Flexible Task Graphs: A Unified Restricted Thread Programming Model for Java

Joshua Auerbach
David F. Bacon
IBM Research

Rachid Guerraoui
Jesper Honig Spring
Ecole Polytechnique Fédérale de Lausanne

Jan Vitek
Computer Science Dept.
Purdue University

Abstract

The disadvantages of unconstrained shared-memory multi-threading in Java, especially with regard to latency and determinism in real-time systems, have given rise to a variety of language extensions that place restrictions on how threads allocate, share, and communicate memory, leading to order-of-magnitude reductions in latency and jitter. However, each model makes different trade-offs with respect to expressiveness, efficiency, enforcement, and latency, and no one model is best for all applications.

In this paper we present Flexible Task Graphs (Flexotasks), a single system that allows different isolation policies and mechanisms to be combined in an orthogonal manner, subsuming four previously proposed models as well as making it possible to use new combinations best suited to the needs of particular applications. We evaluate our implementation on top of the IBM WebSphere Real Time Java virtual machine using both a microbenchmark and a 30 KLOC avionics collision detector. We show that Flexotasks are capable of executing periodic threads at 10 KHz with a standard deviation of 1.2μs and that it achieves significantly better performance than RTSJ's scoped memory constructs while remaining impervious to interference from global garbage collection.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters, run-time environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems.

General Terms Languages, Experimentation.

Keywords Real-time systems, Java virtual machine, Memory management, Ownership types.

1. Introduction

The Java programming language has become a viable platform for real-time systems with applications in avionics [2], shipboard computing [19], audio processing [4, 21], industrial control [17] and the financial sector [8]. High performance real-time Java virtual machines (RT JVMs) are now available from multiple vendors [28, 13, 3, 1]. Ideally, real-time Java applications would not require language restrictions or special language features, they would be written just as any other application using the same design patterns, programming idioms and familiar library classes. Achieving this ideal is becoming feasible due to progress in real-time garbage collection [6, 29, 18], ahead-of-time compilation [16], operating system support [3, 23] and faster processors. Many real-time programs can now be written as simple Java programs where developers pay attention to the timing properties of the application logic without having to be overly concerned about interference from the virtual machine.

However, some applications have latency/throughput real-time requirements that cannot be met by current real-time garbage collection (GC) technology. When scheduling latency goes below a millisecond any interference from the JVM is likely to result in missed deadlines. One of the key design decisions of the Real-time Specification for Java (RTSJ) [14] was to support those applications with a programming model that restricts expressiveness to avoid unwanted interactions with the JVM and the GC in particular. The RTSJ introduced the `NoHeapRealtimeThread` for this purpose. More recently, alternatives to `NoHeapRealtimeThread` have been proposed, such as Eventrons [31], Reflexes [33], Exotasks [5], and StreamFlex [32]. Typically, time critical tasks account for only a fraction of the code of the entire application. The rest of the application is either soft real-time or non-real-time code. The existence of restrictions in a subset of the code, along with rules for communication between that code and the remainder of the application, gives rise to a *restricted thread programming model*, or RTPM. As with any programming model, an RTPM has advantages beyond low scheduling latency, including static error detection and facilitation of development tools and model-driven development strategies.

At first glance, one may wonder what added value these different restricted thread programming models bring over and above RTSJ's `NoHeapRealtimeThreads` which are, after all, supported by all RT JVMs. Experience implementing [9, 15, 25, 2] and using [12, 24, 10, 26, 27] the RTSJ revealed a number of serious deficiencies. In the RTSJ, interference from the GC is avoided by allocating data needed by time critical real-time tasks from a part of the JVM's memory that is not subject to garbage collection, dynamically checked regions known as *scoped memory areas*. Individual objects allocated in a scoped memory area cannot be deallocated; instead, an entire area is torn down as soon as all threads exit it. Dynamically enforced safety rules check that a memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope with a shorter lifetime and that a `NoHeapRealtimeThread` does not attempt to dereference a

LCTES'08, June 12–13, 2008, Tucson, Arizona, USA.

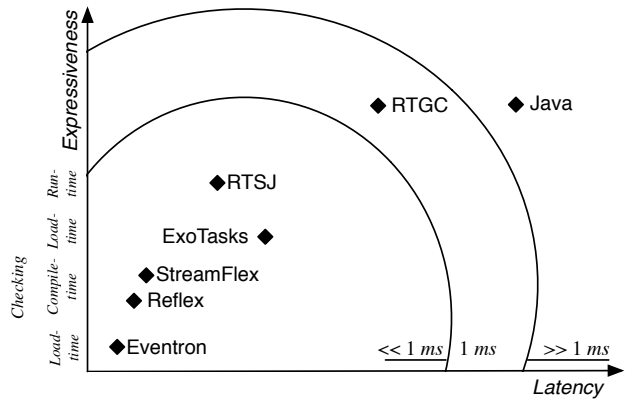


Figure 1. Comparing expressiveness versus worst case latencies of different Real-time Java Programming Models. There is a tradeoff between latency guarantees and expressiveness. We focus on programming models that target sub-millisecond scheduling latencies. The RTSJ is arguably the most expressive programming model with sub-millisecond latency, but it incurs throughput overheads due to run-time scope checks and faces the possibility of run-time failures. This paper presents a unification of four programming models (ExoTask, StreamFlex, Reflex, Eventron) which rely on static checking and thus have no run-time checks.

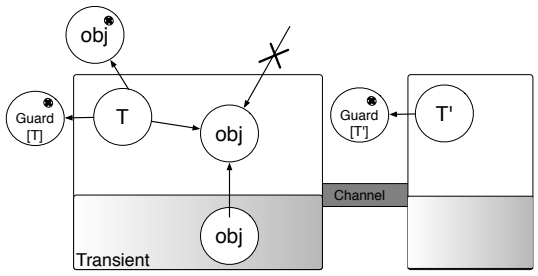


Figure 2. The Flexotasks runtime relationships. A task of type T has a memory region split into a garage collected *private heap* and an optional *transient area* which is reclaimed in bulk between invocations of `execute()`. Task-allocated objects are isolated from the main Java heap and thus not affected by the main heap garbage collector. A task may optionally access objects that are outside its memory region if those objects are *reference immutable*. An `AtomicFlexotask` may be accessed from plain Java code through a *guard*, which is a heap-allocated proxy object. Tasks communicate by the means of *connections*.

pointer into the garbage collected heap. The worst-case cost of using scoped memory is predictable [2]: every store to a reference field incurs a substantial slow-down due to a slow path that performs a range check [25]. Memory reads require two branches (a check whether the thread is a `NoHeapRealtimeThread` and a check to ensure the reference does not point into the garbage collected heap) [27]. Unfortunately, there is a significant difference between slow path and fast path memory accesses which makes predicting worst-case performance difficult. However, use of the fast path is essential to maintaining acceptable average case throughput. Another issue with the RTSJ programming model is that, due to a lack of isolation, it is possible for a `NoHeapRealtimeThread` to block on a lock held by a plain Java task. If this ever occurs, all bets are off in term of real-time guarantees as the blocking time cannot be bounded. Finally, dynamic memory access checks entail a loss of compositionality. Components may work just fine when tested independently, but break when put in a particular scoped memory context. This is because for a RTSJ program to be correct, developers must deal with an added dimension: *where* a particular datum was allocated. Design patterns and idioms for programming effectively with scoped memory have been proposed [26, 11, 12], but anecdotal evidence suggests that programmers have a hard time dealing with `NoHeapRealtimeThread`s and that resulting programs are brittle.

If the programming community were to choose to avoid the problems of `NoHeapRealtimeThread`s by adopting and eventually standardizing the newer RTPMs, the problem of choosing among them is difficult because they make different tradeoffs and emphasize different advantages. The purpose of this work is to provide a unifying framework, which we call Flexible Task Graphs (or Flexotasks), for four of the available RTPMs (Eventrons, Reflexes, Exotasks, and StreamFlex). In doing this, we respect the reality that there are some hard tradeoffs and that not all features of all four models can be available simultaneously. Consequently, a Flexotask program is constructed using a core of unified features and a choice from among alternative optional features. The core provides essential properties needed by any RTPM and confers the maximum practical set of advantages common to the precursor RTPMs. Options add features or guarantees that can be activated according to the requirements of a particular application. We do not aim to subsume `NoHeapRealtimeThread` functionality in Flexotasks, because we want to avoid the need for run-time checks. Flexotasks employ both development-time checking and enforcement at program initialization time. Consequently, dynamic checks are limited to a small number of less frequently executed operations (e.g. JNI callbacks) that are impossible to check statically.

We introduce Flexotasks by summarizing the main features, explaining how and why each was chosen from among the useful properties of Eventrons, Reflexes, Exotasks, and StreamFlex and how some inherent conflicts between those models were resolved. In subsequent sections, we elaborate on aspects of the resulting system, in particular, how Flexotasks are validated, and how the system may be used in practice for programming.

2. Flexible Task Graphs: Features, Origins, and Rationale

Our goal with Flexotasks was to subsume all four precursor RTPMs: Eventrons, Reflexes, Exotasks, and StreamFlex. We used two main strategies to accomplish this. First, where a feature of one model was general enough already to subsume another, we chose the more general feature, and, similarly, we preferred less restrictive rules over more restrictive ones. In choosing the more general or less restrictive capability, we were aware that the less general or more restrictive one may have had advantages of simplicity or efficiency. To recover simplicity for users who desire it, we rely on selective veneer interfaces that provide a simplified semantics (we have such veneers for both Eventrons and Reflexes). To recover efficiency for applications that can live within tighter restrictions, we made some stronger checks available but optional (both allocation and synchronization are allowed by default but may be forbidden by static checking).

Second, where two precursor models simply did something differently, we incorporated both mechanisms, and either required a choice (if they conflicted) or allowed both to be used together (if they did not). Thus, we retained the storage management semantics of both Exotasks and Reflexes, the external communication mechanisms of both Eventrons and Reflexes, and the intertask communication mechanisms of both Exotasks and StreamFlex. We will

Feature	Eventrons	Reflexes	Exotasks	StreamFlex	RTSJ	Flexotasks
Restricted Unit	Task	Task	Graph	Graph	Thread	*Graph*
Long-term Storage	Pre-allocated	Stable region	Private heap	Stable region	Immortal memory	*Private heap*
Short-term Storage	Stack variables	Transient region	Private heap	Transient region	Scoped memory	*Transient region*
External Communication	Shared scalars	Transactions	None	Transactions	Programmed	*Shared and transactions*
Synchronization	Forbidden	Discouraged	Disabled	Discouraged	Allowed	*Allow, forbid, or disable*
Scheduling	Periodic	Periodic	Pluggable	Data Driven	Periodic and event	*Pluggable*
Construction	Direct	Direct	Via template	Direct	Direct	*Via template*
Intertask Communication	–	–	By Deep Copy	By Reference	–	*Copy or reference*
Enforcement	Initialization	Compilation	Initialization	Compilation	Dynamic	*Compilation + initial.*

Figure 3. Features of Different Restricted Thread Programming Models.

explain in each case why multiple mechanisms were felt worthy of retention and how a Flexotask user should reason about which to use in particular applications.

Figure 2 shows the overall components of a Flexotask system. Figure 3 summarizes some key features of previous RTPMs and Flexotasks. We will use this summary in explaining how the features of Flexotasks were chosen. For more details, the reader is referred to published descriptions of Eventrons [31], Reflexes [33], Exotasks [5], StreamFlex [32], and RTSJ [14]. For simplicity, we describe Exotasks as they were presented in [5], although a later version [20] includes a unification with Eventrons that is carried forward and extended to the other models by the present work.

2.1 Threads, Flexotasks, and Flexotask Graphs

Although any RTPM ultimately provides restricted *threads*, a useful model is not obligated to make such threads directly manipulable. Of the preexisting RTPMs, only the RTSJ actually uses the thread as the unit of restriction. Restricting threads directly implies dynamic checking, since a thread can ultimate execute any body of code, and so we did not follow this approach. The other models define *tasks*, which are bodies of code rooted in an executable object, subject to statically checkable restrictions. Eventrons and Reflexes employ single tasks, while Exotasks and StreamFlex employ graphs of tasks with explicit channels of communication connecting individual tasks. In all four cases, restricted threads are managed by the system to execute the restricted tasks without themselves being directly visible to the programmer.

In Flexotasks we employ a graph of tasks. This relates directly to graph-based modeling systems, such as Simulink [30] and Ptolemy [22], that are often used to design real-time control systems, or to stream-based programming languages [34]. Fortunately, a single task is just a degenerate case of a graph, and so selecting a graph-based approach does not result in any fundamental loss.

2.2 Memory Management in Flexotasks

A key aspect of an RTPM concerns their restrictions on the use of memory. A partial characterization of this issue, in terms of short and long-term storage is shown in Figure 3. In Flexotasks, we unify the two-regions solution of Reflexes with the private heap solution of Exotasks. We provide two regions, with the more permanent of them being garbage collected.

Each Flexotask has thus a private memory area that is divided into a *heap* (as in Exotasks) and an optional *transient area* (as in Reflexes). The Flexotask heap is garbage collected on either a scheduled basis (when the task is not running) or on-demand (if memory is exhausted during task execution). Garbage collection can be disabled by the programmer if the application has a steady state that does require reclamation of heap data. The transient area is cleared each time the task's `execute()` method returns.

Allocations are directed to the private heap or the transient area based on the the class being instantiated, as in Reflexes. Allocations by Flexotasks are never directed to the public heap. We adopt the Reflex terminology in calling a Flexotask's heap-allocated classes *stable*, although they are not invariably long-lived as in the Reflex model. We adopt rules similar to those used by Reflexes to provide safety:

1. No transient class may inherit from a stable class.
2. Arrays with stable element type are stable and arrays with transient element type are transient.
3. Primitive arrays are handled according to how the programmer chooses to use transient storage (described below).
4. Fields of stable classes can only refer to stable objects.

These rules ensure that there will be no dangling pointers from objects on the Flexotask heap into the associated transient area. Other rules (described in more depth in Section 3) guarantee that the few permitted pointers into or out of either portion of the Flexotask memory area are strictly controlled.

The two memory areas can be used in one of the following ways:

1. All classes (including primitive arrays) are assumed stable.
2. Stable classes are explicitly declared by the programmer, either by implementing the `Stable` marker interface, or by listing them. The Flexotask object itself is necessarily stable. Other classes are considered to be transient. Primitive arrays are considered transient. A set of encapsulating classes (`StableBooleanArray`, etc.) is provided to enable primitive array usage on the private heap.
3. Stable status may be inferred. Starting with the Flexotask object itself, the transitive closure of classes appearing in instance fields are automatically marked stable and then all subclasses of stable classes are similarly marked stable. All other classes are considered to be transient.

The second option closely resembles Reflexes and StreamFlex, except that the stable region may actually be garbage collected. The first and third options give the semantics of Exotasks. The first is almost exactly equivalent to Exotasks' behavior, while the third confers the benefit of some reduced memory pressure in the event that some classes are found to be safely (and transparently) transient.

The semantics of the more restricted Eventron model can be achieved by simply avoiding allocation in the task. Our validation framework will optionally flag and reject any allocations (exempting `Throwable` objects, the management of which was particularly problematic in Eventrons), but employing this level of checking can also be omitted when there is enough slack to schedule occasional garbage collections.

Choosing whether to use a transient area explicitly or whether to simply heap-allocate everything will generally be predicated on

the kind of code reuse that is contemplated. When the internal data structures of the Flexotask use general-purpose classes (such as the Java collection framework), it is hard to satisfy the type checking rules for explicit declaration of stable classes and so the ability to treat everything as stable is very useful. On the other hand, as one fine-tunes the application to use more specialized data structures (which are usually more efficient and more predictable in their storage utilization), it eventually becomes easy to identify the stable classes and produce more regular memory usage behavior. The example presented in Section 4 went through exactly this evolution before settling on its current configuration, which makes use of the transient area.

Finally, Flexotasks may not make use of `Thread` or any of its subclasses, objects with finalizers, or any of the specialized `Reference` types (weak, soft, and phantom references).

2.3 Communication with Ordinary Threads

A second key differentiator of the precursor models concerns how they handled communication between the restricted thread and ordinary threads. Exotasks disallowed such communication entirely. Eventrons used one model of communication, enabling exchange of scalar values via a limited sharing of references to common objects. Reflexes and StreamFlex introduced an additional option, using a limited form of transaction. In Flexotasks, both the scalar values and transactional forms of communication may be used in the same graph, and the graph may optionally be marked *strongly isolated* to achieve the semantics of the Exotasks model.

2.3.1 Communication through Reference Immutable Objects

In understanding the first of the two Flexotask communication options, it is useful to establish the following recursive definition of *reference immutability*. We first define an *effectively final field* to be one that is either declared `final` or one that is both declared `private` and not mutated by any non-constructor method of its defining class. We then define a *reference immutable field* to be any field that is either (1) of a primitive type (possibly mutable) or (2) an effectively final reference field that is either (2a) `null` or (2b) containing a reference-immutable object. A *reference-immutable object* is defined as either a primitive array or an object that has only reference-immutable fields. This property is readily checked in an incremental fashion, as we revisit in Section 3.

Informally, any reference immutable object provides access to a graph of objects connected by references that cannot change but containing other fields that can change. Some leaf objects in this graph may be primitive arrays, whose elements can change, but not their extent.

By giving a Flexotask access to some reference immutable objects residing on the public heap (prevented from moving to avoid races with the public heap's garbage collector), the Flexotask can communicate with ordinary threads. The processes by which a Flexotask comes to have such references are described in Sections 2.7 and 3. Otherwise, this facility is based quite directly on Eventrons.

To obtain the semantics of the Exotask model, and the associated advantages of strict determinism, the programmer simply labels the Flexotasks of the graph as being strongly isolated, which prevents the creation of any channels of communication through reference immutable objects. However, we permit strongly isolated graphs to have references to completely (and recursively) immutable objects, since no communication can occur through such objects. This allows sharing of global constants (as with Exotasks), simplifying the code without compromising determinism.

2.3.2 Atomic Flexotasks

The second communication option, known as *atomic Flexotasks*, may be used whether or not the graph has been marked strongly isolated. The semantics of atomic Flexotasks is based on the similar Reflex capability, but is implemented differently. In this alternative, ordinary threads are given limited transactional access to state stored in the Flexotasks' private heaps. Changes to this state by the restricted thread running the Flexotask will always commit, but changes made by ordinary threads are unilaterally aborted if the restricted thread needs access to the state while the ordinary thread is still modifying it. The restricted thread then observes the state as it was before the ordinary thread attempted its modification, and the ordinary thread gets an exception.

To obtain these semantics, a programmer writes tasks that specialize the `AtomicFlexotask` abstract class. The programmer also creates one or more interfaces that extend the `ExternalMethods` marker interface. The `AtomicFlexotask` implementation must implement at least one such interface.

The Reflex and StreamFlex models implemented a similar transactional capability but did so in the Ovm virtual machine [28], which has a uni-processor design with threading controlled by the virtual machine. In that type of implementation, it was straightforward to implement preemption by the scheduler, which can immediately roll back any partially applied transactional change to the state of the task. In contrast, Flexotasks are implemented on IBM's WebSphere Real Time VM, which has a multi-processor design and (usually) maps Java threads to OS threads. In that type of implementation, it is very difficult to implement a roll-back approach for transactions, perhaps impossible to do so without introducing locking overheads that would substantially perturb execution predictability.

Consequently, we have adopted a roll-forward approach in which a method that is reachable by ordinary threads must commit its changes explicitly in an epilog. Prior to the epilog, the method is allowed to examine the task's heap state and can accumulate planned mutations in a local log, but should not actually perform these mutations. The epilog checks whether the state of the task has changed out from under the method, and if so throws an `AtomicException`. If not, the method is permitted to commit its changes, with the scheduler briefly locked out. Effective use of the facility is therefore predicated on the epilog being carefully designed to commit its changes efficiently. The Flexotask system handles this automatically by using class rewriting, as is discussed in Section 2.9.

Locking out the scheduler during the commit operation makes it vulnerable to being blocked *indirectly* by the garbage collector, because the committing thread is an ordinary one that can be paused by the collector while holding the lock. We thus require the thread to complete its commit and release the lock before yielding to the collector. Our implementation of this feature is currently incomplete and that is responsible for some of the outliers in the results of Section 5.

At runtime, the transformed `AtomicFlexotask` will reside inside the graph, while the rest of the Java program is given access only to a *guard* object that implements the same set of `ExternalMethods` interfaces as the `AtomicFlexotask`. The guard object delegates to the `AtomicFlexotask` while prohibiting the creation of any improper aliases.

2.3.3 Choosing and Combining Communication Mechanisms

As previously stated, the atomic Flexotasks option can be used in strongly isolated graphs but can also be combined with the Eventron-based mechanism of communication through shared objects. For the most part, the atomic Flexotasks option stresses precise knowledge of what was and wasn't communicated, and hence

is particularly useful when there needs to be coordination (in lieu of Java level synchronization) between restricted threads and normal threads. On the other hand, an atomic Flexotasks' external methods can delay the restricted thread during a roll-forward commit operation, and so this option is not a very good choice for bulk data transfer from normal to restricted threads. Combining the two options in the same program is thus a very powerful feature of Flexotasks: it is usually possible to partition the communication capabilities so that bulk data updates are done in a non-transactional way but notification and coordination of states are done transactionally. This strategy was used in the example of Section 4.

Combining atomic Flexotasks with strong isolation enables very tight control over the balance between isolation and coordination. This is appropriate when the amount of data transferred into the restricted portion of the program is small. Use of strong isolation by itself (no external communication) is appropriate for control applications that are very self contained and for which a high level of determinism is required (the original design point of Exotasks). In particular, strong isolation ensures that tasks are functionally deterministic in their inputs on channels.

Communicating scalars through shared reference immutable objects is useful when tight coordination between the restricted thread and ordinary threads is not needed. As will be seen in Section 2.4, effective use of this option may be enhanced through the use of the `notifyIfWaiting` facility taken from Eventrons. Otherwise, reliance on ordinary Java synchronization is to be avoided.

2.4 Synchronization Operations

For the purposes of this section, we use the term "synchronization operations" to mean specifically the `synchronized` block and `synchronized` methods in the Java language.

To reconcile the behavior of Eventrons, Reflexes, and StreamFlex (which prohibit or strongly discourage synchronization) with that of Exotasks which ignore it, Flexotasks offer two programmer-selectable options for dealing with synchronization operations: they are either permitted, or they are prohibited (in the latter case, the prohibition is via code analysis at development time and at initialization time).

When synchronization operations are permitted, they may or may not have any effect. If the task graph is strongly isolated, then all the objects it can access are either private to a task or completely immutable. Private objects can only be accessed by one thread (that of the task), so the synchronization operation will never have any effect. Synchronization on shared immutable objects by strongly isolated tasks is ignored; semantically it is as though they had been copied into the private heap.

Tasks that are not strongly isolated may execute synchronization operations which actually interact with other threads in the system and therefore may block. Generally speaking we discourage this when using Flexotasks, but there may be situations where backward compatibility with libraries makes it necessary.

To facilitate coordination when the restricted thread is not allowed to or should not use synchronization operations, we adopt the Eventrons `notifyIfWaiting` facility, which permits the ordinary thread to block and be notified in a non-blocking fashion by the restricted thread.

2.5 Scheduling

When the unit of restriction is a task graph rather than a thread, threads must be managed implicitly, and the executions of the tasks are thus *scheduled*. The precursor models handled this differently, but the Exotasks model had the most general solution, which was to make the scheduler pluggable, and to give it the responsibility to schedule not only tasks, but data movement between tasks and the garbage collection of tasks. This solution is adopted in Flexotasks. We carry forward the "time triggered" scheduler from Exotasks, which supports periodic execution, either of single tasks, or of graphs of tasks, with tasks being assigned specific time offsets within the period. We also intend to reimplement the data-driven scheduler of the StreamFlex model as a Flexotask scheduler, although this is future work. As in Exotasks, all restricted threads in Flexotasks actually belong to schedulers and the mapping of threads to tasks is usually not one-to-one (more typically, the number of threads reflects the level of real concurrency available in the hardware).

```
Main template:
  Set of task specifications
  Set of connection specifications
  Timing information for the scheduler
  Handling option for stable versus transient
  Optional list of stable classes
  Is allocation forbidden?
  Is synchronization forbidden?\[2mm]
Each task specification:
  Implementing class
  Input and output port types
  Is strongly isolated?
  Optional guard class (if Atomic)
  Timing information for the scheduler
Each connection specification:
  Data type
  Timing information for the scheduler
```

Figure 4. Contents of a Flexotask Template.

2.6 Construction of a Flexotask Program

As with Exotasks, a Flexotask program is instantiated from a *template* (called a "specification graph" in [5]). In contrast, other models used APIs to construct graphs (or isolated tasks) programmatically. The template idea facilitates the transfer of information about complex programs from development time to runtime, and the independent development of tools that help in the construction of such programs (the template can be constructed with a graphical editor and stored in a special file format). Templates also help concretize the initialization-time validation step by providing a single call that validates the template and returns a handle to the resulting graph. On the other hand, simple programs (e.g. single-task programs) don't especially benefit from templates. Flexotasks mitigate the bias toward complex programs just as Exotasks did by having an API for the construction of templates at runtime. The Flexotask system provides veneers for programmers used to the Eventron and Reflex style of programming. The veneers provide a single operation that constructs a template and validates it producing the desired single-task graph.

The contents of the Flexotask template are summarized in Figure 4. As can be seen, most of the features discussed in earlier sections (and connection properties, which are discussed in Section 2.8) are present in this template. The runtime method that instantiates a Flexotask graph uses as inputs (1) a template, (2) an optional parameter map (discussed next), (3) a choice of scheduler, and (4) optional platform-specific task characteristics to pass to the scheduler. We will not discuss (4) due to lack of space; that aspect is taken directly from Exotasks. Implicitly, the code of all classes mentioned in the template is also an input, and this code is validated as part of instantiation.

2.7 Parameter Maps

A parameter is simply an object that is passed to each task at runtime as part of its initialization, and a parameter map is a map from task names to parameters that is provided to the Flexotask system, along with the template, to instantiate the graph. Parameters were not originally a feature of any of the precursor models, although Exotasks adopted the idea in a later version [20]. They serve two purposes. They promote reuse of tasks, eliminating tasks that are only small variations on each other. They also permit the controlled introduction references to reference immutable objects on the public heap to facilitate communication through such objects as was discussed in Section 2.3.1. To support the second goal, a parameter passed to a strongly isolated Flexotask is (deeply) cloned, while one passed to an ordinary Flexotask is checked for reference immutability and passed by reference.

2.8 Intertask Communication

Since only Exotasks and StreamFlex (among the precursor models) used a graph as the unit of restriction, only they provide possible models for intertask communication. There were two key differences between the two original models.

1. In Exotasks, communication is by deep copy (which avoids introducing aliases). StreamFlex communicates by reference while avoiding aliases in a different way, which requires that transmittable types be restricted to classes with only primitive fields (this includes primitive arrays).
2. Exotask connections are a single-stage buffer (the most recently sent value can differ from the most recently received, but no other "in transit" values are kept). In contrast, StreamFlex connections are parameterizable by depth and can store a limited but flexible number of values.

StreamFlex achieves its model by introducing a third type category, the *Capsule*, which is mutually exclusive with stable and transient and is limited to the transmittable types. Stable classes may not point to capsule classes. A special memory area separate from any task area contains all capsules, and a capsule is deallocated when the current owning task returns without transmitting it.

The Flexotask system provides both styles of connection, based on type. Connections are assigned a data type (as in both precursors) and the type of each connection may be either stable or capsule (but not transient). A list of capsule classes may optionally be supplied as part of the template. There is, therefore, a clean division into connections of (purely) stable type, which are managed by deep copy, and those of capsule type (which are managed by reference).

In Flexotasks, connections are always single-stage buffers, as in the Exotask model, but a specialized task called a *buffer task* may always be interposed (so the sending task has a single-stage connection to the multi-stage buffer task which then has a single-stage connection to the receiving task). Buffer tasks also subsume the more limited single-stage *communicators* in the Exotask model.

The implementation of capsules and buffers for Flexotask will be completed as future work and is not evaluated in this paper. In the present prototype, only stable classes can be sent on connections, and so deep copies are always made.

2.9 The Flexotask Infrastructure

The Flexotask system comes with development tool support integrated in the Eclipse IDE as well as virtual machine support implemented in the IBM WebSphere Real Time virtual machine. Figure 5 gives an overview of the Flexotasks infrastructure. Programs are developed under Eclipse, validated, and selectively rewritten if the program contains atomic Flexotasks (as discussed in Section 2.3.2).

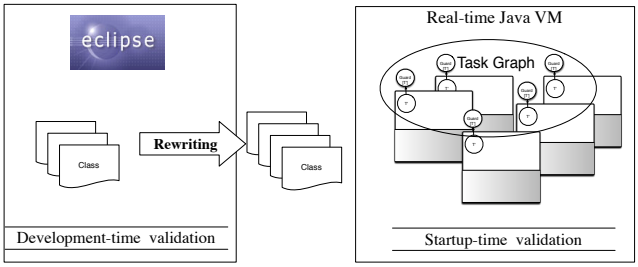


Figure 5. Flexotask Infrastructure. Programs developed within the Eclipse IDE are validated at development time with an integrated bytecode verifier. Class files are rewritten to include support for transactions. The JVM has an initialization-time validator that performs a data-sensitive analysis of a Flexotask graph.

Then, after the code is loaded into the JVM and the program partly initialized, the code (including rewritten classes) is validated again using run-time information about arguments and static variables.

This architecture adopts the best of the precursor models. The Reflex and StreamFlex models performed validation as part of compilation at development time, which has the advantage of early detection of errors. The Eventron and Exotask models perform validation as a step during program initialization, after class initialization and some object construction has already been done. The resulting data-sensitive analysis is more precise and admits a larger set of valid programs. In addition, compile-time enforcement alone does not permit untrusted code to be run in a restricted thread, since there is no guarantee that it went through the appropriate compiler. To achieve the advantages of both kinds of checking, we do the checking twice, with some of the (necessarily) more conservative checks during development reduced to the status of warnings. Note that `NoHeapRealtimeThread`s employ continuous runtime checking, which we rejected, since it makes it much harder to determine whether a program is correct.

3. Validating Flexotasks

Validation of a Flexotask graph occurs both at development time and at initialization time. To ensure that similar rules are enforced in both places, both validators are built on a common framework. The base of this framework is an engine that performs Rapid Type Analysis [7] (RTA) to build a summarized call graph rooted in initially reachable methods of the Flexotasks in the graph. The engine examines every bytecode of every reachable method. At initialization time, for security, these bytecodes are found in the already-loaded and verified classes of the JVM (and classloading is forced by the validator to ensure initialization). At development time, bytecodes are read from the classfiles in the classpath using a conventional classfile parser.

The rules to be enforced include (1) separation of classes into stable and transient in a way consistent with the rules of Section 2.2, (2) enforcement of the strong isolation property (Section 2.3) if requested, (3) checking that parameters (Section 2.7) passed by reference are reference immutable, (4) checking that references acquired through `static` fields are reference immutable (or fully immutable if strong isolation was requested), and (5) checking optional prohibitions on allocation (Section 2.2) and synchronization (Section 2.4). In addition, the development time validator also rewrites any atomic Flexotasks (Section 2.3.2) to conform to a special set of rules, and the initialization time validator checks the rewritten code for adherence to the rules (to avoid counterfeiting). The special rules for atomic Flexotasks require that they always execute their epilog and that they perform all mutations in the epilog.

Checking that the epilog commits its changes "efficiently" is omitted in the interest of practicality, although the rewriter attempts to achieve this goal.

The initialization time validator, as with Eventrons and Exotasks, performs its checking for reference (or full) immutability in a *data sensitive* fashion. That is, it 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 Flexotask code. Initially, the set O consists of objects passed as parameters, but it is augmented when a `static` field is accessed with a `getstatic` bytecode. 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 expand both sets up to a fixpoint. When fields or objects are added to either set, they are checked for reference immutability (or full immutability if strong isolation is requested).

The development time validator has two challenges not faced at initialization time. First, it does not know the actual objects passed as parameters, nor does it know the actual objects present in `static` fields so it must substitute a class-based analysis that is more conservative. The stable/transient rules do not change because they are based on classes rather than objects. However, because the set of potentially live classes inferred by the RTA engine may be larger at development time, some potentially transient classes may not be identified.

Checks for reference immutability, however, require a class-based definition. Thus, a *reference immutable field* is either of primitive type or is effectively final and assignable only from reference immutable classes (among the live classes). A *reference immutable class* is either a primitive array or one with only reference immutable fields. To find the relevant live classes for purposes of this algorithm, the class initialization methods must be examined as well as other methods analyzed by the RTA engine.

A second challenge faced by the development time validator is that of incomplete information. The development time validator analyzes each template that it finds and any classes referenced by it, although this set of classes may be incomplete. It also analyzes (individually) any Flexotask classes not referenced by any template, on the grounds that they may later be so-referenced. It continuously posts error and warning indicators in the Eclipse view, which may be temporarily (or permanently) suppressed by annotations. This is necessary since the initialization time validator, with more information, will be more precise and may permit things that the development time validator flags as suspicious.

4. Example: Collision Avoidance

To illustrate the usage of Flexotasks, we present the example of an aircraft collision detection algorithm (we thank the authors of [35] for making their source code available). Collision detection is performed by a single atomic Flexotask (Section 2.3.2) which periodically processes the latest frame it has received. Each frame contains aircraft call signs paired with the positions and direction vectors of the aircraft. The output of the algorithm is a warning each time any pair of aircraft are on a collision course. In our implementation, the aircraft call signs, positions and direction vectors are all provided by a separately running plain Java thread that simulates this data based on symbolic execution of a set of equations describing plane trajectories.

This example illustrates the simultaneous use of both kinds of external communication, as was discussed in Section 2.3.3. Because each frame potentially contains a fair amount of data, we do not incur the overhead of transmitting this information atomically. Rather, we rely on a ring buffer of `RawFrame` data structures which is shared between the Flexotask and the simulator. These handle the bulk data transfer. The coordination around availability of frames for use by either the Flexotask or the simulator is handled by atomic Flexotask methods. The invariant maintained by these methods is that no frame is ever used simultaneously by both.

The example also illustrates the tradeoff between the use of pure heap allocation and stable/transient allocation discussed in Section 2.2. We were able to get the example working quickly using pure heap allocation, which was necessary because the `StateTable` needed by the Flexotask used Java collection classes. Then, to optimize the example, we replaced the Java collection classes with a small number of custom classes that could be made stable under the rules of section Section 2.2. In all, this example required six classes to be labelled stable (in addition to the Flexotask itself, which is implicitly stable, and arrays of stable classes, which do not have to be labelled).

A sketch of the main program is shown in Figure 6, the relevant Flexotask code is shown in Figure 7, and the shared reference immutable data structure is shown in Figure 8. A non-atomic Flexotask would simply implement the `Flexotask` interface, which requires it to provide a one-time `initialize()` method and a periodic `execute()` method. To use transactions, in contrast, the task extends the `AtomicFlexotask` abstract class (requiring the same methods to be provided but furthermore supplying some necessary behavior for transactionality). Methods reachable via the guard are indicated via a separate interface that extends the `ExternalMethods` marker interface. Any class providing the atomic methods invoked by the plain Java thread must implement such an interface. Furthermore, each of the methods declared in this interface are required to throw `AtomicException`.

Figure 6 illustrates the reconstruction of a template from one previously prepared in the development environment, and stored as an XML file. The `FlexotaskXMLParser` class provides capabilities for loading templates and creating a Java object representation. The template encapsulates the name of the single task ("DetectorTask"), its implementation class (`DetectorTask`), the result of running the development-time rewriter on its code, and the list of stable and transient classes that were marked and checked at

```
// To obtain template
InputStream in = DetectorTask.class
   .getResourceAsStream("Detector.ftg");
FlexotaskTemplate spec =
   FlexotaskXMLParser.parseStream(in);

// To instantiate graph with sharing
Map parameters = new HashMap();
RawFrameArray sharedArray = new RawFrameArray();
parameters.put("DetectorTask", sharedArray);
FlexotaskGraph graph =
  spec.validate("TTScheduler", parameters);

// To obtain reference to Detector
DetectorGuard detector = (DetectorGuard)
  graph.getGuardObject("DetectorTask");

graph.start();

// To communicate a new frame:
int frameIndex = detector.getFirstFree();
if (frameIndex == -1) { ... no buffer available }
frames.get(frameIndex)
  .copy(lengths, callsigns, positions);
detector.setNextToProcess(frameIndex);
```

Figure 6. Constructing a Flexotasks graph.

```
interface DetectorGuard
     implements ExternalMethods {
  void setNextToProcess(int frameIndex)
    throws AtomicException;
  int getFirstFree() throws AtomicException;
}

class DetectorTask extends AtomicFlexotask
         implements DetectorGuard {
  private StateTable state;
  private RawFrameArray frames;
  private int nextToProcess;
  private int firstFree;

  void initialize(..., Object parameter) {
    frames = (RawFrameArray) parameter;
    state = new StateTable();
  }

  void execute() {
    if (nextToProcess != firstFree) {
      cd = new Detector(state,
         Constants.GOOD_VOXEL_SIZE);
      cd.setFrame(frames.get(nextToProcess));
      cd.run();
      nextToProcess = firstFree =
         increment(nextToProcess);
      // increment 'increments' modulo a ring size
    }
  }

  int getFirstFree() throws AtomicException {
     int check = increment(firstFree);
     if (check == nextToProcess) {
       return -1;
     }
     int ans = firstFree;
     firstFree = check;
     return ans;
  }

  void setNextToProcess(int nextToProcess)
       throws AtomicException {
     this.nextToProcess = nextToProcess;
  }
}
```

Figure 7. Implementing an Atomic Flexotask.

development time. The `validate()` method selects a scheduler (pluggable, as in the Exotasks [5] system) and produces a runnable `Graph`. The `parameters` argument passes in the shared object that will be used for communication. The `getGuardObject()` method retrieves a reference to the guard for the task that can be used to invoke task atomic methods by the plain Java thread. This object is automatically generated from the `DetectorTask` class and its transactional interface `DetectorGuard`.

Figure 7 shows the class `DetectorTask` which extends `AtomicFlexotask` and implements `DetectorGuard`. The `initialize()` method establishes the sharing relationship by storing the `RawFrameArray` parameter and initializes the `StateTable` to store stable state. In a graph with more than one Flexotask this method would also pass in representations of the task's ports to use for intertask communication, and these would be saved in instance variables. The `execute()` method establishes the frame to be processed and analyzes the data it contains. Each invocation of `execute()` will create transient objects of types `Detector` and its numerous dependent working

```
class RawFrameArray implements Stable {
 private final ImmutableArray frames;

 public RawFrame get(final int i) {
   return (RawFrame) frames.get(i);
 }

 public RawFrameArray() {
   RawFrame[] innerArray =
     new RawFrame[MAX_FRAMES];
   for (int i = 0; i < MAX_FRAMES; i++)
     innerArray[i] = new RawFrame();
   frames = new ImmutableArray(innerArray);
 }
}
```

Figure 8. A reference immutable data structure.

objects, as well as new stable objects to represent call signs and vectors that will be stored in the `StateTable`. The implementation of the `getFirstFree()` and `setNextToProcess)` methods represents the code as the programmer would write it, i.e., the code *before* being rewritten by the development tools. The actual code executed at runtime is instrumented at the bytecode level so as to redirect all mutations (and subsequent reads thereof) to a transaction log that is then committed by rolling forward the mutations in an epilog, as described earlier. As such, the transactionality of a method is completely transparent to the programmer, except that the invoking program should catch and handle the `AtomicException`.

As previously mentioned, the `RawFrameArray` data structure must be reference immutable. The code of this class is shown in Figure 8. For reference immutability to hold, the `RawFrame` data structure must first be reference immutable, which is easily accomplished since this class is just a set of primitive arrays connected to their parent object by `final` references. But, an array of references is not normally reference immutable. This problem is solved in the Flexotask system as it is in Eventrons by using a special `ImmutableArray` class. This class's constructor copies its argument and does not subsequently leak it, ensuring that no mutations occur to the enclosed array after construction.

5. Evaluation

We evaluate Flexotasks on predictability and performance, both important properties of real-time systems. We employ two applications, a high frequency reader and an airline collision detector. We also report on analysis time for both applications. The experiments were performed using an experimental variant of the IBM WebSphere Real Time (WRT) virtual machine [3]. The virtual machine includes support for high resolution timing on real-time kernels, a real-time garbage collector [6], and an implementation of the RTSJ. WRT also includes experimental features added for Eventrons, providing object pinning/unpinning and the ability to exempt certain threads from being paused by the global heap garbage collector. The Flexotasks implementation extended the Exotasks implementation [5, 20], which provides private per-task heaps, and deep copying between heaps. For the current work we added support for transient allocation and roll-forward transactions.

As our execution platform, we used an IBM blade server with 4 dual-core AMD Opteron 64 2.4 GHz processors and 12GB of physical memory. The operating system used was Linux (kernel version 2.6.21.4 in the RHEL 5 real-time configuration).

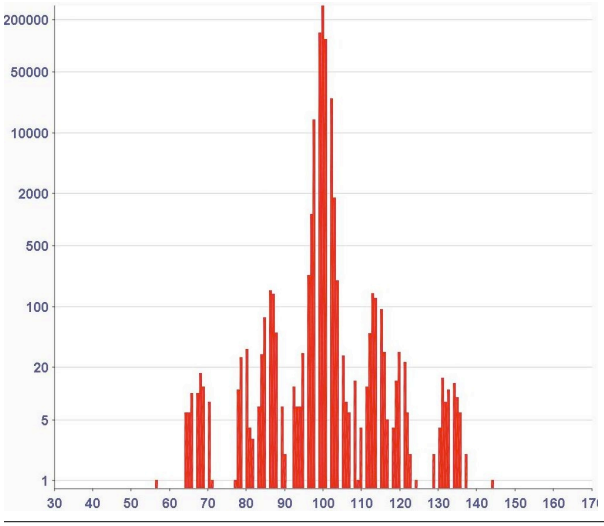


Figure 9. Frequencies of inter-arrival times of an atomic Flexotask scheduled with a period of 100 μs, executing concurrently with (1) a plain Java thread communicating by transactional invocations, and (2) a plain Java thread simulating regular memory consumption by continuously allocating at 2MB per second. The x-axis depicts the inter-arrival time of two consecutive executions in microseconds. The y-axis depicts the logarithm of the frequency.

5.1 Predictability

We evaluate predictability of a *high frequency reader* benchmark (540 LOC). The application has an atomic Flexotask scheduled at a period of 100μs. At each periodic execution, the task reads available data on its input buffer in circular fashion into its stable state. A plain Java thread that runs continuously feeds the Flexotask with data on its input buffer by invoking a transactional method on the task approximately every 20 ms. Out of 3,000 invocations of the transactional method, 516 of them aborted, indicating that the atomic Flexotask feature was being heavily exercised. To evaluate the influence of computational noise and garbage collection, another ordinary Java thread runs concurrently, continuously allocating at the rate of 2MB per second, using 48 byte objects and maintaining a live set of 40,000 objects. To avoid perturbations caused by the JIT-compiler, we ran this test in interpreted mode.

Figure 9 shows a histogram of the frequencies of inter-arrival times of the periodically scheduled atomic Flexotask, i.e., the time between two consecutive executions. The figure contains observations covering almost 600,000 periodic executions. As can be seen in the figure, all observations of the inter-arrival time are centered around the scheduled period of 100 μs. Overall, there are only a few microseconds of jitter to be seen in the figure, with inter-arrival times ranging from 57 to 144 μs.

5.2 Performance

To evaluate performance, we considered a larger application in the form of the collision detector described in Section 4. The collision detector (30 KLOC) consists of three threads running concurrently: the `DetectorTask` running at a period of 20ms, the simulator thread generating flight data and communicating with `DetectorTask` every 20ms, and the 2Mb/second allocator thread described in the previous section. Because the 20ms period allows more slack than in the predictability test (necessary both for realism and to allow the simulator to keep up), we instructed the allocator thread to keep 150,000 objects live in order to cause more garbage collection overhead and a greater opportunity for conflict.

	High Freq. Reader	Collision Detector
Compile-time Validation	33 *ms*	173 *ms*
Bytecode Rewriting	10 *ms*	51 *ms*
Startup-time Validation	332 *ms*	699 *ms*

Figure 11. Analysis times for the two benchmark applications.

To have a baseline with which to compare our measurements, we implemented two additional variants of the collision detector, respectively a plain Java variant where the time critical thread was just an ordinary thread, and a RTSJ variant making use of scoped memory areas. The plain Java version was run both under a normal garbage collector (the non-realtime IBM J9 collector with default parameters), and under the real-time garbage collector of the WRT VM. For this experiment, we measured performance as time taken by the detector to process a frame.

Figure 10 shows the results of our measurements. For the plain Java variant with a non-realtime collector (a), the worst-case observed processing time over the entire run was around 28 milliseconds which is not surprising given that the virtual machine uses an ordinary non-real-time garbage collector. With the real-time garbage collector, this number declines substantially but this comes at the expense of mutator utilization, thereby increasing the average processing time. The smaller but non-negligible jitter in (b) happens because a varying number of the garbage collector's work quanta can fall within the relatively long processing times (about 4ms) for the detector.

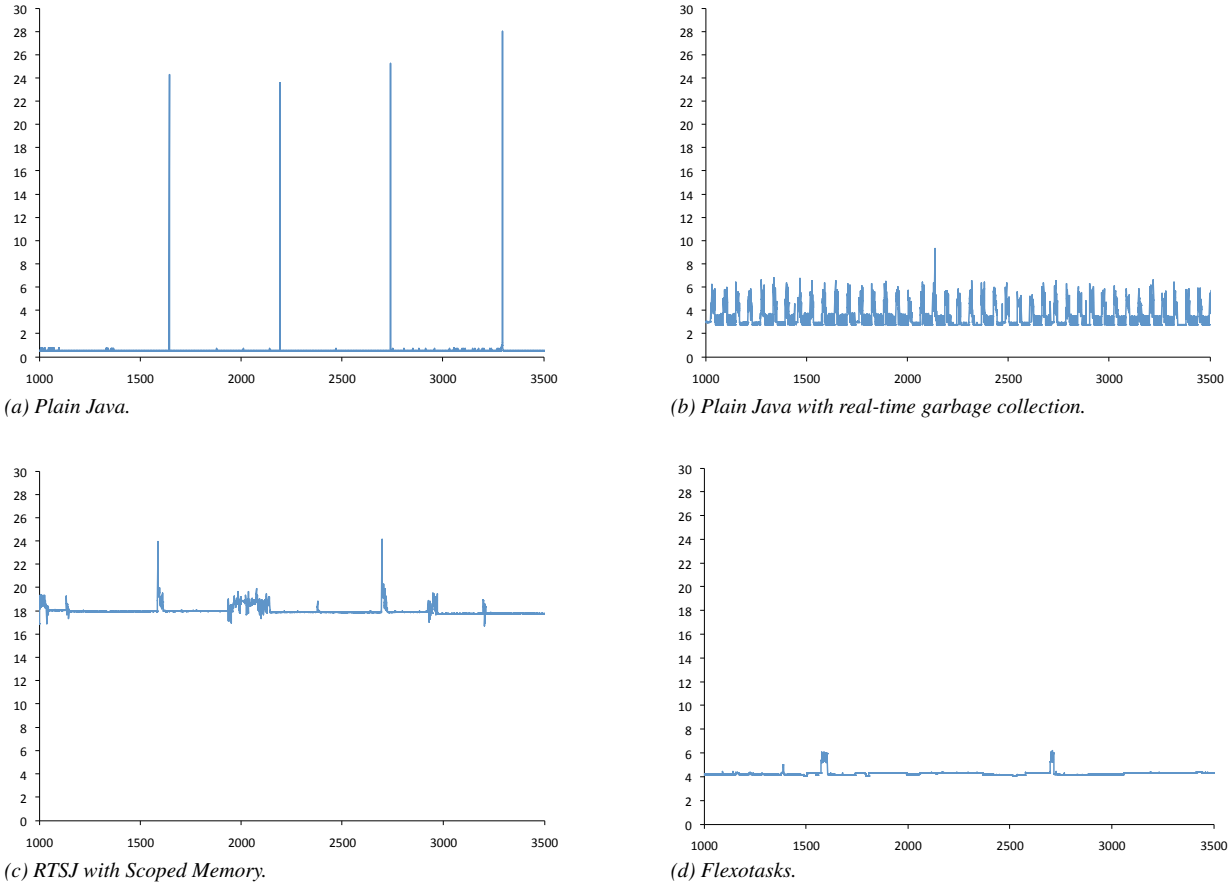
Both the RTSJ variant (c) and Flexotasks (d) are largely impervious to such interference. We note that they are not entirely impervious: each has two spikes that correspond to garbage collections though other garbage collections in the run pass without incident. In the Flexotask version (d), the two spikes are due to a thread being preempted while holding a lock needed by the scheduler, a known problem that will be addressed through the design discussed in Section 2.3.2. In the RTSJ version the spikes are unexplained but probably represent a flaw in the implementation of the VM.

Clearly, the best-case performance time of the RTSJ variant is significantly slower than that of any other variant. We attribute the slowdown to the dynamic checks imposed during runtime by the virtual machine to ensure safety of pointer manipulation when using the RTSJ scoped memory areas.

In summary, we have shown how four different variants of our collision detector application are subject to a varying degree of interference caused by the presence of garbage collection. As expected, the plain Java variant experiences infrequent, but large, latencies when running on a non-real-time collector and much smaller latencies but still noticeable jitter when running on the real-time collector. The RTSJ variant has low jitter but poor average-case performance. In contrast to these, the Flexotasks variant runs at high performance with the smallest amount of jitter.

5.3 Static Analysis Performance

We measure the time needed for validating our two benchmarks at compile time code and at initialization time. Whereas initialization time validation was performed on the platform described above, compile time validation occurred on a development machine running JDK 1.5.0_07-87 on a Intel Core Duo, 2.16GHz with 2GB of physical memory. Figure 5.2 shows the empirical measurements of the time to perform the various stages of the code analysis. As can be seen, it takes twice as long to validate the collision detector. This is not surprising given the difference in code size. The longer time taken at initialization time primarily reflects the fact that "val-



(a) Plain Java.

(b) Plain Java with real-time garbage collection.

(c) RTSJ with Scoped Memory.

(d) Flexotasks.

Figure 10. Comparing performance of four different variants of the collision detector benchmark. The x-axes show the data frames processed, numbering from 1 (only a representative set of frames are shown), and the y-axes the processing time in milliseconds for the individual frame.

idation" actually includes the time to instantiate and schedule the Flexotask graph in addition to simple checking. Also, the checking is more detailed since it is done in a data-sensitive fashion.

5.4 Software Engineering Aspects

We briefly comment on our experience refactoring the collision detector application to use Flexotask APIs. The collision detector code obtained from [35] consisted of 195 files, containing 241 classes (around 30 KLOC), and employed RTSJ APIs. Converting it to Flexotasks required modification of 8 files and adding `Stable` declarations to 7 classes. The main changes were in the setup portion of the application: the original version had code for creating RTSJ-style real-time threads, whereas the Flexotasks version created a one-node graph. The other main change was in the communication between ordinary Java threads and the real-time task. In order to pass the validation phase, we had to ensure that objects shared between the two were reference immutable. Finally, in a number of places where the RTSJ code had to resort to reflective invocation, the calls were transformed into normal allocations of stable classes in the Flexotasks version. The effort going from the earlier version of the code was modest and made the code easier to understand.

6. Conclusion

In this paper, we have introduced Flexotasks, a restricted thread programming model for Java, which unifies four previously existing models. Flexotasks provide programmers with a single framework for developing real-time programs observing timing constraints tighter than those possible with state-of-the-art real-time garbage collection algorithms.

As a unified framework, Flexotasks provide flexibility to choose between a set of programming abstractions supporting different sets of tradeoffs and advantages in order to meet the timing constraints and functionality requirements of a given application.

We implemented our framework in the form of a single API supporting all four models, combined with development time and run-time tools integrated into the Eclipse IDE. These tools ensure safety guarantees of memory operations by verifying the code at the bytecode level against the different sets of type constraints, and thus enable execution free of expensive runtime memory checks as required when using RTSJ scoped memory. To enable non-blocking communication between time constrained tasks and time oblivious code, we have implemented support for atomic methods through bytecode rewriting, thereby making it transparent to the programmer. Furthermore, we have extended previous work by providing support for multi-processors using a roll forward transaction log approach.

Acknowledgments

This work is supported in part by NSF grants 501 1398-1086 and 501 1398-1600.

References

[1] AICAS. The Jamaica virtual machine, www.aicas.com.

[2] Austin Armbuster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(1):1–49, 2007.

[3] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT)*, pages 249–258, 2007.

[4] Joshua Auerbach, David F. Bacon, Florian Bömers, and Perry Cheng. Real-time music synthesis in Java using the Metronome garbage collector. In *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, August 2007.

[5] Joshua S. Auerbach, David F. Bacon, Daniel T. Iercan, Christoph M. Kirsch, V. T. Rajan, Harald Roeck, and Rainer Trummer. Java takes flight: time-portable real-time programming with Exotasks. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, pages 51–62, San Diego, CA, 2007.

[6] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, LA, January 2003.

[7] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 31, pages 324–341, October 1996.

[8] BEA. Weblogic real time. www.bea.com, 2006.

[9] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for Real-Time Java. In *Embedded Software Implementation Tools for Fully Programmable Application Specific Systems (EMSOFT)*, pages 289–305, 2001.

[10] Edward G. Benowitz and Albert F. Niessner. Experiences in adopting real-time java for flight-like software. In *Proceedings of the International workshop on Java technologies for real-time and embedded systems (JTRES)*, pages 490–496, 2003.

[11] Edward G. Benowitz and Albert F. Niessner. A patterns catalog for RTSJ software designs. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 497–507, 2003.

[12] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with non-heap memory in the real-time specification for Java. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 361–369, 2003.

[13] Greg Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, and Frederic Parain. Mackinac: Making hotspot real-time. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 45–54, 2005.

[14] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[15] Angelo Corsaro and Ron K. Cytron. Efficient memory reference checks for real-time Java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.

[16] Mike Fulton and Mark Stoodley. Compilation techniques for real-time Java programs. In *Proc. International Symposium on Code Generation and Optimization*, 2007.

[17] Sven Gestegard Robertz, Roger Henriksson, Klas Nilsson, Anders Blomdell, and Ivan Tarasov. 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)*, pages 104–110, 2007.

[18] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[19] IBM. DDG1000 Next Generation Navy Destroyers, www.ibm.com/-press/us/en/pressrelease/21033.wss, 2007.

[20] IBM Corporation. IBM Expedited Real Time Task Graphs. www.alphaworks.ibm.com/tech/xrtgs, 2007.

[21] Nicolas Juillerat, Stefan Müller Arisona, and Simon Schubiger-Banz. Real-time, low latency audio processing in Java. In *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, August 2007.

[22] E.A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, EECS Department, University of California, Berkeley, 2003.

[23] Ingo Molnar and Thomas Gleixner. The RT-PREEMPT patch set for Linux.

[24] Albert F. Niessner and Edward G. Benowitz. Rtsj memory areas and their affects on the performance of a flight-like attitude control system. In *Proceedings of the International workshop on Java technologies for real-time and embedded systems (JTRES)*, pages 508–519, 2003.

[25] Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 378–404, Darmstadt, Germany, July 2003.

[26] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.

[27] Filip Pizlo and Jan Vitek. An empirical evalutation of memory management alternatives for Real-time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006.

[28] Purdue University. The Ovm virtual machine, www.ovmj.org.

[29] Fridtjof Siebert. The impact of realtime garbage collection on realtime Java programming. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 33–40, 2004.

[30] Simulink. www.mathworks.com/products/simulink. 2007.

[31] Daniel Spoonhower, Joshua Auerbach, David F. Bacon, Perry Cheng, and David Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*, pages 283–294, 2006.

[32] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2007.

[33] Jesper Honig Spring, Filip Pizlo, Rachid Guerraoui, and Jan Vitek. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, 2007.

[34] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction (CC'02)*, April 2002.

[35] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, Lisbon, Portugal, December 2004.