

Low-Latency Time-Portable Real-Time Programming with Exotasks

JOSHUA AUERBACH, DAVID F. BACON, DANIEL IERCAN, CHRISTOPH M. KIRSCH, V. T. RAJAN, HARALD RÖCK, and RAINER TRUMMER

Exotasks are a novel Java programming construct that achieve three important goals. They achieve low latency while allowing the fullest use of Java language features, compared to previous attempts to restrict the Java language for use in the submillisecond domain. They support pluggable schedulers, allowing easy implementation of new scheduling paradigms in a real-time Java system. They can achieve deterministic timing, even in the presence of other Java threads, and across changes of hardware and software platform. To achieve these goals, the program is divided into tasks with private heaps. Tasks may be strongly isolated, communicating only with each other and guaranteeing determinism, or weakly isolated, allowing some communication with the rest of the Java application. Scheduling of the tasks' execution, garbage collection, and value passing is accomplished by the pluggable scheduler. Schedulers that we have written employ logical execution time (LET) in association with strong isolation to achieve time portability. We have also built a quad-rotor model helicopter, the JAviator, which we use to evaluate our implementation of Exotasks in an experimental embedded version of IBM's J9 real-time virtual machine. Our experiments show that we are able to maintain very low scheduling jitter and deterministic behavior in the face of variations in both software load and hardware platform. We also show that Exotasks perform nearly as well as Eventrons on a benchmark audio application.

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

Additional Key Words and Phrases: Real-time scheduling, UAVs, time portability, virtual machine

ACM Reference Format:

Auerbach, J., Bacon, D. F., Iercan, D., Kirsch, C. M., Rajan, V. T., Röck, H., and Trummer, R. 2009. Low-latency time-portable real-time programming with exotasks. *ACM Trans. Embedd. Comput. Syst.* 8, 2, Article 15 (January 2009), 48 pages. DOI = 10.1145/1457255.1457262 <http://doi.acm.org/10.1145/1457255.1457262>

This work is based on an earlier work “Java Takes Flight: Time-portable Real-time Programming with Exotasks” published in *Proceedings of LCTES 2007*. It includes new work to unify Exotasks and Eventrons via a weak isolation model, revised and expanded programming examples, new empirical data, and more detail about the HTL Scheduler and the mapping between the HTL language and Exotasks.

Authors' addresses: email: josh@us.ibm.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2009 ACM 1539-9087/2009/01-ART15 \$5.00 DOI 10.1145/1457255.1457262 <http://doi.acm.org/10.1145/1457255.1457262>

ACM Transactions on Embedded Computing Systems, Vol. 8, No. 2, Article 15, Publication date: January 2009.

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 (RTGC) for Java [Bacon et al. 2003; Siebert 2004; Henderson 2002] have made Java suitable for writing real-time applications whose latency requirements are in the millisecond range. The Real Time Specification for Java (RTSJ) [Bollella et al. 2000] provides standard interfaces for creating real-time threads and mandates a fixed priority scheduler, while allowing for a JVM vendor to provide alternative schedulers. Problems remain, however. First, below some minimum latency, RTGC suffers from cache evacuation effects, throughput compromises, and work granularity limits. Thus, other ways of operating at very low latencies in Java are needed. Second, RTSJ does not standardize an interface between the scheduler and the threads and tasks to be scheduled, so only vendors are in a position to experiment with alternative schedulers. Third, real-time programs in the “portable” Java language are not time-portable because current technologies encourage the use of platform-dependent characteristics in reasoning about the application’s real-time properties. We will consider each of the issues (low latency, pluggable scheduling, and time portability) in turn.

1.1 Low Latency with Fewer Restrictions

Restricted subsets of Java, used in portions of a larger application, are currently being used to achieve lower latencies than what can be achieved with RTGC. These restricted models include the NoHeapRealtimeThread (NHRT) construct of the RTSJ, Eventrons [Spoonhower et al. 2006], Reflexes [Spring et al. 2007], and StreamFlex [Spring et al. 2007]. However, each of these restricted programming models entails problems for the application writer.

Exotasks provide a less restrictive capability for programming at low latencies, compared with the previously mentioned solutions. The key relaxation in restrictiveness is Exotasks’ ability to allocate objects in the knowledge that they will be garbage-collected only when unreachable. As a result, Exotasks are compatible with a larger (although still reduced) set of Java libraries compared to other approaches. This capability is important because so much programming in Java uses objects and most operations in Java end up allocating objects. For example, to iterate over a standard Java Collection, the program requests an `Iterator` object from the collection object. Since the iterator is stateful, the collection invariably allocates a fresh iterator to satisfy each request.

Reflexes [Spring et al. 2007] permitted at least temporary objects to be allocated in a transient region, which helps with objects that are almost always temporary, such as iterators. However, Reflexes require a bifurcation statically by class of all objects into stable and transient, and the stable area is not recovered until program termination. Thus, if some type (say, `String`) is sometimes allocated freshly, and sometimes stored on a nontransient basis, the programmer

must label `String` as a stable class, which results in a long-term storage leak. With Exotasks, none of these problems arise. Each Exotask has a private heap that is garbage-collected on a scheduled basis so that it does not interfere with task execution.

Like the other language restrictions with the exception of NHRTs, Exotasks rely primarily on program analysis (validation) for enforcement. A validated Exotask (as with Eventrons, Reflexes, or StreamFlex) will run without the need for expensive checks whenever object references are loaded or stored (as happens with NHRTs). Runtime checks consist only of the standard Java language checks (e.g., class casts, null pointers, array bounds) plus an added check on JNI callbacks to enforce certain Exotask restrictions for native methods. Like Eventrons, but in contrast to Reflexes and StreamFlex, validation is done at program initialization time when more information is available than is available at compile time. The Exotask model is a functional superset of Eventrons, allowing flexible communication with the rest of the Java application (a strength of Eventrons and Reflexes) to be traded-off against determinism and time portability (a strength of Exotasks).

Exotask restrictions consist of the following:

- (1) Exotasks may not observe or alter mutable state in static fields or in objects reachable from them in a way that would require synchronization with the garbage collector. At a minimum, they may not mutate references found in these fields and objects, but an even stronger condition may be enforced in the interest of determinism (as we discuss later).
- (2) Exotasks may not create new threads or use operations like `wait`, `notify`, and `sleep` that affect thread scheduling.
- (3) Exotasks may not use finalization or soft or weak references (which would introduce time variability into the private heap implementation).
- (4) Native code in Exotasks may not use those parts of JNI that would render the code validation meaningless (this is enforced dynamically since the code validator cannot examine native methods).

1.2 Pluggable Scheduling

In the Exotask system, it is possible to provide new timing grammars that define the rules for specifying timing constraints, and new schedulers that interpret those timing constraints (along with WCET information) to produce correct runtime behavior. Application writers will not typically create new schedulers, but we believe that there are scheduling experts who could do so and make the results available to application writers. While the RTSJ standard allows for alternative schedulers, there are no formal interfaces for installing schedulers while giving them access to the privileged VM and OS interfaces that they need to do their work. The Exotask system provides a privileged interface that schedulers can use to control all Exotask execution details, as will be described. Exotask schedulers can be written by third parties without knowledge of system internals.

1.3 Time Portability

Java provides functional portability across platforms, and it is naturally desirable for observable real-time behavior, also to 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 retuning the application. Today, this is not the case since platform-dependent characteristics are used to determine scheduling parameters when tuning the application.

Exotasks are able to solve the timing portability problem through a combination of essential features. First, they enforce a computational model in which tasks (the 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. Isolation makes use of the fact that each task has a private heap (which, as mentioned, is also useful in enabling the fullest use of Java). The actual isolation takes one of two possible forms, strong or weak. Strong isolation is “complete” in a sense to be described. Weak isolation allows for some nondeterminism in return for increased convenience in communicating with the rest of the Java application. Then, an appropriately designed scheduler achieves portable timing characteristics by scheduling Exotasks using Logical Execution Time (LET) [Henzinger et al. 2003]. 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 current schedulers used with the Exotask system are inspired by Giotto [Henzinger et al. 2003] and its successor hierarchical timing language (HTL) [Ghosal et al. 2006], but Exotasks represent a complete rethinking of the syntax of programs and the development cycle so as to fit naturally into the Java programming model.

1.4 Elements of the Exotask System

Exotasks comprise a programming model defined entirely within the Java language, a supporting tool suite built on the Eclipse framework [Eclipse Foundation 2007], and runtime support in a cooperating Java virtual machine. Like the other restricted models, Exotasks run on special threads that are exempted from preemption by system threads, such as those used to accomplish garbage collection or JIT compilation.

Garbage collection of the private heaps is generally scheduled on a heap-by-heap basis, at times when it will not interfere. Garbage collection of the public global heap occurs completely in parallel with Exotask execution. Exotask private heaps can also be collected “on demand,” but this increases variability in execution time and worst-case execution time. A discussion of the tradeoffs involved is presented in Section 3.7. We do not explore on-demand collections empirically in this paper.

The Exotask system requires a modified Java virtual machine (JVM) that is capable of enforcing Exotask memory isolation and provides the private heap mechanics and deep cloning support between private heaps. We have implemented such a modified Java virtual machine using IBM’s Websphere Real Time VM [Auerbach et al. 2007] as a base. The modifications are packaged as

an add-on to the product VM and can be obtained from IBM alphaWorks [IBM Corporation 2007].

The Exotask system supports pluggable system behavior, not only for scheduling, but for other facilities such as tracing and distribution across machine boundaries. Investigation of distributed Exotask programs is for future work, and distributors are discussed only briefly in this report. Pluggable tracing was used to collect the measurements in this report, but is not described. However, pluggable scheduling will be explored in that we will describe two different schedulers and their associated timing grammars in various examples to follow.

We first use a simple programming example to explain Exotasks basics. Then, we describe the Exotask programming model in full. Next, we present several versions of a control program for controlling a quadrotor helicopter called the *JAviator* that was built by members of our team. These more realistic examples complete our illustration of how Exotask programming proceeds in practice. Subsequent sections present some notes about the system extension mechanism and the implementation. In the measurements section, we present results from many of the Exotask programs presented earlier, as well as an Exotask realization of the audio example that was used in both the Eventrons [Spoonhower et al. 2006] and Reflexes [Spring et al. 2007] papers as a benchmark.

2. EXOTASK BASICS

In this section, we introduce the basic features of the Exotask programming model using a simple example of a controller for an inverse pendulum with one degree of freedom [Ogata 1997]. A more comprehensive exposition is then given in Section 3, after which, we explore the use of the Exotask system to build real control programs for the *JAviator* helicopter in Sections 4 and 5.

Although the inverse-pendulum example is well known, we describe it briefly. The device is a cart riding freely on a track. It has a pendulum rotating freely on one side through a 360-degree arc, but subject to gravity. A sensor reads the angular position and velocity of the pendulum, and another sensor reads the position and velocity of the cart on the track. A motor is capable of driving the cart in either direction with a specified force. The pendulum is started in a near-vertical position, and the controller must cause it to reach the exact vertical and track that position (to the extent feasible) while the cart is subject to perturbing forces that push it one direction or the other along the track. In this study, the inverse pendulum itself was simulated, not implemented in hardware, but the program is fully realized, and we present data from it in Section 8.

An Exotask program consists of a *specification graph* and *user code*, in the form of Java classes, for some graph nodes (in this example, all of the graph nodes). We discuss these elements in turn.

2.1 The Specification Graph

The Exotask specification graph for the inverse-pendulum controller is shown in Figure 1, which depicts a partial screenshot from our Eclipse-based Exotask programming environment.

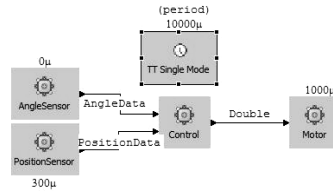


Fig. 1. An exotask controller for an inverse pendulum.

The nodes of a graph specify Exotasks and the edges specify strongly typed *connections* between the *ports* of those Exotasks. A single distinguished node (with the clock icon) selects a timing grammar (informally, a notation for expressing time, to be elaborated later). Timing annotations expressed in the selected grammar may be applied to any node or connection. In Figure 1, the timing annotations and connection data types are overlaid on the screenshot for clarity.

Exotasks may perform computation, I/O, or both. However, because the Exotask is the unit of scheduling, Exotasks are typically specialized as sensors and actuators (which have precise timing requirements) and compute tasks (which do not). That was done in this example.

The example employs the simplest timing grammar (the “time triggered single mode” grammar), which just assigns a period of execution (10ms), and then says when time-critical events should occur within the period. We have implemented two other timing grammars, which are discussed in Section 3.1.1. Note that the AngleSensor task is executed at time 0, and the PositionSensor at $300\mu s$. The Motor actuator task is executed at $1000\mu s$ (1ms). The times for these events were chosen to be realistic, and they will be used in Section 8 to demonstrate that determinism has been achieved. The Control Exotask and all of the data transfers between tasks will execute any time that is convenient based on data dependencies and execution time requirements. The strict isolation of this graph ensures that the scheduling of these activities does not matter as long as it does not interfere with the precise times assigned to externally visible events.

2.1.1 Connections and Ports. Communication between Exotasks is via directed edges (connections), which are strongly typed and are constrained to connect ports of matching type. The type assigned to ports and connections can be any Java class that is otherwise legal for use in an Exotask. Sharing between Exotasks is precluded because the operation of moving an object across a connection is semantically a deep copy of the referenced data structure.

Ports are not shown in the graphical view. By default, the number of input and output ports for each Exotask is implied by the number of distinct incoming and outgoing data types. For instance, in the inverse-pendulum example, the Compute task has two input ports and one output port. It is possible to have multiple input (or output) ports of the same type, though that case must be explicitly disambiguated by the programmer. An Exotask with no ports or connections is also perfectly legal and results in simple periodic execution (as in Eventrons [Spoonhower et al. 2006] or Reflexes [Spring et al. 2007]).

Connections are themselves stateless, but each port (input or output) 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. The scheduler is responsible for determining when this copy is made, and ensuring that the data movement does not overlap with the executions of the Exotasks at either end of the connection.

2.1.2 Sensors, Actuators, and Compute Tasks. When an application is partitioned into sensors, actuators, and compute tasks, the compute tasks are simply those Exotasks that do not perform I/O and have both input and output ports. Sensors are Exotasks that only have output ports and perform hardware input. Actuators are Exotasks that only have input ports and perform hardware output. Since adding direct hardware support to Java was beyond the scope of our effort, sensors and actuators typically invoke native (JNI) code to interface with device drivers. We will discuss this further in Section 2.2.

In the inverse-pendulum example, the `AngleSensor` Exotask provides its output port with an application-defined `AngleData` instance, containing both angular position and angular velocity. The `PositionData` type sent by the `PositionSensor` task is also application-defined, containing both position and velocity. Since the `Motor` actuator only needs a single double to determine the motor force, it uses the standard `java.lang.Double` type. Note that for generality, ports take Java objects rather than primitive types, so that scalars, like double, must be passed in their boxed form (boxing is automated in Java 5). Of course, the low-level hardware requires integers rather than double-precision floating-point numbers. But, since the sensors and actuators in the Exotask model are tasks with user-written code, the values seen by the rest of the program logic can be expressed in types and units convenient for the algorithm.

2.2 User Code

Obviously, what we have seen so far is not a complete program. Once the specification graph is complete, the remainder of the Exotask program is written in Java (plus, typically, a small amount of native code to reach the actual device drivers). Eight Java classes were written to complete the inverse-pendulum example. Two (`AngleData` and `PositionData`) are the data classes transmitted from the sensors to the compute task, as already discussed. Four others provide the implementations for the four Exotasks in the specification. One class defines a parameter (described in Section 2.2.1), and one is the main program that causes the Exotask graph to be instantiated and executed (described in Section 2.2.3).

Each class that implements an Exotask must have a zero-argument constructor. The class must also implement the Exotask interface, which defines a one-time initialize method and a periodic execute method. When the system instantiates this class, it will already have a private heap, so anything it creates in its constructor or initialize method will go there. The execute method will be

```

// Generated:
public class Controller implements Exotask {
    private ExotaskInputPort<AngleData> in0;
    private ExotaskInputPort<PositionData> in1;
    private ExotaskOutputPort<Double> out0;
// User Supplied:
    private ControllerParameters K;

// Generated:
    public void initialize(ExotaskInputPort[] inputs,
                          ExotaskOutputPort[] outputs,
                          Object parameter) {

        in0 = inputs[0];
        in1 = inputs[1];
        out0 = outputs[0];
// User Supplied:
        K = (ControllerParameters) parameter;
    }

    public void execute() {
        AngleData angle = (AngleData) in0.getValue();
        PositionData position = (PositionData) in1.getValue();
        double u = K.angleParm * (0.0 - angle.angle)
            + K.angularVelocityParm * (0.0 - angle.angularVelocity)
            + K.positionParm * (0.0 - position.position)
            + K.velocityParm * (0.0 - position.velocity);
        out0.setValue(new Double(u));
    }
}

```

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

called by the scheduler in each period of execution and must return (it is *not* like the run method of a Thread: in fact, there is no necessary mapping between threads and Exotasks).

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

If the Exotask program is developed using the Eclipse environment, a button click will create, for each Exotask, a partial implementation based on the specification. This consists of port instance variable declarations, a partially completed initialize method, and an empty execute method. The rest of the code is written by the user (as shown in Figure 2).

The initialize method's first few statements just assign arguments to instance variables to provide the running Exotask with its ports (which are created for it by the system). However, the parameter argument is up to the programmer to use as he sees fit. In this example, the parameter is expected to be of the application-defined type `ControlParameters` (one of the two additional classes

mentioned above). This is basically a data class whose four fields are shown in the third statement of the `execute` method. There are a variety of places that the parameter can come from, as we discuss more fully in Section 3.6. In this example, a definition of the parameter was provided as part of the specification graph (not shown).

2.2.2 Sensor and Actuator Details. The code of the three other Exotasks is not shown, but two observations can be made about these sensors and actuators: they are very simple (simpler than the `compute` task), and they have native code in order to communicate with the real hardware. Although we employed a simulation for this study, we still used native code to communicate with the simulation, for greater realism.

Native code in Exotasks must observe certain rules.

- (1) JNI methods may not call back into the JVM by invoking Java methods.
- (2) JNI methods may not modify reference-valued fields of objects (even not of objects they create), although they may modify fields of any primitive type.
- (3) If a JNI method returns a reference type, it must return that type exactly, not a subclass.

As a consequence of rule (1), JNI code cannot invalidate the reachability assumptions made by the validator. As a consequence of rule (2), JNI code can introduce new objects only by creating them, filling in their scalar fields, and returning them. As a consequence of rule (3), the types of all objects, thus introduced, are known. These rules guarantee that code validation will still be meaningful in the presence of nonmalicious native code that is free of memory usage bugs.

2.2.3 Startup. A conventional Java main program is needed to run an Exotask program. The key statements in the main program are the following.

```
ExotaskGraphSpecification spec = GraphGenerator.generateGraph();
ExotaskGraph graph = spec.validate("TTScheduler");
graph.getRunner().start();
```

The `GraphGenerator` class was completely generated by the development environment from the graphical form of the specification. It constructs an object that is the runtime representation of the specification graph. Section 3.8 discusses other ways of getting this object (including ones that do not use the graphical representation at all).

The `validate` method of the specification object actually does three distinct operations.

- (1) *Validation* determines whether all Exotask rules are followed by the graph and all of its code.
- (2) *Scheduling* analyzes the timing annotations and determines a schedule for executing the tasks, garbage-collecting the tasks, and moving data across connections.

- (3) *Instantiation* creates the graph and returns an object to represent it. This object has some methods of its own, and will return a subsidiary object (an `ExotaskRunner`) to perform other operations like `start`. This separation into two objects helps to support pluggable scheduling, as discussed later in this section.

Validation enforces the basic rules mentioned in the introduction and are considered in more detail in Sections 3.4 and 3.5. JNI methods are not validated since that is impractical; instead, we rely on dynamic enforcement in the JNI layer of the VM.

The scheduling step is performed by a scheduler that is fully pluggable and can (in principle) be written by any knowledgeable expert without detailed knowledge of Exotask system internals. Multiple schedulers can be configured to the Exotask runtime and are selected by name (the “time-triggered” or **TTScheduler** in this case). The scheduler can reject a specification graph whose timing annotations render it infeasible. The scheduler also provides the `ExotaskRunner` object mentioned above, to which it may add more methods depending on the semantics of the scheduler. Typically, the choice of timing grammar at development time constrains the choice of scheduler since the scheduler must understand the timing annotations.

In this example, no WCET information is passed to the scheduler, hence, scheduling is done without regard to WCETs. This is perfectly reasonable when there is an expectation of ample slack. An alternative form of the validate method (discussed in Section 3.6) accepts an additional parameter of type `ExotaskSchedulingData` to convey WCETs for the target platform. By conveying WCETs separately from the specification graph, we let the specification be platform-independent. It can be fully developed in terms of platform-independent requirements and WCETs need only be considered once a platform is selected. The specification graph can then be reused on other platforms by recomputing the WCETs.

The `ExotaskSchedulingData` parameter is also used to convey worst-case allocation (WCA) information used to schedule garbage collections and determine heap sizes. In the absence of provided WCA information, the scheduler uses a default heap size and makes a default assumption about allocation rate (both assumptions are reasonably conservative, but this clearly does not mean they will always be met). Having a default behavior allows an Exotask program to be put into test quickly. WCET and WCA information can be added later. In fact, the schedulers used in this paper do not make use of WCET information and only use WCA information in a fairly limited way to control resource consumption. Section 3.6.1 provides more details.

3. THE EXOTASK MODEL IN FULL

The initial example explained a number of things, but omitted others in the interest of getting through a complete example. In this section, we present the Exotask programming model in more detail, but without reexplaining terminology and semantics presented as part of the example.

3.1 Elements of the Model

The basic model establishes that programs are expressed as graphs, consisting of Exotasks, written in Java, and connections between them. In the absence of anything else, the graphs express only data-flow information, which is strongly typed. Timing annotations add information about when tasks execute and when data move across connections (concretely, the annotations attach to Exotasks, connections, or the graph as a whole). This information is intended to be platform-independent. Schedulers translate the data flow information plus the platform-independent timing annotations into potentially platform-sensitive schedules using optional WCET and WCA information. Timing annotations on a given graph conform to a timing grammar, which must be understood by the selected scheduler. Intuitively, a timing grammar is a set of syntactic well-formedness rules for attaching timing annotations to elements of the specification graph.

There are very 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. Cycles are permitted, but each timing grammar will impose rules on how cycles must be annotated to ensure schedulability.

There is no prohibition in the basic model against graphs that are not fully connected, including ones with multiple isolated subgraphs, or a collection of tasks with no connections. A timing grammar may prohibit some of these cases in the interest of schedulability. When the graph has no connections, allocation by the Exotasks of the graph may optionally be forbidden by the programmer, except for Throwable objects. In programs that only throw exceptions for termination, this allows all garbage collections to be avoided. The allocation prohibition option is unavailable when there are connections, since the deep copy semantics of a connection requires allocation to occur in the target Exotask's heap.

Although the Exotask system assumes it is running exactly one graph, we provide a graph composition operator that is capable of composing both related and unrelated graphs to form the executable graph. It is always the composed graph that is given to the scheduler. Composition is discussed in Section 3.3.

We discuss timing grammars in the next section. Scheduler responsibilities, and their plugin interface, are discussed in Section 6.1. Of our two existing schedulers, the specifics of the one scheduler have already been sketched. The alignment of our two schedulers with our three timing grammars is described in the next section. The HTL scheduler will be further described in Section 5.

3.1.1 Timing Grammar Details. This aspect of the system was made pluggable for two reasons. First, there are a number of different possible ways of specifying timing constraints. Second, we wanted to support pluggable schedulers since scheduling is an active research domain. Constraining the

semantics of timing annotations would inevitably constrain the schedulers as well. A consequence is that the Exotask programming model is parameterized by timing grammars, and consequently not limited to a particular timing semantics.

We have, so far, concentrated on timing grammars in the spirit of the logical execution time (LET) model introduced in Giotto [Henzinger et al. 2003]. LET specifies time for external events, but does not specify any timings for internal tasks. We believe the framework can also be used to provide radically different (non-LET) timing semantics but this category of usage has not been explored in depth.

We have, in fact, developed three timing grammars and two schedulers (all in the spirit of LET). Two of the grammars work with a time-triggered (TT) scheduler and so are called TT grammars. Both have the concept of a period of execution, which applies to the entire program during its current phase of execution. Within that period, tasks and connections are assigned timing offsets. A task or connection can have multiple timing offsets and so execute more than once in the period. In the TT single-mode grammar, used in the example already presented, one period declaration is in force during the entire execution of the program.

In the TT multimode grammar, the program's execution may be divided into modes. Each mode can have a different period, if desired. Tasks and connections belong to one or more modes, and only execute when those modes are active. At the end of each period, the scheduler may switch modes, causing a different subset of tasks and connections to execute thereafter (with perhaps a different period) until the next mode switch. The TT multimode grammar will be used in the JAviator controller.

Finally, we have a timing grammar which consists of a full injection of the HTL language [Ghosal et al. 2006] into Exotask timing annotations. This grammar (along with a JAviator controller that uses it) is presented in Section 5. A key improvement in the HTL grammar compared to the TT grammars is that it supports parallel composition and hierarchical refinement in addition to modes (which provide only sequential composition). The fact that these additional semantics can be supported in a straightforward fashion on top of the Exotask basic model shows the flexibility of that model in practice. The details of the HTL injection are discussed in Section 5.

3.2 Special Exotask Types

To support the inverse-pendulum example using the TT single-mode grammar, it was sufficient to have one homogeneous kind of Exotask. However, to support more sophisticated timing grammars, we introduce two special kinds of Exotask, called communicators and conditions. These will be used in the several JAviator control programs presented later.

3.2.1 Communicators. A communicator is a system-provided Exotask that has a single input port and a single output port of the same type and an execute method that copies its input to its output. One specifies a communicator in the specification graph by simply naming its type, and the system takes care of the rest.

At first glance, this feature (a buffer task that can be obtained without writing code) seems like a minor convenience. However, such tasks are frequently needed as the program becomes more complex and employs the more sophisticated timing grammars. For example, if the program will switch modes, then it is valuable to keep some state at the end of every period in a place that will be accessible to the tasks in the next mode (which may, in general, be an arbitrarily different set of tasks). Also, in grammars like the HTL grammar that support parallel composition, the reconciliation of simultaneously executing subgraphs with different and potentially inharmonic periods becomes impossible without buffering. The term “communicator” comes from HTL and communicators are used in a specialized fashion by the HTL grammar as will be described in Section 5.

3.2.2 Conditions. A condition is a user-written Exotask with an extra method that can be invoked by the scheduler, returning a boolean value. Conditions have a role assigned by the timing grammar in letting the program affect the schedule. The TT single-mode grammar does not use conditions but the other two grammars use them to let the program determine when a mode switch should occur.

3.3 Graph Composition

Just prior to submitting a graph to validation, a series of separate specification graphs can be combined to form a single one. Syntactically, the method

```
ExotaskGraphSpecification.composeWith(ExotaskGraphSpecification
                                     other);
```

is executed any number of times. Each execution produces a composed graph without mutating either of the original graphs. The semantics of composition is straightforward: if the graphs are unrelated, this should be expressed by not using the same names for any Exotasks in either of the two graphs. If they are related, then the same name is used in both for one or more Exotasks: these are equated to each other in the resulting graph (only one task of that name will exist). If the specification of the same task in two composing graphs differs, the composition fails. The connections of the composed graph are the union of those in the original graphs, except that connections from different source graphs that connect like-named ports of like-named tasks (the same ports and tasks after composition) become a single connection.

As each of the graphs will have its own timing annotations, all graphs being composed must use the same timing grammar and the timing grammar implementation participates in the composition. We will see examples of graph composition in Sections 4 and 5.

3.4 Exotask Memory Isolation

We provide both a strong- and a weak-isolation model for Exotasks. These have advantages and disadvantages, as will be described in the subsequent sections. First, we state properties that are true in both models.

- The ports of an Exotask, which are system-generated, are passed to its initialize method. The Exotask gets no constructor arguments, and its initialize arguments are controlled by the system. Thus, it begins with no references at all, except to its ports, to its parameter, 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. Whether or not its parameter resides there depends on the isolation model.
- 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.
- 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 the chosen isolation model through accessing static fields. This is enforced by the validator as was briefly discussed in Section 2; details on the treatment of static fields is given in Section 3.4.3 and differ between the two models. Once an Exotask is validated, it is allowed to execute without dynamic checks (except for JNI callbacks which are checked to make sure they do not violate the special rules for JNI methods).

3.4.1 Strong Isolation. In a strongly isolated Exotask graph, there is no possibility of intercommunication between the Exotask graph and any other part of the Java program in which it lives. Furthermore, there is no communication between the individual tasks in the graph except by their explicit connections. This is accomplished by adding these additional restrictions to what was listed as common to both models.

- The parameter passed to each Exotask is deep-copied into the Exotask heap. Therefore, the Exotask starts with no references to anything outside itself (recall that, in both models, there are no pointers from outside to inside). The values that are visible to the Exotask reside only in its private heap.
- No static field may be written and, if a static field is read, the value that is read must be (recursively) immutable in a sense described in more detail in Section 3.4.3. Logically, the Exotask program receives a deep copy (physically, the deep copy is avoided as discussed in Section 3.4.3), so even synchronization on this object will not interlock with anything outside the Exotask graph.

The strong model guarantees time portability (given adequate resources, i.e., assuming the system is time-safe [Henzinger and Kirsch 2007]). More formally, strongly isolated graphs are environment-determined [Henzinger and Kirsch 2007]: their behavior is entirely determined by the values and timings of sensor values read in native code in Exotasks.

However, we have mentioned that an Exotask program exists within a larger Java main program that starts it up. The guarantees of the strong model come at the expense of this larger program not being able to communicate with the Exotask program (other than to pause it, shut it down, or diagnose it if it fails). If the program was developed entirely from scratch using Exotasks (or a modeling system like Simulink [Simulink 2007], that is conceptually compatible), this restriction may be no problem, as the Exotask program can do everything that is required. The inverse-pendulum example of the previous section uses strong isolation, and we will use it in Section 8 to show complete repeatability of behavior.

3.4.2 Weak Isolation. In some realistic scenarios, the larger Java program may be performing non-time-critical work but needs to interact with the Exotask program. In that case, weak isolation provides the required mechanism. Under weak isolation, some objects are accessible both to the Exotask program and to the rest of the Java program. Both parties may freely modify scalar fields (including elements of primitive arrays) in these objects and hence communicate with each other. However, they cannot change any reference fields in the shared objects; hence, the membership and connectivity of the shared set cannot change. The objects are pinned in the global heap so that they cannot move. Consequently, this object sharing cannot affect the independence of the Exotasks from the main heap garbage collector.

The insight that a pinned object set with unchanging connectivity can be safely shared between independently garbage-collected threads originated with Eventrons [Spoonhower et al. 2006], and was also exploited by Reflexes [Spring et al. 2007], which gave the name reference immutability to the property of unchanging connectivity (we will use that term here). By “reference-immutable” we mean that the pointers between objects cannot change even though scalar values stored in the objects may change. The JAviator control program described in Section 4 uses weak isolation to accomplish certain goals that would have been difficult to accomplish otherwise. In an earlier publication on this work [Auerbach et al. 2007], weak isolation was not available, but we got around the problem by exploiting the distributor interface (see Section 6.2). Distributors can bridge the isolation boundary but are hard to write (and impossible to completely verify) and so are really system-level components. The addition of weak isolation was needed to make the programming model complete without requiring the dangerous use of system interfaces for application purposes.

Mechanically, the enforcement of weak isolation differs from strong isolation as follows. To make the operation clear, we distinguish between a weakly isolated (individual) Exotask and a weakly isolated Exotask graph (or program). The presence of any weakly isolated Exotask in a graph makes the entire graph weakly isolated.

—The parameter passed to each weakly isolated Exotask is checked for reference immutability. If it is not reference-immutable, validation fails. If it is, the Exotask receives a reference to it (not a copy). Thus, a weakly isolated

Exotask does have references to objects outside itself. However, there are still no references in the opposite direction and no unanticipated references in either direction can be created once execution begins. Also, deep copying continues to preclude the introduction of any additional aliases.

- On the other hand, there can also be strongly isolated Exotasks in the same graph that are treated just as in strong isolation (parameters are cloned). This permits control over the handling of parameters on a task-by-task basis as needed.
- Although no static reference field may be written, a static scalar field may be written. If a static field is read, the value that is read must be (recursively) reference-immutable (not necessarily fully immutable). And, an actual reference to a shared object is read, not a logical copy as with strong isolation. The details of static-field handling are covered more fully in Section 3.4.3.

In gaining these communication advantages, a weakly isolated Exotask program is no longer environment-determined and so we cannot guarantee that it is time-portable. Three sources of possible variation are problematic.

- (1) If synchronization occurs on a shared object, the timing of the Exotask program may be delayed. Synchronization may be optionally forbidden at validation time as described in Section 3.4.3.
- (2) Under the Java memory model, if synchronization is not used, then it is often not determined when a modification by one thread becomes visible to another. In practice, weakly isolated programs should be constructed on the assumption that such values *eventually* become visible and are not relied on for anything time-critical.
- (3) If the program makes extensive use of library code, then it may not know what static fields are actually read. Therefore, execution may depend on things that are not strictly controlled. In practice, all Exotask programs need to be cautious about the code they use in proportion to how badly they require deterministic execution.

When these three factors are considered and carefully controlled, we believe that weakly isolated Exotask graphs are still adequately time-portable in practice for many purposes. When a stricter model is needed, the strong model is available.

3.4.3 Use of Static Fields by Exotasks. 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 preexisting 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 the 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`.

In Auerbach et al. [2007], we greatly expanded the amount of usable library code with a two-pronged solution. First, we added a more powerful analysis which was able to detect that many objects were either recursively immutable or not accessible to mutating code. Second, in the case of objects that are, in fact, never mutated but for which the analysis is still not sufficiently powerful to discover the fact, the Exotask system (but not the user!) can specify some classes as “known to be immutable.”

The automated part of this solution requires a practical recursive definition of immutability. We first define a field as effectively final if it is either declared final or it is both declared private and not mutated by any nonconstructor method of the class. We then define a recursively immutable field as an effectively final field that is of a primitive type or null, or contains a recursively immutable object. A recursively immutable object is defined as one that has only recursively immutable fields.

In the present work, we add sensitivity to whether the Exotask graph is only weakly isolated. For that case, we define reference immutability in a similarly recursive fashion. A reference-immutable field is any field of a primitive type or an effectively final reference field that is either null or contains a reference-immutable object. A reference-immutable object is defined as one that has only reference-immutable fields.

We can then state the rule to be checked simply: a strongly isolated Exotask program may read only recursively immutable static fields and may not write any static fields. A weakly isolated program may read only reference-immutable static fields and may write only static fields of a primitive type.

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.

Second, the appropriate degree of isolation must be preserved in the face of synchronization operations on the object. In the strong-isolation model, the runtime system simply ignores locking operations by the Exotask as if the Exotask had a private copy. This is both safe and semantically invisible because the object is necessarily immutable, the Exotask is single-threaded, and any Exotask code which invokes wait or notify is rejected by the validator. The result is just as if these objects were copied at the moment of access by the Exotask.

In the weak-isolation model, synchronization on pinned heap objects is allowed, unless the user requests that synchronization be checked for at validation time and explicitly disallowed. Because code analysis does not have precise object identity information, this check is necessarily conservative.

3.5 Validation Details

When the Exotask validator runs, it uses rapid type analysis (RTA) [Bacon and Sweeney 1996] to build a summarized call graph rooted in the constructors, the initialize methods, and the execute methods of the Exotasks in the graph. This

results in a conservative closure of the reachable methods. It examines every bytecode of every reachable method.

As in the Eventron analysis, the validator also maintains the set F of field signatures (static or instance) found to be referenced (for reading or writing) in any method, and the set O of objects residing on the global heap but accessible to Exotask code. Initially, the set O consists of objects passed as parameters to weakly isolated Exotasks (it is initially empty if all Exotasks are strongly isolated). Whenever a field signature is added to F , the validator considers objects in O that contain a matching field. The referents of such fields are added to O . Whenever an object is added to O , it is inspected for fields that match F . Thus, an addition to either set can increment both sets up to a fixpoint.

As each bytecode is examined the following actions are taken. Note that some actions depend on whether isolation is strong or weak, and whether the user has elected to make allocation illegal or synchronization illegal.

- `invoke` operations: the summarized call graph is expanded as called for by RTA. In addition, if the invoked method is native, its return type is added to the live classes used to inform RTA in its search for live methods. Finally, methods from the JDK whose behavior is known, and that are manifestly illegal (e.g., `Object.wait()`), cause immediate validation failure. Methods are manifestly illegal when they violate one of the rules, according to the chosen isolation model and the optional prohibitions against allocation and synchronization.
- `getfield` or `putfield`: the set F is augmented as described above (this can cause the set O to be augmented as well, since all objects already in O that implement F will have their referents added to O).
- `putstatic`: checked for legality according to the isolation model (illegal if strongly isolated or if the target is a reference field).
- `getstatic`: checked for legality according to the isolation model (must be a recursively immutable field for strong isolation or a reference-immutable field for weak isolation). In addition, the object read (which can be determined since this analysis is performed after all class initialization is complete) is added to the set O and then processed as described above.
- `monitorenter`: examined only if synchronization is forbidden at the user's request, in which case it causes an exception.
- `new`: if allocation is forbidden at the user's request, this causes an exception unless the object being allocated is a `Throwable`. Otherwise, validation fails only if the class is in the forbidden set (classes that inherit from `Thread` or `Reference` or having a nonempty `finalize` method). In any case, the set of live classes is augmented as called for by RTA.
- `newarray` and variants: if allocation is forbidden, an exception is raised.

As objects are added to O (regardless of whether this was the direct effect of a `getstatic` or the indirect effect of examining objects made reachable by a `getfield` or `putfield`), they are tested for conformance to the isolation rule (recursive immutability for strong isolation, reference immutability for weak isolation).

Violations cause exceptions. Because of the recursive definitions, this might sound like an expensive operation. In practice, however, objects need only be examined “shallowly” as they are entered into O ; objects that cannot possibly be accessed need not be examined. Although determining that a field is effectively final requires examining all of the methods, this only needs to be done on class basis (since all objects of the same class share the same methods), so, in practice, this cost is saved when many objects in O are of the same class.

At the end of the validation, all objects in O are pinned. The set of objects in O are linked by effectively final reference fields. For strongly isolated Exotask graphs, this set of objects does not contain any mutable fields whatsoever. Thus, the requirements of either isolation model are met.

This analysis is powerful enough to admit Exotask usage of most classes in `java.util`. For example, it successfully validates the `HashSet` class for use by strongly isolated Exotasks. The only static object in `HashSet` 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`.

3.5.1 Immutability Declarations. There are some cases where the data-sensitive analysis is still not powerful enough to determine that a static object is in fact immutable or reference-immutable. This occurs with the class `Integer`, which caches boxed versions of integers smaller than 256 bytes 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 any Exotask.

To handle this case, the Exotask runtime system maintains a table of fields (class names and the names of fields therein) that are to be treated as immutable without requiring analysis. Note that a field may not be declared effectively immutable unless it really is immutable: this is an augmentation for analysis but not for cases where a runtime mutation is “unlikely.” Thus, the table is maintained as a resource in the boot classpath, where it can be edited by a responsible system maintainer but not modified casually by applications.

The current Exotask prototype, running on the IBM virtual machine, so far requires 30 immutability declarations applying to 11 classes in the JDK. Note that the Java 5.0 JDK contains over 4,000 classes, of which at least 400 are loaded by any application that makes serious use of JDK facilities. Our prototype’s immutability declarations mostly apply to classes that do math or number-to-string conversion using tables. The list is not necessarily exhaustive, since we may not have exercised all code paths in the Java class libraries, but it has remained stable over a considerable period of experimentation.

3.5.2 Impact of Exotask Restrictions. Other than the restrictions explicitly checked by the validator, 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.

3.6 Validation Time Inputs

The abbreviated syntax `validate(String schedulerName)` used in the inverse-pendulum example is just one of several short forms of the full validator syntax, which takes four inputs. One is the scheduler name. One optional input (already mentioned) is the `ExotaskSchedulingData` with platform-dependent information: this is further elaborated in Section 3.6.1. Another optional input is a map from Exotask names to objects that are to be passed to them as parameters. This provides a more dynamic alternative to storing Exotask parameters in the specification graph (as we did in the inverse-pendulum example). Runtime specification of Exotask parameters is needed in practice when parameters are to be passed by reference (instead of by copy) to weakly isolated Exotasks. The final input relates to the distributor facility which is briefly described in Section 6.2.

3.6.1 Platform-Dependent Scheduling Data. The `ExotaskSchedulingData` class encapsulates the following information items (schedulers may also subclass this class to add more scheduler-specific information).

- An initial storage limit applying to the entire program. The sum of the heaps assigned to individual Exotasks may not exceed this size.
- A maximum storage limit. If this is greater than the initial value, heap expansion is allowed in increments that pragmatically partition the headroom between the initial and maximum size.
- For each Exotask,
 - the WCET of the Exotask’s execute method;
 - the WCA of the Exotask’s execute method (in bytes);

- the WCET of the garbage collection of this Exotask's heap;
 - the maximum live objects aggregate size of the Exotask (in bytes) while its execute method is running;
 - the maximum live objects aggregate size of the Exotask (in bytes) while its execute method is not running.
- For each connection, the WCET of the deep-copy operation over that connection.

It should be emphasized that not all of these items need to be filled in, and, in fact, no `ExotaskSchedulingData` need to be provided at all (it was, for example, omitted in the inverse-pendulum example). Developing a program with Exotasks can be done in a very disciplined fashion, with WCET and WCA information gleaned analytically, when the individual Exotasks are sufficiently simple (tools to support such an analytic approach are beyond the scope of our project). On the other hand, one of our purposes in supporting real-time programming in Java is to support more complex programs, which are necessarily harder (or impossible) to bound analytically, and which may require somewhat more capable computing hardware. Fortunately, embedded computers are becoming more capable, and so it is feasible to do quite a bit of development using assumptions of ample slack and generous defaults built-in to the system. Later, observations of the program's actual behavior, along with some analysis, can produce probabilistically firm WCET and WCA values that work for practical purposes in all but the most safety-critical programs.

In our current TT scheduler, the storage-related information is used to compute heap sizes and garbage collection frequencies for individual Exotask heaps. At present, this is always done in a way that avoids the need for on-demand garbage collection (see Section 3.7), and no meaningful use is made of heap expansion (which will involve still other tradeoffs). Empirical investigation of these possibilities is subject to future work.

None of our current schedulers use WCETs in computing schedules, so, in a sense, more sophisticated scheduling is also future work. However, what we believe we have contributed is a well-founded framework for experimenting with scheduling. The public availability of this framework [IBM Corporation 2007] is expected to stimulate research by others as well as ourselves in this area.

The Exotask system also does not guarantee that a task cannot exceed its WCET or its WCA: it simply trusts these values. If the WCA is exceeded, then an on-demand garbage collection will happen even though this was not anticipated. This may in turn cause a WCET violation. A WCET violation will generally cause subsequent task executions to be delayed (although the details are ultimately up to the scheduler). We currently do not offer to schedulers the ability to abort tasks that have exceeded their WCET.

3.7 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.

As was discussed in Section 3.6.1, an Exotask has storage-related parameters like WCA as well as WCETs. 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 bit) 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 [Pillai and Shin 2001].

While on-demand collections may seem risky to programmers used to older methodologies based on static memory allocation, from a scheduling perspective they are not 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.

3.8 Alternate Ways to Obtain the Specification Graph at Runtime

In the previous section, we showed the runtime specification graph being produced by a `GraphGenerator` class. The development environment indeed has a utility which produces generator classes as in the example. Under the covers, the generator class employs a runtime API that is capable of constructing any valid (and many invalid) Exotask specification graphs programmatically. This API may be used directly, an option that is quite suitable for small programs, although it is tedious for large ones. We do not describe it in detail here, but details are available in [IBM Corporation 2007].

In addition, the development time environment will save a graph as a particular form of XML document and there is a runtime parser that will read this document from a variety of media to reconstruct the specification graph at runtime. This means that, even for complex programs, it is possible for tools other than our development time environment to produce a convenient and somewhat human-readable expression of the specification graph. The details of this format are also available in IBM Corporation [2007].



Fig. 3. The JAviator: a custom-built quadrotor helicopter used for experiments in this article (javiator.cs.uni-salzburg.at).

3.9 JIT Interference

A JIT compiler can indeed interfere with deterministic execution, both by altering the execution times of methods and by taking up processor cycles doing the compilation. The strongest means we have of combating this problem is to use the ahead-of-time compiler [Fulton and Stoodley 2007] that comes with the IBM VM. Unfortunately, this only works on the Linux x86 platform and so is not suitable for all embedded applications. Another technique is to run the Exotask program in a “warm-up” mode for a while after which the JIT is disabled from further compilation. This is feasible for many control programs but is application-specific. In the case of the JAviator control program, which runs on an XScale PXA270 processor, there is no JIT in our port of the VM to that architecture. Thus, although execution is slowed thereby, determinism is not compromised. Finally, it is easy in principle to force early JIT compilation of all the methods in the Exotask call graph because the complete call graph is known at instantiation time. We have not attempted to do this since the problem is solved for us in other ways on both x86 and XScale.

4. ADVANCED EXOTASKS: THE JAVIATOR

Now that the full Exotask model has been explained, we can look at how the control program for the JAviator is constructed. For explanatory purposes (somewhat following the actual evolution of the program but not precisely), we will start with a very simple version and then move from it to the actual program that we use.

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 roll, pitch, 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.

Currently, the processor is an XScale PXA270 on board the JAviator, which runs a scaled-down Linux with some real-time patches. The medium of transmission is an RS-232 connection between a UART on the XScale and one on

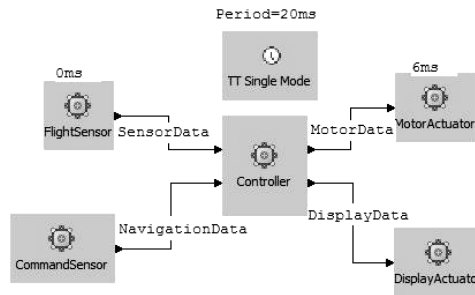


Fig. 4. A (Too) simple control program for the JAviator.

the microcontroller. The microcontroller does no significant computation. It is needed because the XScale processor cannot directly produce PWM signals.

The Exotask-based control also communicates via a UDP socket with the 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. By design, the medium of transmission is WiFi, so that the JAviator will have no wires attached. Currently, problems with power distribution for the WiFi within the JAviator cause us to use an Ethernet cable in practice. This perturbs the flight somewhat and so we still fly the JAviator in a confinement cage for safety.

Figure 4 is the simplest possible control program for this hardware. It is presented in the same style as our first example and looks very similar. Four custom data classes encapsulate information that must flow from the JAviator’s flight sensors to the controller (*SensorData*), from the controller to the motor actuators (*MotorData*), from the joystick of the ground station to the JAviator (*NavigationData*), and from the controller to the instrument panel of the ground station *DisplayData*. There is a period of 20ms, during which the reading of the flight sensors takes place at time 0 and the sending of motor signals at time 6ms. No other task is precisely timed: even the sensor and actuator for communication with the ground station are not timed: information to and from the ground station travels over a network connection and hence arrives asynchronously: it is impractical to wait for it in any case, so the controller relies on the fact that usually a ‘fairly up-to-date’ navigation command is available.

An adequate controller could be written to populate the single Controller task in Figure 4, but the code of the Controller would be quite complex. We noted that the code actually divides into four distinct tasks that use somewhat different algorithms: (1) resting on the ground, revving up the motors and waiting for take-off (or revving down the motors if landing), (2) changing altitude once airborne, (3) hovering, and (4) responding efficiently to loss of control by attempting an emergency landing. The periods that make sense for these tasks might be different (although, in practice, we are currently running them at the same period) and, in any case, the controller logic is really quite distinct. Consequently, we decided to divide the JAviator controller into four modes, and use

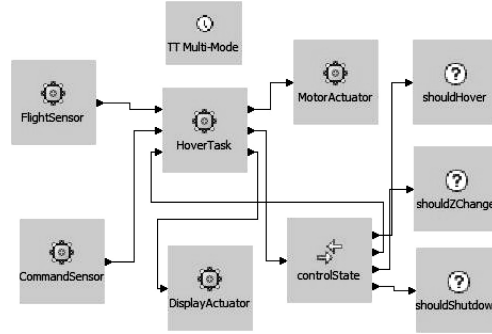


Fig. 5. The Hover Mode (One of four modes).

the TT multimode timing grammar. The graph for the hovering mode is shown in Figure 5. In Auerbach et al. [2007], the four modes were shown already composed, which gives a sense of the program’s full complexity but does not aid understanding and so we omit that view here. To avoid clutter, the timing annotations and data types are not overlaid in this figure; they are the same as in Figure 4 except as noted.

Note that the same four I/O tasks exist and the central compute task (now called `HoverTask`) is still connected to them in the same way. However, in a multimode program, it is no longer possible to keep all the states related to the ongoing control problem in a single Exotask, because the mode may switch at any period boundary. Consequently, the `HoverTask` writes out the current state at the end of each execution and reads it back in for the next. A communicator called `controlState` is used for this purpose (see Section 3.2.1). The new form of the program also introduces conditions (Section 3.2.2), one for each of the modes that this mode can switch to. These are evaluated at the boundaries between periods and the appropriate switch becomes true when the conditions of the new mode are met. When programming in this way, it is important that the conditions be mutually exclusive and that at most one of them be true (we have chosen a style in which even staying in the same mode is reflected by a positive condition: this is a useful programming discipline but not strictly necessary). As can be seen, communicators and conditions are visually distinguished by different icons.

The data types flowing on the links between the I/O tasks and the compute task are the same as in Figure 4 and the timing annotations are nearly the same. Instead of executing the `FlightSensor` at time 0 and the `MotorActuator` at time 6 ms, we use 5 ms and 11 ms (the external behavior is the same since no external process actually observes the period boundaries). This offsetting of the time allows ample time for the mode switches to be computed so that we can get the fairest possible measurements of the system’s performance (some further improvements to the TT scheduler would obviate the need for this adjustment). The data type flowing to and from the `controlState` communicator is of type `JControlAlgorithm` and contains data and methods relating to the overall control state.

The graphs for the other three modes are similar: all four modes have a central compute task, four I/O tasks, the `controlState` communicator, and the conditions necessary to switch to other modes. All four have the same global timing annotations specifying that there are four modes and giving their periods. When the four graphs are composed, the result has only one instance of each I/O task (since all are identically named), one instance of the `controlState` communicator (thus accomplishing communication of state across modes), and one instance of each condition. However, it has all four compute tasks, one for each mode. This program will accomplish the same goal as the one shown in Figure 4 but with key differences. First, the code specific to individual Exotasks is substantially simpler, making reasoning about WCETs much easier. Second, although we do not currently exploit this fact, different periods could be easily assigned to different modes. Third, as the specific algorithms for each mode diverge in detail, there are no nonobvious interactions within the program logic that need to be understood.

Not shown explicitly in Figure 5 is that this version of the control program also relies on weak isolation. For safety, the `shouldShutdown` condition, which is evaluated in every mode, must learn as soon as possible of a shutdown signal caused either by a message from the ground station or by various loss of connectivity conditions that are detected outside the Exotask program. Consequently, that predicate and the `CommandSensor` task both receive a pointer to an object on the public heap that contains the latest information aggregated from the ground station and from various other watchdogs in the program. This area always contains the latest command when needed by the `CommandSensor` and the latest state of the shutdown flag.

5. HTL AND EXOTASKS

In this section, we describe how the semantics of the HTL language [Ghosal et al. 2006] is effectively injected into the Exotask system by virtue of implementing a timing grammar and scheduler that reflects those semantics. We also present an alternative form of the control program for the JAviator written first in HTL and then translated to Exotasks, using the features of the HTL timing grammar and scheduler.

The HTL scheduler and its supporting grammar contrast with the TT grammars and scheduler in its support of higher-level real-time programming abstractions, in particular, (multithreaded) parallel task composition and hierarchical task refinement. We thus demonstrate that the Exotask system is sufficiently general to support more advanced semantics such as HTL's semantics. Nevertheless, much is left to the scheduler and the timing grammar in such a system. However, when, as in the present case, the timing grammar and scheduler are drawn from previous research and proven techniques, the result is to marry such results with the full power of the Java language, which was one of our goals.

HTL has two key features: tasks can be composed in parallel without changing their individual I/O timing behavior (logical execution time) and abstract tasks can be refined by concrete tasks without losing schedulability. An abstract

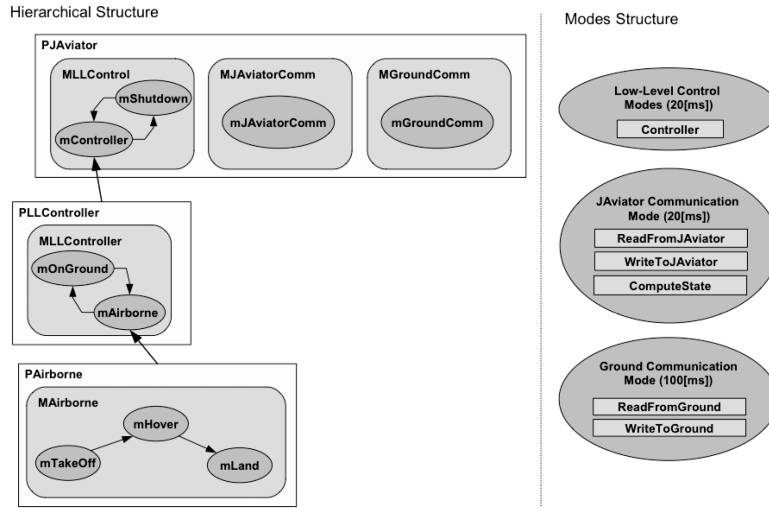


Fig. 6. The HTL program structure of a JAviator flight controller.

task is a placeholder for concrete tasks that refine the abstract task, that is, have less stringent timing requirements than the abstract task (less WCET, later deadline, earlier release, fewer precedence constraints). If an abstract HTL program is schedulable, then any concrete HTL program that refines the abstract HTL program is also schedulable [Ghosal et al. 2006]. In general, checking refinement and schedulability of abstract HTL programs is exponentially faster than checking schedulability of concrete HTL programs. Note that concrete HTL programs may contain mode switches that select different sets of concrete tasks, which may describe exponentially many combinations of concurrently executing concrete tasks, as long as the tasks in each combination uniquely refine abstract tasks. The schedulability test on abstract HTL programs is a sufficient but not a necessary condition since there might be unschedulable abstract HTL programs that are refined by schedulable concrete HTL programs.

5.1 HTL Grammar Annotations

In the following, we give an overview of how HTL programming primitives are specified in an Exotask graph by means of an example program, which implements the altitude and attitude (i.e., roll, pitch, and yaw) control for the JAviator. The inputs of the four controllers are the altitude, roll, pitch, and yaw sensed and target values, respectively, and the output is the thrust that has to be produced by each of the rotors. The control program implemented in this example can be seen as the low-level part of a two-level control structure, the high-level part being a position controller (i.e., control of the x- and y-dimension). So far we have only implemented the low-level part, the high-level part is future work.

Figure 6 depicts, in visual syntax, the HTL program that implements the above low-level control. The program consists of running in parallel functionality that implements the low-level control as well as functionality that

implements the communication with the JAviator and with the ground station. In HTL, parallelism can be specified through parallel HTL modules, which consist of HTL modes, which are sets of periodic, abstract and concrete *HTL tasks* and some mode-switching logic. In each module, exactly one mode can be active at any time. A module also specifies a unique start mode. All tasks in a mode have the same period, and all modes in a module, except top-level modules, have the same period. An HTL mode may be refined by an HTL program, which may then only contain modes with the same period as the refined mode. In an Exotask graph the hierarchical structure of an HTL program (i.e., all the subprograms, modules, and modes and all relations between them) are specified as global timing annotations.

In the JAviator low-level control example, the top-level program contains three modules, namely, MLLControl, MJAviatorComm, and MGroundComm. The MJAviatorComm module specifies the timing of tasks that implement communication with the JAviator. It also computes the next state for the altitude and attitude controllers. The module consists of a single mode, which has a period of 20ms. The mode invokes the ReadFromJAviator, WriteToJAviator, and ComputeState tasks. The MGroundComm module specifies the timing of tasks that implement communication with the ground station. This module also contains only one mode that has a period of 100ms and invokes the ReadFromGround and WriteToGround tasks. Module MLLControl contains the mController and mShutdown modes, which both have a period of 20ms. The mShutdown mode specifies the timing of the emergency shutdown, while the mController mode specifies the timing of the altitude and attitude controllers. The mController mode invokes the Controller task, which implements the altitude, roll, pitch, and yaw controllers.

The Controller task is refined by two tasks in the HTL program PLLController of which one task is a concrete task that is invoked in the mOnGround mode and the other one is an abstract task that is invoked in the mAirborne mode. The abstract task is further refined in the HTL program PAirborne by three other concrete tasks, one for each of the three possible states of a flying helicopter (i.e., take-off, hover, and land). In an Exotask graph that uses the HTL timing grammar, HTL tasks are mapped directly to Exotasks, which are annotated by the mode name to which the original HTL task belongs, and by the name of the parent task if the task is a refinement of another task. The mode switching logic of an HTL program is mapped to conditions (see Section 3.2.2), which are evaluated through the standard scheduler interface. The annotation of a condition specifies the modes in which the condition is defined, and the mode to which the condition switches when it evaluates to true.

Figure 7 shows the data flow between tasks within and across modules in the top-level program. In the example, the data-flow structure on the top level is maintained by the refinements in the rest of the program. HTL tasks in the same mode can communicate with each other through HTL ports. HTL ports, therefore, potentially induce precedence constraints on task execution in the same mode. Although there is no direct counterpart for an HTL port in an Exotask graph, dependency relations can still be specified through

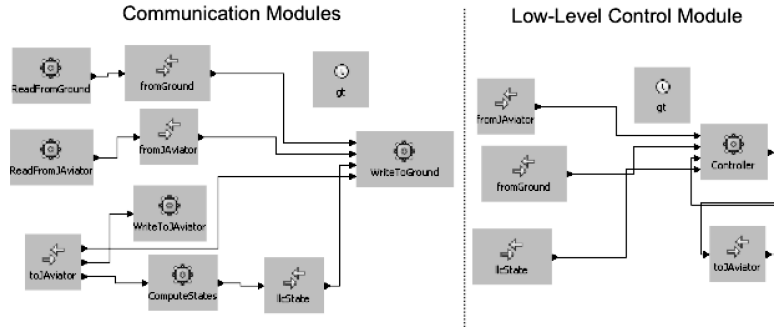


Fig. 7. Data-flow view of the top-level HTL program in Figure 6.

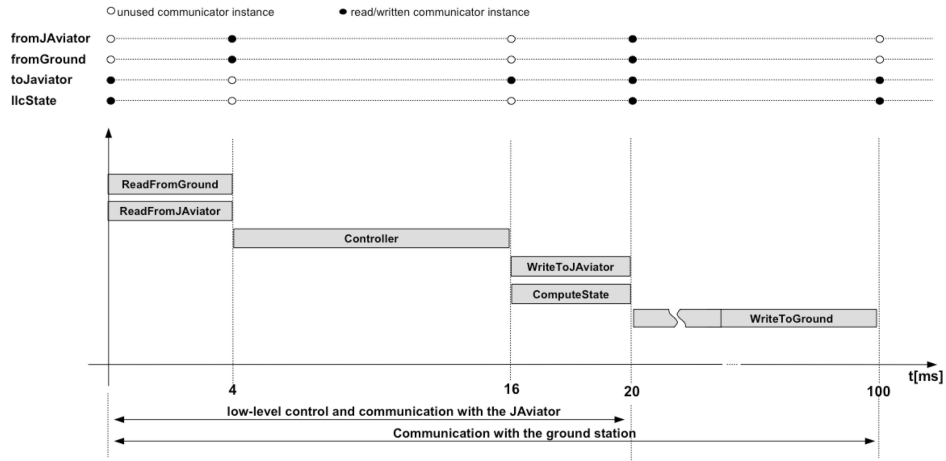


Fig. 8. Timing view of the HTL program in Figure 6.

connections between Exotask output and input ports. Tasks in different modules possibly running at different frequencies can also communicate but only through HTL communicators, which need to be declared in HTL programs. For example, PJAviator declares the communicator fromGround, which is used to communicate data between the ReadFromGround task in the MGroundComm module and the Controller task in the MLLControl module. HTL communicators are mapped directly to Exotask communicators, which are annotated by the program name to which the original HTL communicator belongs, and its period. Communication between tasks and communicators are mapped to connections between Exotask ports. Connections that connect Exotasks corresponding to HTL tasks to communicators are annotated by an instance number, which determines the communicator period (harmonic to the task mode's period) at which communication to or from the communicator is done.

Figure 8 depicts the timing of the HTL program. Tasks invoked in the MLLControl and MJAviatorComm modules are executed once every 20ms, while tasks in the MGroundComm module are executed once every 100ms. HTL communicators also have periods and can only be read and written at the

beginning (end) of their periods similar to Exotask communicators. For example, the *fromJAviator* communicator has a period of 4ms and is updated by the *ReadFromJAviator* task at the time instant that corresponds to the second instance of *fromJAviator* with respect to the 20ms period of the task. The same instance will then be read by the Controller task. Writes are always done before any reads. Multiple writes to the same instance are not permitted to avoid race conditions. The choice of communicator instance must be done at compile time, and requires, in this example, that the *ReadFromJAviator* task must complete execution before the second instance of *fromJAviator*, while the Controller task must not start executing before this instance. Note that the time instant that corresponds to the communicator instance may be chosen by a scheduler within an adequate range (less than 20ms here) without changing the logical semantics of the program, unless the involved communicator is connected to an external device for which real-time updates matter. The *fromJAviator* communicator is also used to communicate between tasks in different modules that run at different frequencies (e.g., the *ReadFromJAviator* task) which runs every 20ms and writes the second instance of the *fromJAviator* communicator, while the *WriteToGround* task runs every 100ms and reads the twentieth instance of the same communicator.

In order to check schedulability of the overall HTL program, it is sufficient to show that the *PJAviator* program is schedulable and properly refined by the other lower-level programs. The example presented in Figure 6 has been implemented in the Exotask system using the HTL grammar. However, so far we have only been able to test the implementation against a simulated *JAviator* plant. Real *JAviator* flights with this implementation are future work. Nevertheless, note that we have implemented full HTL support in the Exotask system: for every HTL program, there is an equivalent Exotask specification graph annotated using the HTL grammar which, when instantiated, is semantically equivalent to that original HTL program.

One important drawback of implementing real-time applications using Exotask graphs with the HTL grammar is the fact that currently there is no view to show the Exotask graph so that the hierarchical structure can be easily understood (e.g., as in Figure 6). Nevertheless, for complex real-time applications that require complex Exotask graphs, one can divide such a complex graph into smaller graphs. This is possible because both Exotask graphs and HTL programs are composable. Some elements of the complex Exotask graph will be replicated in multiple smaller graphs, but the number of elements that have to be replicated is acceptable (e.g., only portions of the hierarchical structure and some communicators used in multiple modules/modes will be replicated).

5.2 HTL Scheduler Implementation

HTL programs are compiled into E code [Henzinger and Kirsch 2007], which is virtual-machine code that specifies the timing and interaction of hard real-time tasks. E code is interpreted in real time by the Embedded Machine (or E machine). E code generated from HTL programs is time-portable to all platforms

on which the programs are schedulable and for which an E machine implementation exists. Time portability means that the code's functional and temporal behavior, in particular its I/O timing, does not change across platforms.

In the Exotask system, we have implemented a Java E machine and an HTL compiler that translates the HTL annotations on an Exotask graph specification into a form of E code that is designed to work with the corresponding instantiated Exotask graph. The compiler and the E machine together play the formal role of a pluggable Exotask scheduler. E code instructions that release tasks cause those tasks to be assigned exclusively to a scheduler thread responsible for running it once (in general, the binding of tasks to threads is temporary and dynamic, but the compiler has determined the maximum concurrency level and the scheduler then requests enough scheduler threads to ensure that there will always be a thread available to run each released task). E code instructions whose purpose is to copy values between ports, or between ports and communicators, use the Exotask system interface made available to schedulers for performing the deep copying between Exotask heaps. E code instructions that perform mode switches interrogate condition nodes in the graph, just like any other scheduler.

The latest HTL compiler supports two compilation strategies: flattening the hierarchy of an HTL program before compiling it, and directly compiling the original program into a hierarchical extension of E code called HE code [Ghosal et al. 2007], which has instructions for maintaining the original program hierarchy at runtime.

Flattening the hierarchy first results in E code, which is efficient if there is no parallelism in any refinements. However, the generated E code may be exponentially larger than the original program, and can introduce exponential runtime overhead, specifically as parallelism increases in refinements. Separate compilation in the flattening compiler is currently not supported and may be difficult to do. Directly compiling HTL programs into HE code results in code sizes that are linear in the sizes of the original programs [Ghosal et al. 2007]. Separate compilation of any subprogram in any order is possible and already implemented. The drawback of HE code is its increased runtime overhead on programs without parallelism in any refinements. However, HE code pays off on programs with parallelism in refinements, since its runtime overhead only grows linearly in the degree of that form of parallelism. In the Exotask version of the HTL compiler, we have only implemented the hierarchy-preserving compilation strategy targeting HE code.

In adapting the latest HTL compiler to work with the Exotask system, compilation is always done “on demand” at the point where the Exotask system invokes the HTL scheduler. That is, despite the opportunity for separate compilation when using the hierarchical strategy, there is no ability to save code artifacts across successive runs. In fact, the time taken to compile the complete hierarchical JAviator controller to E code is not a serious problem, as will be shown in Section 8.1, where the compilation time is included in the scheduling time for the example. However, a framework allowing schedulers to save and reuse intermediate results across runs (assuming the annotated Exotask specification graph has not changed) is a candidate for future work.

6. EXTENDING THE SYSTEM

We have already covered pluggable timing grammars and have described one of our two schedulers (the HTL scheduler) in detail. This section describes the system extension interface for plugging in schedulers, and also briefly covers the distributor interface, another pluggable extension point for constructing distributed Exotask programs. Construction and evaluation of actual distributed Exotask programs is for future work.

6.1 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 execute 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 validation 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 nontrivial 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 nondeterminism 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 nondisruptive 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 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.

6.2 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.

In a distributed graph, the graph is first partitioned into portions that run on different machines. We do not yet have an automated tool to do this, so, in developing and testing this facility we used manual partitioning. After partitioning, the graphs running on different machines will have communicators sharing the same name. These are considered to be replicas of each other. The distributer component then attaches itself to each such replicated communicator in each machine-specific subgraph within the distributed graph. The schedulers on the individual machines then run independently, relying on the distributer to replicate the communicators and distribute a common clock (the latter is optional: asynchronous execution is also permitted).

Distributers are written by using the weak-isolation model: that is, communicators to which distributers are attached are treated as weakly isolated by the validator, with an object provided by the distributer (called a channel) provided to them as their parameter. This allows much of the logic of the distributer to be written with off-the-shelf components, such as socket libraries, which need to execute outside the Exotask graph.

We have so far implemented only very simple distributers and presenting a serious empirical evaluation of this interface is thus future work. In an earlier publication of this article [Auerbach et al. 2007], the JAviator control program used a JAviator-specific distributer to overcome the absence of weak isolation in the general programming model. We later decided that this was not a good idea: distributers are system components and should not be application-specific. But, the facility exists as a serious design awaiting empirical evaluation.

7. JVM IMPLEMENTATION

Our implementation is packaged as a modular addition to the IBM WebSphere Real Time (WRT) product JVM [IBM Corp. 2006], which includes RTSJ [Bollella et al. 2000], the Metronome real-time garbage collector [Bacon et al. 2003], and an ahead-of-time (AOT) compiler. AOT compilation can be used to eliminate nondeterminism due to JIT compilation [Fulton and Stoodley 2007].

The internal JVM data structure for a thread includes a set of flag bits, two of which are used to indicate that a thread is exempt from being preempted by the global garbage collector. One of the bits causes the thread to behave as an NHRT (which is subject to runtime restrictions that are checked dynamically), while the other bit, originally added to support Eventrons, does not engender those restrictions. In the Exotask add-on, the Eventron bit was simply reused with a different set of supporting facilities. WRT also has a high-performance native method (one that does not use JNI) that invokes the Linux `nanosleep` function as required by both Eventrons and Exotasks for precise scheduling.

The Exotask add-on provides additional native methods that examine byte-codes of already loaded classes, perform deep cloning, and implement private heaps.

The Exotask add-on for IBM WebSphere Real Time, along with the Exotask development system described in this paper, is available for download [IBM Corporation 2007]. The HTL timing grammar and scheduler plugin is available separately [University of Salzburg 2007].

7.1 Exotask Garbage Collection

In WRT, heaps are constructed from multiple memory spaces, which are collections of physical areas along with lower-level logic for managing them. Exotask private heaps are just memory spaces that are detached from the main heap and that use the sliding compacting collector described in Bacon et al. [2004].

The collector's root scan uses 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 `execute` method and any newer stack frames are scanned, because only those frames can have pointers into the private heap.

7.2 Deep Cloning

In order to support RTSJ (scopes and immortal memory), the WRT VM has support for dynamically changing the active memory space of a thread and for switching memory spaces. This support was exploited in our implementation of the deep cloning required when objects are sent across ports. The implementation first temporarily switches the target memory space of the scheduler thread doing the deep clone to be the target heap. The standard (shallow) clone method of Java is then used (at a low level, bypassing the Java language checking for `Cloneable`) to copy objects; the internal allocation done by clone will automatically go to the target heap. To make the clone "deep" (i.e., recursive), 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. A work queue is maintained to avoid stressing stack space through excessive recursion. As nonnull reference fields are found, objects are put on the queue (or found on the queue if they were previously cloned). Objects on the queue that have not yet been cloned will then be cloned until the operation reaches a fixpoint.

8. MEASUREMENTS

This section presents performance measurements from three of the programming examples used in the paper. These are (1) the inverse pendulum presented in Section 2, (2) our production JAviator control using the TT schedule (the multimode version) presented in Section 4 (hereafter TTJAvControl), and (3) the still-evolving hierarchical JAviator control using the HTL grammar, presented in Section 5 (hereafter HTLJAvControl). We also measure a microbenchmark to shed light on the scheduling precision and freedom from garbage collection interference that is attainable at various periods, with and without Exotasks, on different machines. This should help to put other results in perspective. Finally, we present results from an Exotask reimplementation of the audio example that is used in the Eventrons [Spoonhower et al. 2006] and Reflexes [Spring et al. 2007] papers.

Measurements of TTJAvControl were done on the actual JAviator, which was airborne and hovering during the test (though confined in a cage for safety). The other two programming examples were run only as simulations. In the case of the inverse pendulum, we did not possess the necessary hardware and we also wanted to make the inputs deterministic and thereby test the repeatability of behavior across platforms, which was only possible using a simulation. In the case of the HTLJAvControl, controlling the airborne JAviator is future work, (for reasons discussed in Section 8.5).

Wherever possible, all tests, including simulations, were run on the same XScale PXA270 processor that is used in the JAviator, running on the same VM and the same kernel. In some cases, data were also collected using an AMD64 four-way 2.4GHz machine. There were two motivations for using the AMD64. First, this machine contrasts sharply in capability with the XScale and so sheds light both on time portability and on the behavior of Exotasks when given very different resources. Second, the RT Linux kernel on the AMD64 machine (based on RHEL5-RT, kernel version 2.6.21.4), stands out among general-purpose operating systems in providing good real-time behavior. In contrast, the XScale processor's kernel suffers from problems in scheduling precision (as the microbenchmarks will show). Of course, because the larger machine is so much more powerful, it is indeed likely that a schedule that is feasible on the XScale is trivially feasible on the AMD64. However, part of the issue in time portability is to ensure that time-critical events happen exactly on time (neither too late nor too early). This aspect is still meaningfully exercised by a comparison with a much more powerful processor, where there is a possibility of results being delivered too early due to faster processing. Admittedly, another class of problems could arise in porting between processors of similar power, especially when only modest slack exists in the schedule. Evaluation of this case is future work.

8.1 Exotask Fixed Costs

Figure 9 shows the times taken by one-time activities that occurred during initialization of the JAviator Exotask program. These are (1) the time taken to validate the specification graph and the program's Java code, (2) the time

	Inverse Pendulum		JAvControl		HTLJAvControl	
Tasks	4		13		21	
Connections	3		28		45	
	XScale	AMD64	XScale	AMD64	XScale	AMD64
Validation	2433.11	32.1	15,672.6	259.76	22,132.5	329.83
Instantiation	103.12	0.86	503.1	18.83	189.4	6.79
Scheduling	211.51	5.2	321.3	13.10	813.2	31.164

Fig. 9. One-time costs in the JAviator program (ms).

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. The times are from single runs (not averages) but the variance from run to run is extremely low.

As shown in the figure, these times are small relative to the typical duration of program execution. On the AMD64 processor, which loads classes from a conventional hard disk, all times are well under a second, even for the 21-task HTLJAvControl. The XScale uses compact flash in lieu of disk and data must be decompressed as well, which causes validation times to rise considerably (up to 22 seconds for the HTLJAvControl). Note that all classes used by an application are loaded as part of validation, so this cost is counted. A positive corollary is that these slowdowns do not affect the execution of the program.

Otherwise, validation time depends on code complexity. Instantiation time includes the time taken to construct private heaps, but also includes Exotask constructor execution and copying of parameter information into the Exotask heaps. Thus, this overhead is the least obviously related to the number of tasks. Scheduling time varies according to the scheduler chosen and the complexity of the timing annotations that must be processed.

8.2 Achievable Period Frequencies in the Presence of Garbage Collection

This section presents results from a microbenchmark aimed at determining the scheduling precision available for Exotasks on different platforms in the presence of garbage collection. It demonstrates that Exotasks achieve low latency despite the challenges of running in a full-function Java VM. It provides some sense of the relative jitter at different frequencies. Clearly, these benchmarks are very sensitive to the timing and scheduling precision of the underlying kernel. We present them partly to make clear the degree to which Exotasks may or may not overcome inherent problems in the underlying platform, and also to set some expectation bounds for the other more “real-world” measurements.

In the benchmark, a single, empty Exotask was scheduled using various periods on both of the hardware platforms. The runs were done both in an Exotask-enabled version of the target VM, and in the identical VM without using Exotasks (the code was the same but the Exotask scheduler thread was running as an ordinary thread). The runs were done both with and without a concurrent ordinary thread (the interference thread) running on the main

	XScale			AMD64		
Quantum Size	3ms			500 μ s		
Heap Size	12MB			64MB		
Alloc.Rate	256k/sec			2MB/sec		
	Min	Max	Avg	Min	Max	Avg
GC Quantum	1.320ms	17.535ms	5.021ms	45.0 μ s	751.3 μ s	508.8 μ s
GC Cycle	2746.3ms	2901.4ms	2836.0ms	183.2ms	192.9ms	190.9ms
GC Interval	7.906sec	7.916sec	7.912sec	9.510sec	9.524sec	9.518sec

Fig. 10. Global garbage collection parameters and times.

heap. This thread was present to simulate the effect of significant other activity within the VM, including activity that would cause global-heap garbage collections.

The global-heap garbage collector (GC) in both systems was Metronome, a real-time GC using time-based scheduling [Bacon et al. 2003; Auerbach et al. 2007]. Metronome divides its GC cycles into short quanta, so that GC activities are interleaved at a fine time scale with application activities. Metronome starts a GC when the amount of used memory reaches half the heap size. Cycles may take quite a bit longer than would be the case if the application were paused throughout, but real-time goals of the application are preserved.

Architectural realities required use of a different VM version on the XScale versus the AMD64, and some GC-related parameters are set differently to accommodate the coarse-grained timing of the XScale. We also chose heap sizes and allocation rates that were aimed at rough parity in the frequency of GCs. The quantum size, heap size, and allocation rate values for the two systems are shown in Figure 10. In both cases, the interference thread achieved its allocation rate using 48-byte objects and keeping the most recent 40,000 of those objects live. The actual GC quantum durations, the GC cycle durations, and the intervals between GC cycles are also shown in Figure 10.

Figure 11 shows the results of this microbenchmark. The first thing to note is that periods below 500 μ s are infeasible on the XScale. We show the one best result we have at that rate (for Exotasks without GC interference), and the minimum time is well above the period time, indicating that the processor is essentially saturated. Note that not only is this processor limited to 600MHz but the VM that we run there has no JIT compiler. The next thing we note is that, across all periods, and regardless of platform, the jitter (as measured either by the standard deviation or by the worst-case values) is far better when using Exotasks compared to ordinary Java threads, even with a real-time GC to help out.

The microbenchmark also shows that the presence of global-heap GC activity makes the non-Exotask measurements degrade sharply while the Exotask measurements are much less affected. The Exotask measurements do show *some* GC interference, which is fairly negligible on the AMD64 but more noticeable on the XScale. The reason that there is any interference at all stems from two factors. First, the extra thread is executing quite a bit of the time, affecting the OS scheduler's behavior. Second, while our design guarantees that Exotasks

Period	No Exotasks			Exotasks		
	XScale, No Concurrent Global Heap GCs					
	Min	Max	SDev	Min	Max	SDev
5,000 μ s	3,000	7,000	730.24	4,430	5,160	20.20
1,000 μ s	0	3,000	748.12	418	2,075	42.67
500 μ s	0	3,000	535.29	373	2,071	82.02
100 μ s	–	–	–	369	1,575	42.96
	XScale, With Concurrent Global Heap GCs					
	Min	Max	SDev	Min	Max	SDev
5,000 μ s	0	115,000	5495.79	2,495	7,570	86.44
1,000 μ s	0	553,000	5212.81	404	1,957	33.08
500 μ s	0	223,000	3874.72	365	2,069	81.43
	AMD, No Concurrent Global Heap GCs					
	Min	Max	SDev	Min	Max	SDev
5,000 μ s	4,951	5,050	2.95	4,951	5,006	1.63
1,000 μ s	808	1,198	3.09	953	1,005	0.67
500 μ s	231	770	1.89	445	506	0.67
100 μ s	7	571	1.51	38	128	0.52
50 μ s	7	202	5.42	14	95	0.47
25 μ s	7	397	15.44	10	98	1.29
	AMD, With Concurrent Global Heap GCs					
	Min	Max	SDev	Min	Max	SDev
5,000 μ s	3,469	6,530	81.72	4,968	5,031	2.31
1,000 μ s	7	3,238	48.06	969	1,017	1.11
500 μ s	7	2,399	85.53	460	510	0.74
100 μ s	7	2,253	22.76	64	116	0.50
50 μ s	7	2,313	27.58	14	83	2.17
25 μ s	7	2,363	18.46	10	85	0.62

Fig. 11. Scheduling precision with and without exotasks (microseconds).

are not paused by the global-heap GC, there is still some residual interference from locking of low-level data structures. Both effects are worse on the XScale, which is a uniprocessor with a less advanced kernel, and the VM technology there represents fewer man hours of intensive testing and development (an experimental VM versus an IBM product).

Finally, the results on the AMD64 show that very low jitter is achievable down to the 100's of microseconds range, and even the tens of microseconds, if an occasionally rare outlier can be tolerated.

8.3 Time Portability

To measure time portability across platforms, we used the inverse-pendulum example. The example was started with the pendulum vertical, and the cart was subjected to (simulated) random perturbing forces of $\pm 1,200$ Newtons at intervals ranging from 25 μ s to 25ms. Each random force decayed after first being applied, such that its value declined by half every 25 μ s. The controller was deemed successful if, within a minute, the pendulum never destabilized by going below the horizontal plane. All random distributions were uniform over the stated ranges.

Processor	Same Input	Different
XScale	.9929	.0301
AMD64	.9999	.0292
Both	.9396	.0345

(a) Correlation

Processor	Read Angle				Write Motor			
	Min	Mean	Max	SDev	Min	Mean	Max	SDev
XScale	9,411	9,999.88	10,331	29.84	2959	9,999.09	16,798	354.68
AMD64	9,942	9,999.99	10,016	1.09	10,000	10,000	10,003	0.38

(b) Interrival Variation (μ s)Fig. 12. Time portability statistics: Inverted pendulum (times in μ s).

Since this example employs strong isolation, this example should be completely time-portable, which in effect means that the behavior should be environment-determined [Henzinger and Kirsch 2007]. That is, if identical sensor measurements are presented at identical times (relative to a starting point), the control program should produce exactly the same cart motor force values at exactly the same relative times.

Because the hardware was simulated, we have the capacity to produce identical sensor inputs but we needed a methodology to produce a time-sensitive replay. If we simply replayed the same values independent of the relative time at which control program read them, the test would be meaningless. But, to produce accurate values based on the time of reading, using “live” simulation, the simulator would be obligated to simulate continuously or to do so instantaneously on demand. When the simulator and controller are run on the same processor, live simulation would be highly perturbing. If they are run on different processors, communication latency would affect the result substantially. Therefore, we used a table-driven approach. The simulation and controller were first run in virtual time with the simulation determining and recording all the sensor values at 25μ s intervals and the controller running at its designed period of 10ms. For the experiments, tables from various “training” runs are read into memory. Then, the controller was run along with a version of the simulator that used table lookup rather than live simulation to replay values in a time-sensitive fashion (25μ s granularity). The overhead of the table lookups was negligible and we were able therefore to evaluate repeatability directly. We captured the sequence of motor values (outputs) issued by each run. We were then in a position to compute the correlations between the outputs of different runs, either with the same inputs or different inputs, on the same platform, or on different platforms.

Figure 12 shows the results of the time portability experiment run on both the XScale and the AMD64 with the simulator’s table derived from the same training runs (same inputs) and from different training runs (different inputs). Correlations (Pearson’s ρ) are extremely high when the same inputs are replayed and negligible when different inputs are used. Replays of the same inputs on the same platform, while not the goal of the experiment, are shown

to provide a baseline of repeatability that one might expect from this experimental technique. When results on the same input are compared across the two different platforms, we get a ρ value that is only slightly lower (0.9396), indicating that the environment-determinedness goal is effectively met, limited by the available timing precision. As the algorithm itself is deterministic, the drop in correlation compared to same-platform cases is due to the greater timing variability on the XScale. Indeed, the remaining columns in Figure 12 show that this is the case. The interval between successive executions of the ReadAngle Exotask and that between successive executions of the WriteMotor Exotask should each be exactly 10ms. The variations (especially in the WriteMotor task, which is subject to more perturbation if the longer-executing compute task that precedes it violates its time limit) are negligible on the AMD64 but noticeable on the XScale. The fact that the XScale runs self-correlated at a higher level suggests that there is determinism in these perturbations (they happen at approximately the same point in each run). We are, therefore, confident that improvements in the technological underpinnings on the XScale will improve the repeatability achieved in the future.

8.4 Flying the JAviator

This section presents measurements of the JAviator in flight using the TTJAvControl program (as presented in Section 4). The JAviator was airborne for about 5 minutes, during which time it was hovering most of the time with occasional adjustments of attitude and altitude to keep it oriented. For safety, the JAviator was confined in a cage. A single Ethernet cable connected the JAviator to a separate computer running the joystick and console display. Communication between the two was via UDP, and a TCP connection was also present for collecting data. As previously explained, the XScale processor on which the Java VM was running, and the separate microcontroller for generating PWM signals, were both on board the JAviator.

Figure 13 shows the distribution of times between executions of the fromJAviator Exotask (see Section 4) when the controller was run alone in the JVM as is typically the case during flights. Figure 14 shows distribution of times for the same Exotask when a separate thread is allocating memory concurrently, using the same algorithm as in Section 8.2. As can be seen by comparing both the histograms and the accompanying statistics, the TTJAvControl behaves quite well in both cases. The histograms make explicit that the worst-case outliers tend to be rare, as reflected in the low standard deviation. This is consistent with microbenchmark results at longer periods (at the 20ms period the XScale handles the load comfortably).

The fact that the pattern of outliers is different between the two histograms (a larger number of smaller outliers when there are concurrent global heap garbage collections) is consistent with earlier results we have reported in similar [Auerbach et al. 2007] and not-so-similar [Spoonhower et al. 2006] systems. We cannot claim to fully understand why this pattern tends to occur, but at least it is consistent.

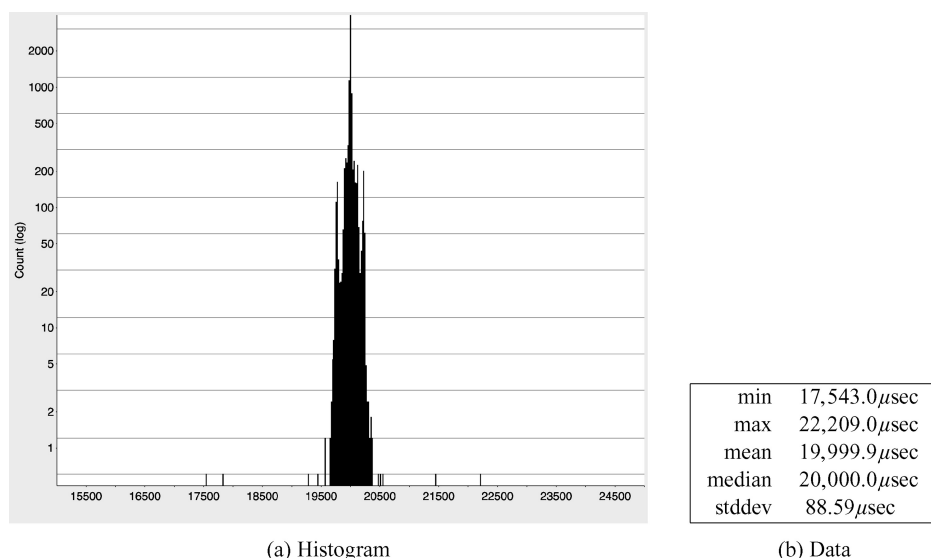


Fig. 13. TTJAvControl: Interarrival times of the fromJAviator task, when no concurrent allocation is done.

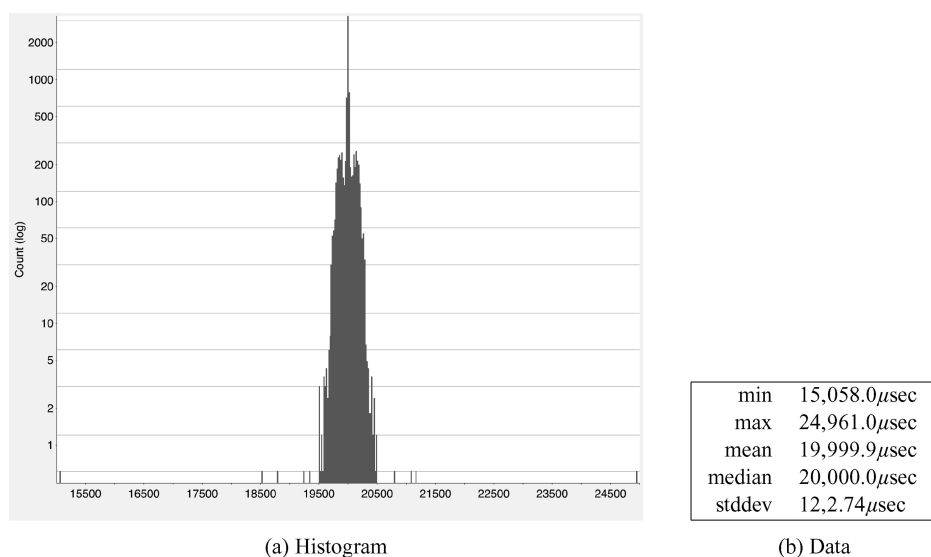


Fig. 14. Interarrival times of the fromJAviator task, when concurrently allocating 256KB per second.

8.5 The Hierarchical (HTL) Version of the JAviator Control

Section 5 describes the HTLJAvControl, a hierarchical controller for the JAviator which is still being tuned for actual use in flight. This controller cannot be directly compared to the TTJAvControl results from the previous section because those were done on the XScale processor with the JAviator in flight. We could not

Controller	min	max	mean	median	Std.Dev
HTLJAvControl	19,955.1	20,254.0	20,070.4	20,066.0	22.7
TTJAvControl	19,967.0	20,026.0	20,000.0	20,000.0	0.9594

Fig. 15. Interarrival times of the ReadFromJAviator task on the AMD64 machine, when no concurrent allocation is done (times in μ s).

Controller	min	max	mean	median	Std.Dev
HTLJAvControl	19,909.5	20,334.0	20,069.2	20,064.0	22.8
TTJAvControl	19,972.0	20,029.0	19,999.99	20,000.0	1.2244

Fig. 16. Interarrival times of the ReadFromJAviator task on the AMD64 machine, when concurrently allocating 2MB per second (times in μ s).

even run the HTLJAvControl on the XScale processor because the interpretive E machine currently used by the HTL scheduler had too high an overhead and left too little time left for running the Exotasks. This problem is being addressed by further compiling the E code to a more efficient representation, as is done in the original HTL language. Meanwhile, both controllers run well on the AMD64 processor, controlling a simulated JAviator, and that is what we report on in this section.

Figure 15 depicts the interarrival times of the ReadFromJAviator task in the HTLJAvControl. Figure 16 depicts the interarrival times of the same task in the same controller when the interference thread described in Section 8.2 is used (with the AMD64 parameters shown in Figure 11). For comparison, times for the TTJAvControl running under the identical circumstances are also shown. As can be seen, both controls perform quite adequately on this platform, and both controls are reasonably unaffected by the concurrent global heap garbage collection activity, but the HTL version does have noticeably more jitter in absolute terms. Of course, the jitter of the HTLJAvControl on the AMD64 is actually less than that of the TTJAvControl on the XScale, but we know that more work will be needed before the HTLJAvControl can run adequately on the XScale. We do not believe this represents an inherent limitation of this approach, but simply represents a need to further improve the efficiency of our E machine implementation.

8.6 The Audio Example Revisited

Both Spoonhower et al. [2006] and Spring et al. [2007] employed an example of an audio tone generator running at sample-size buffering directly against a hardware audio card. For CD-quality sound (22.05KHz), this requires a period of only 45μ s. The audio example also uses a less time-critical “low-frequency” task to do audio synthesis. The synthesis task communicates with the audio frequency generator using whatever mechanism is called for by the programming model. Both Eventrons and Reflexes reported good results for this example, with over 99% of periods starting “on time,” defined as within 5μ s of the target.

	Min	Max	SDev	$\pm 5\mu s$	$\pm 10\mu s$
Eventron, no GC	35	56	1.03	99.196%	99.999%
Exotask, no GC	12	146	1.00	98.848%	99.998%
Eventron, with GC	32	59	1.05	98.991%	99.997%
Exotask, with GC	12	168	0.99	98.883%	99.999%

Fig. 17. Eventrons and exotasks running the audio example (times in μs).

The audio example is readily reimplemented as an Exotask, with the communication mechanism being the exchange of scalar values via weak isolation. A “channel” abstraction was used in the Eventrons research to hide the complexity of unsynchronized scalar value sharing. The same abstraction is readily implemented on top of the Exotask weak-isolation model.

For purposes of comparison, we were able to run the Eventron and Exotask versions of the program on the same hardware and kernel, using the original Eventron software and the current Exotask software. We were not in a position to run the Reflex version (it requires a different VM). For technical reasons, we could not deploy the Eventron software on the XScale processor and so measurements were taken only on the AMD64 (this is similar to the class of hardware used in the original Eventron and Reflex papers).

Figure 17 shows the results of this study. The Eventron implementation has less distant outliers and a marginally higher percentage of intervals falling within $5\mu s$ of the target. However, the percentage of such intervals for the Exotask implementation is nearly as high and so Exotasks are clearly in the same league running the identical application. It should be clear both from these results and from those presented in the Reflex paper that there is little or no necessary loss of performance or precision when going from a less expressive programming model to a more expressive one, provided the code is using only capabilities that were present in the simpler model. In fact, our reimplementation of the audio example makes no use of features that were not present in Eventrons beyond the ability to allocate exception objects when failing (the program does not allocate anything in its main loop). What a richer programming model provides, however, is the ability to exploit additional capabilities at modest cost. Both JAviator examples and the inverse-pendulum example made use of Exotask features not present in any earlier model, such as the ability to allocate objects and have them collected on a scheduled basis and the ability to communicate objects of arbitrary type to other tasks. As can be seen, those results, although not employing the aggressively short period of this one, exhibit low jitter and high repeatability.

9. 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. 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 [Burns and Wellings 1997] and Erlang [Armstrong et al. 1996]. Synchronous reactive programming [Halbwachs 1993] 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) [Henzinger et al. 2003]. 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 [Bacon et al. 2003], incorporated in the IBM WebSphere Real Time VM [IBM Corp. 2006], is one of several recent examples (others are [Siebert 2004] and [Henderson 2002]). 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.

One approach which shares with Exotasks the idea of partitioning the heap is the hierarchical real-time garbage collection [Pizlo et al. 2007]. This approach splits the heap into disjoint partitions, called heaplets, which are collected independently by different garbage collectors. Each garbage collector can be tuned to match the requirements of the set of tasks running in the respective heaplet. The approach differs from Exotasks in that hierarchical RTGC allows cross heap pointers but assesses a performance penalty, so some determinism is sacrificed in order to achieve an even less restrictive programming model (no restrictions at all). Exotasks disallow such pointers (for the most part) in order to stress greater determinism. Also, garbage collection in hierarchical RTGC is not scheduled, and so its impact on performance is less predictable.

Another approach is to avoid collisions with the garbage collector by avoiding heap allocations entirely, which is the approach taken by NHRTs [Bollella et al. 2000], Eventrons [Spoonhower et al. 2006], Reflexes [Spring et al. 2007], and StreamFlex [Spring et al. 2007]. 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, which can throw unexpected exceptions as a result).

StreamFlex shares with Exotasks the property that computing is done by a graph of communicating tasks, not just a single task. The emphasis is on “streaming” (data-driven rather than time-driven processing) but the system achieves very low latency by exploiting a real-time environment (the StreamFlex system is built on Reflexes). The channels in a StreamFlex graph carry capsules, which is a much more restricted data type than those allowed by Exotasks. However, capsules, because of this restriction, can be copied by reference and do not require deep cloning.

One rather heavyweight way of implementing private heaps is the isolates construct [Java Community Process] 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.

In addition to the WebSphere Real Time VM used as our implementation base, there are a number of others now available [Purdue; Bollella et al. 2005; AICAS]. For example, OVM [Purdue] was used to implement the Reflex and StreamFlex systems.

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 [Simulink 2007] and Ptolemy [Lee 2003]. 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 [Real-Time-Workshop 2007] 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 validator 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 remains as future work.

The Exotask programming model is designed to optimize code efficiency, portability, and determinism. 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 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) [Stewart et al. 1997] 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.

Finally, we note that the JAviator example is a real application and that papers concerning real applications are important contributions to the field. Other reports on applications written in Java using real-time Java VMs or similar technology include ones in avionics [Armbuster et al. 2006], shipboard computing [IBM 2007], audio processing [Auerbach et al. 2007; Juillerat et al. 2007], and industrial control [Gestegard Robertz et al. 2007].

10. CONCLUSION

We have introduced Exotasks, a novel Java programming construct that achieves low latency with the fewest to-date restrictions on the use of Java features. The Exotasks system supports pluggable schedulers, and the schedulers we have written, in conjunction with the isolation properties of Exotasks, achieve 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 recently extended this virtual machine and the Exotask model so that all the capabilities of Eventrons are also provided. We have used Exotasks to fly a quadrotor model helicopter, the JAviator. Our experiments show that Exotasks are adequately efficient and achieve freedom of interference from other Java code and the garbage collector. Comparisons of runs on different hardware show that time portability has been achieved, at least for the 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.

REFERENCES

- AICAS. The Jamaica virtual machine. <http://www.aicas.com>.
- ARMBUSTER, A., BAKER, J., CUNEI, A., HOLMES, D., FLACK, C., PIZLO, F., PLA, E., PROCHAZKA, M., AND VITEK, J. 2006. A Real-time Java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst. (TECS)*.

- ARMSTRONG, J., VIRDING, R., WIKSTRM, C., AND WILLIAMS, M. 1996. *Concurrent Programming in Erlang*, 2nd Ed. Prentice-Hall.
- AUERBACH, J., BACON, D. F., BLAINEY, B., CHENG, P., DAWSON, M., FULTON, M., GROVE, D., HART, D., AND STOODLEY, M. 2007. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of The 7th ACM/IEEE International Conference on Embedded Software*. ACM, New York, 249–258.
- AUERBACH, J., BACON, D. F., BO MERS, F., AND CHENG, P. 2007. Real-time music synthesis in Java using the Metronome garbage collector. In *Proceedings of the International Computer Music Conference*.
- AUERBACH, J., BACON, D. F., IERCAN, D. T., KIRSCH, C. M., RAJAN, V. T., ROECK, H., AND TRUMMER, R. 2007. Java takes flight: time-portable real-time programming with exotasks. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools (LCTES'07)*. ACM, New York, 51–62.
- BACON, D. F., CHENG, P., AND GROVE, D. 2004. Garbage collection for embedded systems. In *Proceedings of the 4th ACM International Conference on Embedded Software*. ACM, New York, 125–136.
- BACON, D. F., CHENG, P., AND RAJAN, V. T. 2003. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 285–298.
- BACON, D. F. AND SWEENEY, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, 324–341.
- BOLLELLA, G., DELSART, B., GUIDER, R., LIZZI, C., AND PARAIN, F. 2005. Mackinac: Making hotspot realtime. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE, Los Alamitos, CA, 45–54.
- BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. 2000. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, Boston, MA.
- BURNS, A. AND WELLINGS, A. 1997. *Concurrency in ADA* 2nd Ed. Cambridge University Press, Cambridge, UK.
- ECLIPSE FOUNDATION. 2007. The Eclipse open development platform. <http://www.eclipse.org>.
- FULTON, M. AND STOODLEY, M. 2007. Compilation techniques for real-time Java programs. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, New York, 221–231.
- GESTEGARD ROBERTZ, S., HENRIKSSON, R., NILSSON, K., BLOMDELL, A., AND TARASOV, I. 2007. Using real-time Java for industrial robot control. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. ACM, New York, 104–110.
- GHOSAL, A., HENZINGER, T., IERCAN, D., KIRSCH, C., AND SANGIOVANNI-VINCENTELLI, A. 2006. A hierarchical coordination language for interacting real-time tasks. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software*. ACM, New York, 132–141.
- GHOSAL, A., IERCAN, D., KIRSCH, C., HENZINGER, T., AND SANGIOVANNI-VINCENTELLI, A. 2007. Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code. In *Proceedings of the APGES Workshop*.
- HALBWACHS, N. 1993. *Synchronous Programming of Reactive Systems*. Kluwer, Norwell, MA.
- HENDERSON, F. 2002. Accurate garbage collection in an uncooperative environment. *SIGPLAN Notices* 38, 2, 256–263.
- HENZINGER, T. AND KIRSCH, C. 2007. The embedded machine: Predictable, portable real-time code. *ACM Trans. Prog. Lang. Syst.* 29, 6.
- HENZINGER, T., KIRSCH, C., AND HOROWITZ, B. 2003. Giotto: A time-triggered language for embedded programming. *Proc. IEEE* 91, 1, 84–99.
- IBM. 2007. DDG1000 next generation Navy destroyers. <http://www.ibm.com/press/us/en/pressrelease/21033.wss>.
- IBM CORP. 2006. *WebSphere Real-Time User's Guide*, 1st Ed.
- IBM CORPORATION. 2007. IBM expedited real time task graphs. www.alphaworks.ibm.com/tech/xrtgs.
- JAVA COMMUNITY PROCESS. JSR-121 application isolation API. <http://www.jcp.org/aboutJava/communityprocess/final/jsr121>.

- JUILLERAT, N., MÜLLER ARISONA, S., AND SCHUBIGER-BANZ, S. 2007. Real-time, low latency audio processing in java. In *Proceedings of the International Computer Music Conference*.
- LEE, E. 2003. Overview of the Ptolemy project. Tech. rep. UCB/ERL M03/25, EECS Department, University of California, Berkeley.
- OGATA, K. 1997. *Modern Control Engineering*. Prentice Hall, Upper Saddle River, NJ.
- PILLAI, P. AND SHIN, K. G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. ACM, New York, 89–102.
- PIZLO, F., HOSKING, A. L., AND VITEK, J. 2007. Hierarchical real-time garbage collection. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools (LCTES'07)*. ACM, New York, 123–133.
- PURDUE. The OVM virtual machine. <http://www.ovmj.org>.
- REAL-TIME-WORKSHOP. 2007. <http://www.mathworks.com/products/rtw>.
- SIEBERT, F. 2004. The impact of realtime garbage collection on realtime Java programming. In *Proceedings of the 7th Annual IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*. IEEE, Los Alamitos, CA, 33–40.
- SIMULINK. 2007. <http://www.mathworks.com/products/simulink>.
- SPOONHOWER, D., AUERBACH, J., BACON, D. F., CHENG, P., AND GROVE, D. 2006. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 283–294.
- SPRING, J. H., PIZLO, F., GUERRAUI, R., AND VITEK, J. 2007. Programming abstractions for highly responsive systems. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*. ACM, New York, 191–201.
- SPRING, J. H., PRIVAT, J., GUERRAUI, R., AND VITEK, J. 2007. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, New York, 211–228.
- STEWART, D. B., VOLPE, R. A., AND KHOSLA, P. K. 1997. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Engin.* 23, 12, 759–776.
- UNIVERSITY OF SALZBURG. 2007. Exotask htl scheduler. htl.cs.uni-salzburg.at/exotask-htl.

Received October 2007; revised March 2008; accepted July 2008