptable. It should be noted that the exotask programming model is the most liberal of the three in what it allows, and this application is more thoroughly "real" than the audio application used to illustrate the other two. These things might be expected to come at some cost. Still, elimination of all sources of interference that we can possibly control will be a priority in future work.

7.3 Time Portability

Figure 9 shows the same experiment as is shown in Figure 8, but with the exotask program and its surrounding application running on the faster AMD 64 four-way processor. As can be seen, chang-

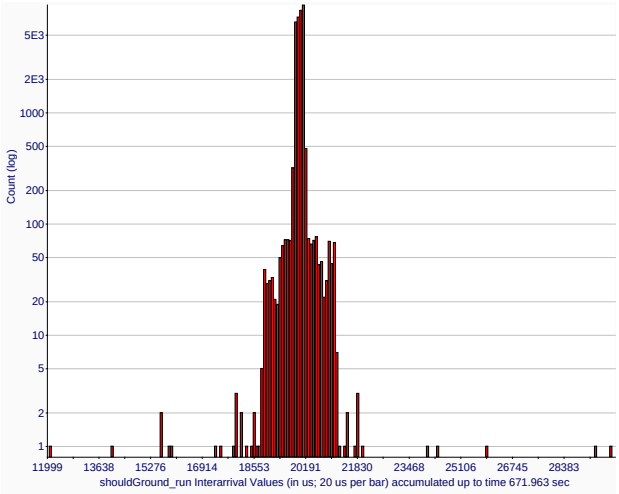


Figure 9. Interval between shouldGround executions, all periods (2 GHz AMD64 platform)

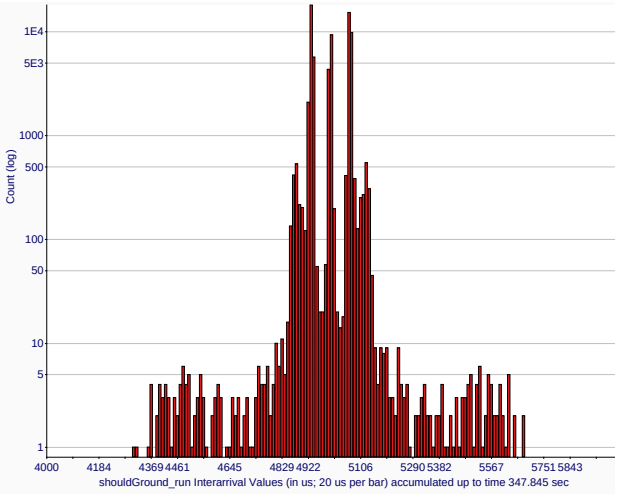


Figure 10. Interval between shouldGround executions, with a 5ms period (simulated JAviator on AMD64 platform)

ing the platform to one with substantially more processing power and different memory characteristics shows the desired result: the period of 20ms is maintained with a highly similar distribution. The fact that the distribution is actually somewhat tighter illustrates that, even when using a technique that preserves timings across platforms, there are effects of hardware on processing that are difficult to control.

7.4 Stress Test

Figure 10 shows the exotask program of Section 3 executing at four times normal speed for five minutes, on the AMD processor, with the allocation thread producing garbage collections as in the other experiments. In order to collect these results, we used a simulation of the JAviator running in the same process as the exotask control (this also eliminated artifacts due to communication). As can be seen, the vast majority of intervals are still closely centered on the target 5ms. In fact, the 1.5ms jitter that is apparent in the other figures is gone, suggesting that the communications activity was implicated in this jitter. The accelerated processing in this slightly more artificial situation illustrates that quite good precision can be achieved by the exotask scheduler assisted by the exotask VM when other sources of interference from the kernel are eliminated. These results also confirm that exotask programming is efficient enough to scale down to weaker processors.

8. Related Work

Time-portable real-time programming in a modern high-level language such as Java requires combining two already established real-time technologies: deterministic real-time scheduling and deterministic real-time memory management. Scheduling in the exotask system is done in two stages: first, events involving I/O are executed at precise points in real time and, second, all remaining events are executed based on data dependencies between tasks. Deterministic I/O timing is the key to time portability but often not available in concurrency models of other real-time languages such as Ada [6] and Erlang [1]. Synchronous reactive programming [9] is an early approach to deterministic I/O timing in which computation is assumed to take zero time, which results in deterministic input (i.e., sensor update timing), but not necessarily in deterministic output timing. A more recent, less abstract approach is the notion of logical execution time (LET) [11]. The LET of a task is the time from the instant when the task reads its inputs to the instant when the task writes its outputs, but not the time when the task actually computes. A LET task's I/O timing is thus user-determined, not system-determined, provided sufficient CPU time is available for the task. Both timing grammars that we developed here are based on the LET concept. However, other potentially non-LET grammars are also possible and subjects for future work.

Real-time memory management (in the context of Java) is a currently active research area. One approach is to improve the garbage collector. The Metronome collector [3], incorporated in the IBM WebSphere Real-Time VM [13], is one of several recent examples. However, this approach is limited by caching and context-switching effects to some lower bound on achievable latencies. And, it does nothing to achieve time portability.

Another approach is to avoid collisions with the garbage collector by avoiding heap allocations entirely, which is the approach taken by NHRTs [5], Eventrons [20], and Reflexes [21]. Eventrons disallow allocation at the programming level (the new keyword is illegal), whereas NHRTs and Reflexes allow allocations while directing those allocations to special memory areas that do not have heap-like semantics. Exotasks are similar to the latter two, except that allocations are directed to a true heap (just not the public one). This is an improvement in programming convenience, and it may prove just as effective if the collections of these private heaps can be made very efficient and scheduled in a way that guarantees no interference with time-critical deadlines. Eventrons and Reflexes achieve a degree of time portability when there are adequate resources, because the Eventron or Reflex executes at regular intervals. Exotasks provide a powerful generalization of this capability by computing with arbitrary graphs of interconnected nodes and pluggable ways of expressing timing constraints. Similar to Eventrons, exotasks use program analysis at initialization time to check conformance to a set of restrictions, rather than with annotations at compile time (as with Reflexes) or continuously during runtime (as with NHRTs).

One rather heavyweight way of implementing private heaps is the *isolates* construct [14] that is now part of Java. Exotasks are not as isolated as isolates: they share classes with the global heap, can read their static final fields, and have explicit connections with other exotasks. In addition, their isolation is achieved in a more streamlined fashion, without the use of memory protection, copy-on-write or separate processes. Isolates, on the other hand, are fully

transparent (almost any Java program can be run as an isolate) while exotasks require observing programming restrictions.

The exotask system provides a visual concurrent and real-time programming environment related to other model-driven development (MDD) environments such as, for instance, MathWorks' Simulink [19] and Ptolemy [15]. The key difference to MDD environments is that the exotask system is firstly and foremost a *programming* and only secondly a *modeling* environment. Simulink and Ptolemy have originally been designed as modeling environments for *simulation* of the models' concurrent and real-time behavior. Subsequently, code generators such as the Real-Time Workshop [18] in Simulink have been added to support real-time *execution* of the models. However, automatic generation of efficient code from models is difficult and often results in insufficient performance. The exotask system does not generate code but instantiates user-written exotask specifications and code bodies, which typically involves much smaller differences in levels of abstraction. The exotask development environment does *some* simulation in order to find major errors (e.g., it runs the same exotask verifier that will be used at runtime). A fuller subset of the behavior of the instantiated code, similar to the generated code in MDD environments, may eventually be simulated in the exotask system, but that lies in the scope of future work.

The exotask programming model is designed to optimize code efficiency, portability, and determinism, rather than semantic expressiveness. Code generated from Simulink and Ptolemy is usually memory-static and not time-portable. Nonetheless, the timing behavior of exotask models is parameterized by the notion of timing grammars and supporting schedulers, which is somewhat related to the notion of abstract syntax and directors, respectively, in Ptolemy. In the exotask system, time portability is a paramount objective.

There are of indeed many systems in which computation is done by a graph of nodes connected by directed edges. The terms "input port" and "output port" are in widespread use. For example, port-based objects (PBOs) [22] also have input and output ports similar to exotasks. However, the exotask system guarantees memory isolation properties and enables time portability while PBOs rely on using coding conventions (in C) and real-time scheduling techniques, which are not semantics-preserving, and therefore, not portable.

9. Conclusions

We have introduced *exotasks*, a novel Java programming construct that achieves deterministic timing, even in the presence of other Java threads, and across changes of hardware and software platform. Exotasks achieve time portability by enforcing a deterministic computational model in which exotasks communicate via explicitly declared channels and are otherwise isolated. Exotasks are logically isolated in time by executing I/O-relevant portions at precise, deterministic points in real time. Exotasks are physically isolated in space by allocating objects in private, individually garbage-collected heaps.

We have implemented a virtual machine that supports exotasks and an eclipse-based development environment to support it. We have used exotasks to fly an actual quad-rotor model helicopter, the JAviator. Our experiments show that exotasks are adequately efficient and achieve freedom of interference from the garbage collector. Comparisons of runs on different hardware show that time portability has been achieved, at least for the one example investigated. In the future, we intend to use exotasks for more difficult control problems involving tasks with different periods executing in parallel. We believe that the same time portability can be achieved, because the scheduling problem has already been explored in the context of HTL.

Acknowledgments

We gratefully acknowledge advice and suggestions from Perry Cheng, David Grove, Michael Hind, and Jan Vitek. David and Perry also helped by fixing TuningFork bugs on short notice.

References

[1] ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*, second ed. Prentice-Hall, 1996.

[2] BACON, D. F., CHENG, P., AND GROVE, D. Garbage collection for embedded systems. In *Proc. EMSOFT* (Pisa, Italy, Sept. 2004), pp. 125–136.

[3] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proc. POPL* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

[4] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA* (San Jose, California, Oct. 1996). *SIGPLAN Notices*, *31*, 10, 324–341.

[5] BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[6] BURNS, A., AND WELLINGS, A. *Concurrency in Ada*, second ed. Cambridge University Press, 1997.

[7] ECLIPSE FOUNDATION. The Eclipse Open Development Platform. `www.eclipse.org`.

[8] GHOSAL, A., HENZINGER, T., IERCAN, D., KIRSCH, C., AND SANGIOVANNI-VINCENTELLI, A. A hierarchical coordination language for interacting real-time tasks. In *Proc. EMSOFT* (Seoul, South Korea, 2006).

[9] HALBWACHS, N. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[10] HENZINGER, T., AND KIRSCH, C. The Embedded Machine: predictable, portable real-time code. In *Proc. PLDI* (Berlin, Germany, 2002), pp. 315–326.

[11] HENZINGER, T., KIRSCH, C., AND HOROWITZ, B. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE 91*, 1 (January 2003), 84–99.

[12] IBM CORP. TuningFork Visualization Tool for Real-Time Systems. `www.alphaworks.ibm.com/tech/tuningfork`.

[13] IBM CORP. *WebSphere Real-Time User's Guide*, first ed., 2006.

[14] JAVA COMMUNITY PROCESS. JSR-121 application isolation API. `jcp.org/aboutJava/communityprocess/final/jsr121`.

[15] LEE, E. Overview of the Ptolemy project. Tech. Rep. UCB/ERL M03/25, EECS Department, University of California, Berkeley, 2003.

[16] OGATA, K. *Modern Control Engineering*. Prentice Hall, 1997.

[17] PILLAI, P., AND SHIN, K. G. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. SOSP* (Banff, Alberta, Canada, 2001), pp. 89–102.

[18] REAL-TIME-WORKSHOP. `www.mathworks.com/products/rtw`.

[19] SIMULINK. `www.mathworks.com/products/simulink`.

[20] SPOONHOWER, D., AUERBACH, J., BACON, D. F., CHENG, P., AND GROVE, D. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proc. PLDI* (Ottawa, Ontario, Canada, 2006), pp. 283–294.

[21] SPRING, J. H., PIZLO, F., GUERRAOUI, R., AND VITEK, J. Programming abstractions for highly responsive systems. In *Proc. VEE* (San Diego, California, 2007).

[22] STEWART, D. B., VOLPE, R. A., AND KHOSLA, P. K. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Eng. 23*, 12 (1997), 759–776.