The Gnu Ada Runtime Library (GNARL): Design and Implementation

E.W. Giering III, T.P. Baker*
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019
Internet: giering@cs.fsu.edu Phone: (904)644-3441

Abstract

The GNu Ada Runtime Library (GNARL) is an implementation of Ada 9X tasking. It is designed to support the Gnu-NYU Ada Translator (GNAT), and is part of the GNAT distribution. This report describes the design of the interface which GNARL presents to compiler-generated code, some of the alternatives considered, and the motivation for choosing the current design. It also discusses the GNARL implementation, how it is intended to be layered over the services of an off-the-shelf commercial operating system, and the expected compiler translation of various Ada 9X tasking constructs.

1 Introduction

The GNu Ada Runtime Library (GNARL) is an implementation of Ada 9X tasking for the Gnu-NYU Ada Translator (GNAT). The goal of the GNARL development effort is to provide an easily portable, publicly accessible implementation of the full Ada 9X[1] tasking semantics.

The development GNARL is a cooperative effort of the Florida State University POSIX/Ada Real-Time (PART) project with the NYU GNAT development team. The goal of the PART project is to advance the development and standardization of Ada language bindings to the IEEE POSIX standards, a family of application program interfaces for operating system services. A central concern of the PART project has been reconciliation of the Ada tasking model with the proposed standard POSIX thread services[7] (Pthreads). Pthreads provides C-language applications with a capability similar to Ada tasking. To validate the practicality of layering Ada tasking over Pthreads, the PART project developed an Ada 83 tasking runtime system (RTS)[4, 5, 6]. This Pthreads-based RTS provided the starting point for GNARL.

The design objectives of GNARL, in order of priority, are:

1. Semantic correctness. GNARL must support the full core tasking semantics, and as much of the Real-Time Systems and Systems Programming annexes as is permitted by other constraints.
2. Timeliness. Development should be incremental, so that working partial implementations can be delivered early. The RTS and compiler development should proceed in parallel. This means the RTS responsibilities should be cleanly separated from those of the compiler. Ada features that early versions of the compiler are not likely to be able to translate should be avoided. Ideally, it should be possible to compile and test it using an existing Ada 83 compiler. It should use the existing PART RTS code for leverage, to speed development.
3. Compatibility with GNAT. The GNAT library mechanism requires the recompilation of each library unit spec named in a **with** clause. This exacts a penalty for large monolithic package interfaces. GNARL should be partitioned so as hide information that the compiler does not need to know, including information about tasking features that are not used by a particular compilation. The GNAT strategy for implementing tasking is based on tree-to-tree translation, converting tasking constructs in the intermediate syntax tree representation into equivalent Ada language constructs, with interspersed RTS service calls. Thus, the GNAT interface should be expressible in terms of ordinary Ada packages whenever possible.
4. Portability. GNARL should be written in Ada,

*Partially funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through HQ U.S. Army CECOM, Software Engineering Directorate.

[1]The term "Ada" is used without further qualification to refer to Ada 9X, as described in draft 4.0 of the Ada Reference Manual (ARM)[1]. The term "Ada 83" is used to refer to the original language standard[11].

with target-specific code cleanly isolated and kept as small as possible. It should be possible to produce configurations that run both over commercial off-the-shelf operating systems and on bare machines with minimal modifications. Initially, it should be supported on several commonly used operating systems.

5. Research and technology transfer. GNARL should serve as a test-bed for implementation ideas, providing experience that will be useful in designing other implementations. Lessons learned should be reported promptly, and code made publicly available. Among other experimental goals, GNARL should provide a basis for measuring the overhead imposed by implementing Ada over Pthreads.
6. Efficiency. GNARL should be as efficient as possible, consistent with the other objectives. The design should allow for future optimizations, including inline expansion and optimization of RTS calls. This means using ordinary procedure calls in the interface, as opposed to traps or calls via procedure pointers.

This report describes the way in which GNARL cooperates with GNAT and an underlying operating system to implement the new Ada tasking features, including protected types and asynchronous transfer of control (ATC).

2 Interface to Tasking Services

Since GNARL and GNAT are being developed by two geographically separated teams, there is a premium on clear division of responsibilities between the tasking runtime services, which are provided by GNARL, and the generation of code for tasking constructs, which is done by GNAT. To this end, we developed the GNARL Interface (GNARLI) document, which serves as contract between GNARL and GNAT. The GNARL interface is shown at the top level in Figure 1.

The original intent was for GNARLI to be an upward-compatible extension of earlier attempts to converge on a common RTS interfaces for Ada 83, the Common Ada Run-Time System interface (CARTS)[2] and the Model Run-Time System Interface (MRTSI)[3]. This goal dominated several early design decisions. As a result, GNARLI is similar to CARTS and MRTSI in several respects. It attempts to implement Ada tasking as directly as possible, isolating as much of the tasking semantics as possible in the RTS and providing the compiler with as much help as possible in compiling task constructs. It is a procedural interface. The compiler communicates with the RTS through procedure and function calls, without direct reference to RTS data structures aside from the parameters of the RTS subprograms. The RTS data structures may be in a separate address space, protected from access by the application. The direction of call is always from application code to the RTS. There are no *callbacks*, where the RTS uses a pointer passed to it by the application, to make calls to application subprograms[2].

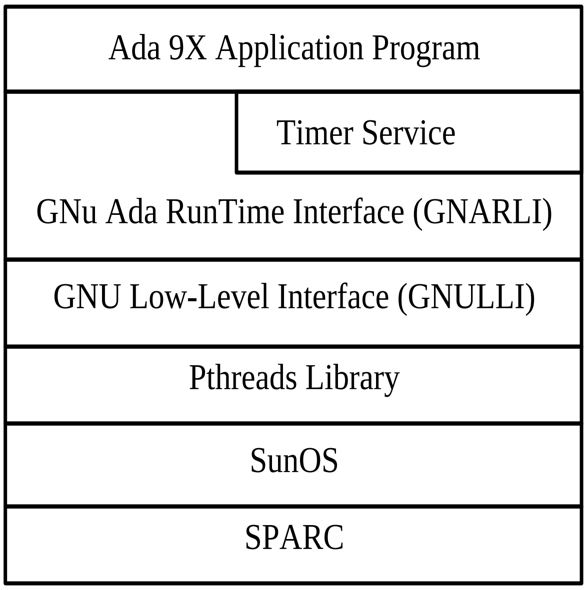

Figure 1: GNARL components

There were some places where we seriously considered departing from the CARTS/MRTSI model. One of these, which we resisted completely, is the use of callbacks. The other is the strict separation of RTS data structures, on which we made an exception. Both of these are related to the implementation of protected objects, and are discussed at greater length in Sections 5 and 6.

3 Interface for Timer Service

The Timer Service layer provides support for the various forms of the **delay** statement, including delay clauses in **select** statements. The interface is a sort of appendix to the rest of the GNARLI. It is implemented as a set of ordinary Ada packages, which are pure clients of the rest of the GNARLI. They do not depend on any internals of the rest of the GNARL, nor does the rest of the GNARL depend on any timer services. There are no references to time in the rest of the GNARLI or its implementation. In this respect, GNARLI is unlike MRTSI or CARTS, both of which have `Duration` parameters on some rendezvous operations.

The original reason for separating out Timer Service from the rest of GNARLI is the need to support delays on multiple time types. Ada 9X introduces the **delay**

[2]The one borderline exception to this rule, in MRTSI and CARTS as well as GNARLI, is task creation, in which the compiler passes the RTS the address of a procedure corresponding to the task body.

until clause, which allows a task to delay until some absolute time has arrived. A **delay until** clause can appear anywhere a **delay** clause can. This includes the various forms of **select** statement, such as timed entry calls and selective waits, where it is used as a timeout. The **delay until** must be supported on types `Calendar.Time` and `System.Real_Time.Time`, and it can be legally extended to apply to additional time types.

The CARTS interface supports timeouts on **select** statements by including an optional timeout parameter of type `Duration` on the corresponding RTS calls. Extending the CARTS model in a straightforward way would mean either adding more parameters (one for each time type) to each RTS interface procedure that has a timeout, or adding versions (one for each time type) of all the RTS interface procedures with timeout. This would be very cumbersome. Worse, the RTS interface specification becomes dependent on the packages defining the time types, which tends to lead to circularity, because the implementation of timer services needs other RTS services. The problem could be deferred to the bodies of the RTS packages, by defeating typing of the timeout parameter at the interface, but then the bodies of the RTS procedures would still depend on the packages providing the time services, so there would remain elaboration order difficulties.

In principle, the timeout parameter could be a classwide type, and the actual timeout values could be extensions of this type, with components of the appropriate implementation type. This solution was considered, but rejected, mainly because of concern that reliance on dispatching operations would prevent parallel development of the RTS and the implementation of tagged types in the compiler.

The present design, explained in Section 7, avoids this problem, by removing all timeout parameters from the core GNARLI interface. The compiler translates both forms of **delay** into protected entry calls. All forms of **select** with **delay** are translated into structures using the asynchronous **select** statement. The addition of new time types is simple, requiring only that the compiler be told the location of the time type. The appropriate delay object and relational operator, needed by the compiler to generate **delay** statements, can be located automatically.

This design provided early tests for ATC and protected entries. The ACVC tasking tests for Ada 83 use delays extensively, both in isolation and in combination with **select** statements. As translated according to the GNARLI model, these tests all exercise the implementation of ATC and protected entry calls. Before GNAT was able to generate RTS calls for these constructs, we were able to run the ACVC tests using hand-translated glue-code and the Verdix[3] Ada compiler.

Of course, the use of ATC and protected entries to implement delays incurs a risk of high overhead. We are aware of this, but felt taking that risk is consistent with the research nature of the GNARL project. The leverage works in both directions, providing strong motivation to work for a correct and efficient implementation of ATC and protected entries — a situation analogous to the effect of improved optimizations on a compiler that is used to bootstrap itself.

[3]Verdix is a trademark of the Verdix Corporation.

4 Interface to Lower-Level Services

The original PART Ada RTS, for Ada 83, was constructed as a layer over Pthreads. The intention was to provide interoperability with C-language libraries that might use Pthreads, limit blocking on OS services to a single task, and make the RTS trivially portable across operating systems that comply with the POSIX standards. For GNARL, a higher level of portability is desired, to operating systems that do not support Pthreads, and even to bare machines. To this end, a new lower-level interface is introduced.

The GNU Low-Level Interface (GNULLI) hides the implementation of services that GNARL will need to request from the host operating system, if it is implemented over an operating system, and any dependences on the host machine. Services provided by the GNULLI interface are not Ada-specific. Because GNULLI will need to be modified when GNARL is ported to a new platform, it must be small enough to be implementable without a great deal of effort on most operating systems, or on a bare machine.

The GNULLI interface is an abstraction of a subset of the POSIX interfaces, including Pthreads. Therefore, it is trivially implementable over an operating system that supports the POSIX standards. In order to permit a simpler and more efficient implementation over other operating systems, or a bare machine, many features of Pthreads have been left out or restricted. The deleted features are ones that the Ada RTS does not need, or cannot use. For example, the POSIX semantics of thread cancellation do not fit the Ada semantics of abortion, so the Pthread cancellation services are not included in GNULLI. The features retained include thread creation and operations on mutexes, condition variables, and signals. In cases where the Ada RTS does not need the full strength or generality of the Pthread interface, the semantics are relaxed. For example, GNULLI mutexes have only one form of priority inheritance (the priority ceiling emulation locking protocol) and are required to be unlocked in FIFO order. Condition variables are only allowed to be used by one task at a time. Further simplifications are contemplated, such as the hiding of condition variables behind a suspend/resume interface.

We expect that certain features of GNULLI may not be supported by some commercial operating systems. One of these is per-thread asynchronous signals. Another is preemptive priority scheduling, and priority ceiling emulation. The interface is intended to allow the GNARL to operate, with reduced functionality, in such environments. If the OS does not support per-thread signals, the GNULLI can provide null bodies for the per-thread signal operation. GNARL will then have to postpone abortion until synchronization points. If the OS supports dynamic priority scheduling, but not priority ceiling locks, the GNULLI layer can itself do the priority boost/unboost for the mutex lock/unlock operations. If the OS does not support preemptive priority scheduling priority scheduling at all, the GNULLI layer can ignore priorities, and GNARL will not support the Real-Time Systems annex scheduling model.

The current GNULLI implementation is based on Pthreads, using pragma Interface and a C-language library implementation of Draft 6 of the Pthreads interface developed by the PART project over the SunOS 4.1 version of UNIX[4][9]. It has also been ported to the Silicon Graphics [5] implementation of Pthreads.

5 Protected Objects

Protected objects are a new intertask synchronization mechanism of Ada 9X. They provide synchronization based on a data object rather than between tasks. A protected object can contain data, subprograms, and entries. The data is private to the protected object, and can only be accessed by the subprograms and entries (the *protected operations*) of that object. Only a single task can execute a protected procedure of a given object at one time. Protected entries are similar to protected procedures with the addition of a *barrier expression*; a protected entry call can only proceed when the barrier is `True`. Multiple tasks can execute the protected functions of an object at one time, but they have only read access to the data of the object; no protected procedures or entries can execute while the functions are executing.

Protected entries present the most complex problem in the implementation of protected objects. When an entry call is made, its barrier is evaluated. If the barrier is `False`, the entry call cannot continue; the caller must wait for the barrier to become `True`. Since the intent is that the barrier will depend on data private to the protected object that can only be changed by some protected operation, Ada only requires that this be checked after each protected procedure or entry call. This will usually be done by a task other than that waiting on the barrier.

When the barrier expression is `True`, the corresponding entry body is executed. In principle, protected entry bodies can be executed by any task, regardless of which task made the corresponding call. The two most reasonable possibilities are the task that made the call (the *self-service model*), and the task that opened the barrier that allows the call to proceed (the *proxy model*). The self-service model allows the task that opened the barrier to continue with its own processing, whereas as a proxy it would experience a potentially unbounded delay as it serviced all the calls waiting on open barriers. This might be important on systems with large numbers of processors. However, this requires that the mechanism which guarantees mutual exclusion within a protected object be transferable, that is, the task opening the barrier must be able to transfer the lock to the calling task without allowing any other task to lock the object in the meantime. This is necessary because waiting entry calls have precedence over other protected operations (including new protected entry calls). Ada does not permit a new protected operation to proceed on an object while entry calls are waiting on open barriers.

When operating system support for multithreaded programming is used to implement Ada operations, the ability to transfer a lock in this way will not always be available. For example, Pthreads provides mutual exclusion through *mutexes*, data objects with lock and unlock operations defined on them. Once a thread locks a mutex it owns it, and must unlock it to relinquish ownership; no other thread can perform the unlock. Transferring ownership of the lock would effectively require the implementation of a new locking mechanism over that provided by Pthreads, which would impose considerable overhead. For this reason, the GNARL implementation uses the proxy model of entry body execution. This is also the model described by the Ada 9X Mapping/Revision Team's Ada 9X implementation modules[10].

Originally, an attempt was made to make the GNARLI neutral with respect to these two execution models, by allowing either the caller or the proxy to execute the entry body at the discretion of the RTS. However, with inline code generation, this required a test in both the proxy and the caller to determine whether the entry body needed to be executed. This complicated code generation, and added distributed overhead, for both implementations. Therefore, GNARLI made a commitment to the proxy model.

The GNARLI commitment to implement tasking constructs by inline generation of GNARL subprogram calls has a profound effect on the interface for protected objects, in particular for protected entry calls. Whenever any task completes a protected procedure or entry call, it must check the barriers on any entry for which other tasks are waiting and, if any evaluate to `True`, execute the corresponding body. If callbacks are used, this can be implemented in a relatively straightforward manner. The compiler translates all barrier expressions into Boolean functions and all entry bodies into procedures. Pointers to all of these subprograms are incorporated into the runtime structure used to control the protected object. An RTS operation can then be

[4] UNIX is a trademark of UNIX Systems Laboratories, Inc.
[5] Silicon Graphics is a trademark of Silicon Graphics, Inc.

defined that evaluates the barrier function corresponding to each non-empty entry queue and, if the function returns `True`, dequeues an entry call and executes the corresponding entry body procedure. We cannot do this, since we have ruled out callbacks.

Our inline interface requires that both barrier evaluation and entry body execution be executed by the application code directly, using GNARL calls to adjust task state and obtain the information needed to make local control-flow decisions. This requires the compiler to perform a somewhat more complex transformation of protected object declarations.

The basic form of a protected object implemented using GNARLI calls is an ordinary procedure or function corresponding to each protected procedure or function for the object, and a single procedure containing the code to all of the object's entry barriers and bodies.

Consider the following schematic protected object.

```
protected type PT is
  procedure P;
  function F return T;
  entry E1;
  entry E2;
  ...
private
  ...
end PT;
protected body PT is
  procedure P(...) is
  begin
     ...sequence of statements...
  end P;
  function F(...) return T is
  begin
     ...sequence of statements...
    return  expression;
  end F;
  entry E1 when  barrier expression is
  begin
     ...sequence of statements...
  end E1;
  entry E2 when  barrier expression is
  begin
     ...sequence of statements...
  end E2;
end PT;
PO : PT;
```

The following is a schematic representation of the code that might be generated by a compiler to implement such a protected object using the GNARLI.

```
type PT is record
  Object : Protection(Num_Entries =>  N);
   ...additional object state...
  end record;
procedure P(PO : in out PT;
     ...additional parameters...) is
  Dummy : Boolean;
begin
  Defer_Abortion;
  Lock(PO.Object);
   ...sequence of statements...
  Service_Entries(PO, Dummy);
  Unlock(PO.Object);
  Undefer_Abortion;
end P;
function F(PO : PT;
     ...additional parameters...) is
  Temp : T;
begin
  Defer_Abortion;
  Lock_Read_Only(PO.Object);
   ...sequence of statements...
  Temp :=  expression;
  Unlock(PO.Object);
  Undefer_Abortion;
  return Temp;
end P;
procedure Service_Entries(
    PO : in out PT;
    Pending_Serviced : out Boolean) is
  P : System.Address;
  subtype PO_Entry_Index is Protected_Entry_Index
     range 0..2;
  Barriers : Barrier_Vector(1..2);
  E : PO_Entry_Index;
  PS : Boolean;
  Cumulative_PS : Boolean := False;
begin
  loop
    Barriers(1) :=  ...barrier expression...;
    Barriers(2) :=  ...barrier expression...;
    Next_Entry_Call(PO.Object, Barriers, P, E);
    case E is
    when Null_Protected_Entry =>
      exit;
    when 1 =>
       ...sequence of statements...
      Complete_Entry_Body(PS);
    when 2 =>
       ...sequence of statements...
      Complete_Entry_Body(PS);
    end case;
    Cumulative_PS := Cumulative_PS or PS;
  end loop;
  Pending_Serviced := Cumulative_PS;
end Service_Entries;
PO : PT;
```

This translation leaves out a number of important details (such as provision for abortion, exception handling, and finalization), but it does show points where calls to the GNARL are needed to inform it of a change in task state and to return needed information on that state to the application code. The GNARL's portion of the protected object's state is encapsulated in an object of the private type `Protection`. This type is discriminated by the total number of distinct entries of the protected object, with each member of an entry family counting as a separate entry. This allows the `Protection` object to contain structures sized by the number of entries (e.g. an array of entry queues). A unique entry index between 1 and this limit is assigned to each entry by the compiler. Note that `Protection` objects are allocated by the compiler; this is an exception to the usual GNARLI rule that runtime structures are allocated internally. This makes the allocation of protected objects similar to that of other data objects, avoiding the overhead and memory management problems associated with allocation by the GNARL. It does make the runtime more vulnerable to corruption of compiler data structures, however.

The `Lock` operation is used to implement mutual exclusion for protected operations. A task which locks a `Protection` object owns that object, and there can be only one owner at a time. Other threads attempting to lock the object will be prevented from returning from the `Lock` operation until the owner calls `Unlock` on that object. For protected functions, there is also a `Lock_Read_Only` operation.

Both whole-task abortion and ATC must be deferred for a task while it executes a protected operation. Since there are other instances in which application code must be protected from abortion, the separate Defer_Abortion and Undefer_Abortion operations are provided. Since deferred operations can be nested, these operations are also nestable, and abortion is deferred in a task until it has executed as many Undefer_Abortion as Defer_Abortion operations.

The Service_Entries procedure implements all entry bodies for this object. It is used wherever an entry call might be waiting on a True barrier condition, whether the barrier is True when the call is made or it becomes true as the result of a later protected operation. The latter can happen whenever the state of a protected object changes, which can only be the result of a protected procedure or entry call. Whenever this happens, Ada dictates that the protected operation cannot terminate (and therefore no new protected operations can commence) so long as there are entries waiting on True barriers. Therefore, before unlocking the protected object after such an operation, Service_Entries must be called. An example of its use is seen in the implementation of the protected procedure P. Note that the implementation of the protected function F does not call Service_Entries, since it cannot change the state of the protected object and therefore should not change the value of any barriers.

The Next_Entry_Call GNARLI operation is used to control the execution of a Service_Entries procedure. The Service_Entries procedure evaluates all entry barriers and places the results in a Boolean array indexed by the corresponding entry indices. Next_Entry_Call uses the values of the barriers, internal entry queues, and the queuing discipline in effect to determine the next entry body that should be executed. It passes back the index of this entry, together with an address value provided at the time of the entry call (shown below) that can be used to implement entry parameters. This index can then be used by the application code to choose the sequence of statements associated with the corresponding entry for execution. In the example, this is done using a **case** statement.

When the body of an entry is complete, the application code notifies the GNARL using the Complete_Entry_Body call. This call has various effects depending on the type of entry call that was serviced. For example, if it is a simple or conditional entry call, it will wake up the calling task, which is waiting for the completion. Complete_Entry_Body returns True in the **out** parameter PS if the entry just serviced is the currently *pending* entry call, that is, it was just called and was able to be serviced immediately, without being put on a queue. This information is returned by Complete_Entry_Body rather than Next_Entry_Call because a pending entry call may become queued after service begins, by a **requeue** statement (not described here) in the entry body. The PS value returned by Complete_Entry_Body is used to update the Pending_Serviced **out** parameter of Service_Entries. Some of the processing of an entry call can be skipped if the call is serviced immediately, and Pending_Serviced allows this, as will be explained below.

The loop in Service_Entries repeatedly executes entry bodies until Next_Entry_Call does not find an entry eligible for execution. Next_Entry_Call then returns the distinguished Null_Protected_Entry, causing Service_Entries to return.

To translate a call to a protected procedure, the compiler generates a call to the corresponding translated procedure. The translation of a protected entry call is somewhat more complex, and requires the additional GNARLI operation Protected_Entry_Call. For example, the simple entry call PO.E1; might be translated as

```
declare
  Cancelled : Boolean;
  Pending_Serviced : Boolean;
begin
  Defer_Abortion;
  Lock(PO.Object);
  Protected_Entry_Call(
      PO.Object, 1, Null_Address, Simple_Call);
  Service_Entries(PO.Object, Pending_Serviced);
  Unlock(PO.Object);
  if not Pending_Serviced then
    Wait_For_Completion(Cancelled);
    if Cancelled then
       Service_Entries(PO.Object, PS);
       Unlock(PO.Object);
    end if;
  end if;
  Undefer_Abortion;
  Raise_Pending_Exception;
end;
```

The Protected_Entry_Call operation *pends* the entry call on the protected object. This is as distinguished from queuing the call, since a queued entry call would contend with all other entry calls already queued on the object, and Ada requires that an entry call that can be executed immediately take precedence over those that have been queued. If this call is eligible for service, that is, its barrier expression evaluates to True, the Next_Entry_Call operation in Service_Entry_Calls will return its index (1); otherwise, it will queue the call. The act of queuing the call is a protected operation, and may make other barriers become True, since barriers can depend on the 'Count attribute of entries.

As described above, Service_Entries returns a Pending_Serviced flag indicating whether the pended call was serviced immediately. This increases the average efficiency of the interface, since if the call has been serviced, there is no need to wait for its completion.

If the call is queued, Wait_For_Completion suspends the calling task until the entry call is complete, when it is awakened by the corresponding call to Complete_Entry_Body. Note that Wait_For_Completion is called with abortion deferred. The abort operation will cause Wait_For_Completion to return, after dequeuing the call. Since dequeuing the call is a protected operation, Service_Entries must be called after the call is dequeued. The application would be unable to do this if abortion caused an immediate transfer of control. The object must be locked to dequeue the call and to call `Service_Entries`. To avoid unnecessary locking, this is only done if `Wait_For_Completion` actually dequeues the call. This means that the object may or may not be locked on exit from `Wait_For_Completion`. This design would be error-prone if `Wait_For_Completion` were intended to be used directly by human programmers. However, the compiler can ensure that `Unlock` is called when and only when needed. The `Cancelled` **out** parameter of `Wait_For_Completion` allows it to do this, skipping unnecessary post-processing unless the call was cancelled.

This implementation requires that all the processing of each RTS call complete, even when the operation detects an error that should raise an exception in the application. For this reason, any exceptions which occur in the runtime are caught and saved. They are raised later, after return from certain GNARLI operations and before executing further application code, by a compiler-generated call to the GNARLI operation `Raise_Pending_Exception`.

6 Asynchronous Transfer of Control

The syntax of the asynchronous **select** statement is

```
select
   triggering_statement [ triggered_statements ]
then abort
   abortable_part
end select;
```

The asynchronous **select** statement allows a task wait on an entry call or **delay** statement (the *triggering statement*) while executing a sequence of statments (the *abortable part*). While the triggering statement is being serviced (by an **accept** statement or entry body), it logically executes in parallel with the abortable part. Once the triggering statement has completed, the abortable part is aborted, and execution continues with the *triggered statements*, and thence to the statement after the **select** statement. This effect is called *Asynchronous Transfer of Control*, or ATC. Since GNARL is intented to support the Ada Real-Time annex, which requires abortion to occur immediately when the triggering statement completes, the RTS cannot simply perform the ATC at abort synchronization points.

Two ways of implementing ATC were considered. The *two-thread* model spawns a separate thread of control to execute the abortable part. This has the advantage that the abortable part can be aborted by cancelling this new thread, allowing the RTS to use forms of thread cancellation that do not permit recovery of the thread.

The *one-thread* model treats ATC as an actual transfer of control. When the trigger completes, control is transferred out of the abortable part to the sequence of statements after the trigger, and thence to the statements after the asynchronous **select** statement. This requires some means of asynchronously interrupting a task and redirecting its flow of control. For example, this can be done using the Pthreads primitives, by sending it a signal (a form of software interrupt), and performing a non-local jump out of the signal handler to the new continuation point.

The two-thread model requires the creation and destruction of a thread of control for each execution of an asynchronous **select** statement, and requires that each task keep track of multiple threads of control, resulting in more complex per-task data structures. For this reason, the GNARL implementation uses the one-thread model. The non-local jump is performed by raising a special Ada exception in a signal handler. The propagation of this exception aborts one or more levels of abortable parts, or the entire task.

Again, an attempt was originally made to design the GNARLI so that it could be implemented with either model. However, this was difficult to reconcile with the use of exceptions to transfer control. Further, the commitment to the proxy model negates most of the advantages of the two-thread model. The GNARLI is therefore is committed to the one-thread model of ATC.

The effect of transferring control out of an asynchronously invoked handler to some point in a task, by any mechanism, is likely to be unpredictable, unless one knows something about what was going on in the task when the handler was invoked. For example, the C-language *longjmp()* operation allows return from a signal handler to the point of a previously executed *setjmp()* operation, popping the activation records of any intervening subprogram calls from the runtime stack. However, *longjmp()* and many other POSIX operations, including all the Pthreads operations, are *async-signal unsafe*. The effect of executing an unsafe operation in an asynchronous signal handler that has itself interrupted an unsafe operation is "unspecified". For this reason, since our GNARL implementation calls (through our Pthreads-based GNULLI implementation) many async-signal unsafe operations, most of it must be protected against abortion.

Protection of the RTS against abortion can be accomplished using the same mechanism provided to the application by the GNARLI, i.e., the `Defer_Abortion` and `Undefer_Abortion` operations. The `Defer_Abortion` operation prevents abortion from taking place until the corresponding `Undefer_Abortion` operation is executed. Any requests for abortion that occurred in the meantime will be serviced by the `Undefer_Abortion` call corresponding to the first `Defer_Abortion` call by the task. Since these operations protected GNULLI calls, they must be implemented without using GNULLI[8].

For the compiler to translate the **abort** statement, GNARLI provides the `Abort_Tasks` call. However, local abortion (ATC), occurs as a side-effect of completion of an entry call or delay, and so requires no special GNARLI call. It is implemented internally by the GNARL.

For example, an asynchronous **select** statement with a protected entry call trigger is implemented using `Protected_Entry_Call` as for a simple entry call, but with a mode of `Asynchronous_Call`. This modifies the effect of the `Complete_Entry_Body`, so that it aborts the task to the appropriate ATC nesting level (that is, aborts the corresponding abortable part and all other asynchronous **select** statements nested within it) rather than just waking it up. This mechanism takes care of the case where the triggering statement starts before the abortable part completes. On the other hand, if the abortable part completes first, the triggering statement must be cancelled (dequeued). As described for `Wait_For_Completion` in the previous section, this is a protected operation, and requires entries to be serviced.

We now describe the translation of asynchronous protected entry calls. Since asynchronous task entry calls are similar (and somewhat simpler), we will not discuss them separately. Consider an asynchronous call of the protected object of the example in the previous section:

```
select
  PO.E1;
  ...triggered statements...
then abort
  ...abortable part...
end;
```

This might be translated as:

```
declare
  Cancelled : Boolean := False;
begin
  declare
    PS : Boolean := False;
  begin
    Defer_Abortion;
    Lock(PO.Object);
    Protected_Entry_Call(PO.Object,
        1, Null_Address, Asynchronous_Call);
    Service_Entries(PO, PS);
    Unlock(PO.Object);
    if not PS then
      Wait_Until_Abortable;
      Undefer_Abortion;
      Raise_Pending_Exception;
       ...abortable part...
    else
      Undefer_Abortion;
      Raise_Pending_Exception;
    end if;
  at end
    if not PS then
      Cancel_Protected_Entry_Call(Cancelled);
      if Cancelled then
        Service_Entries(PO, PS);
        Unlock(PO.Object);
      end if;
    end if;
  end;
  Raise_Pending_Exception;
  if not Cancelled then
     ...triggered statements...
  end if;
end;
```

The code at the start of the entry call is the same as for the simple entry call described in the previous section. If this call is serviced immediately, the result is also the same; no further processing is needed, except that in this case the triggered statements are executed.

The abortable part cannot run until the call is queued abortably. If the call is queued non-abortable (as is the case if the call is pending and the entry body executes a **requeue** statement without abort), `Wait_Until_Abortable` suspends the task until the call is queued abortably or completes. This is done with abortion still deferred, since the task should not be abortable when waiting on a non-abortable call. Once the call is queued abortably the abortable part is executed.

Unlike the simple entry call, where abortion is deferred until the call is complete, abortion must be undeferred while executing the abortable part. This poses a problem for describing the asynchronous call in terms of an Ada source-to-source transformation. If the abortable part raises an exception, the call must still be cancelled before the exception can propagate from the **select** statement. However, if an ordinary exception handler were used to do this, there would be a window between the time that the handler started executing and the time that it called `Defer_Abortion`. During this window the task could be aborted (by task abortion, or by an outer asynchronous `select` statement). This would confuse the GNARL implementation, which would assume that the call was still in progress, since it has no way of associating the internal representation of a call with a particular asynchronous **select** statement other than the assumption that the current call matches the currently executing construct.

The compiler must provide some means to close this window. There does not appear to be any way of expressing this using standard primitive Ada source constructs, so in the above translation, we have represented this by an "**at end**" handler. This is a syntactic convention for the mechanism GNAT uses internally for the translation of constructs requiring finalization. The sequence of statements of the **at end** handler is always executed, with abortion deferred, before leaving the associated block. It catches all all exceptions, including abortion. If it is entered via propagation of an exception, the exception is re-raised when the handler is complete.

If the abortable part of the asynchronous `select` statement completes, `Cancel_Protected_Entry_Call` is called to attempt to dequeue the entry call which serves as trigger. If the call is already in progress it cannot be cancelled, so `Cancel_Protected_Entry_Call` waits for the call to complete. Only if the call has not yet started will it be dequeued. As with `Wait_For_Completion`, the `Cancelled` flag of `Cancel_Protected_Entry_Call` allows the application code to avoid servicing entry calls if the call has already been dequeued.

7 Timer Service

Timer service provides support for the various forms of the **delay** statement, including delay clauses in **select** statements. It forms a separate interface layer within the GNARL, between the application code and other GNARLI operations.

This layering is made possible through the use of protected types and ATC. For `Duration` and each type representing absolute time, the Timer Service layer defines a protected object with a `Wait` entry. This entry takes an argument of the corresponding time type and is serviced when the requested time has arrived. Various forms of simple and asynchronous calls on this entry can then be used, in conjunction with core GNARLI services, to provide all of the timed constructs of Ada.

For example, a simple delay statement `delay 1.0;` is implemented as a simple entry call on the appropriate delay object `Calendar.Delays.Delay_For(1.0);`. An asynchronous **select** statement with a delay trigger is simply converted to an asynchronous entry call on the appropriate delay object.

The delay statement can be combined with other Ada features to implement timed entry calls. For example, the following statement:

```
select
  PO.E1(1);
   ...entry statements...
or delay 10.0;
   ...delay statements...
end select;
```

might be translated as:

```
declare
  use System.Real_Time;
  Timed_Out : Boolean := True;
  Deadline : Time := Clock + To_Time_Span(10.0);
begin
  select
    PO.E1(1);
    Timed_Out := False;
  then abort
    delay until Deadline;
  end select;
  if Timed_Out then
     ...delay statements...
  else
     ...entry statements...
  end if;
end;
```

The timed entry call is implemented as an asynchronous **select** statement in which the entry call is the triggering statement and the abortable part consists of a delay statement, which will be implemented as a simple call on the appropriate `Wait` entry. If the triggering statement is accepted before the `Wait` call, the latter is aborted, effectively cancelling the delay. Conversely, if the `Wait` call is accepted first, the abortable part will be left, cancelling the trigger. The `Timed_Out` flag is used to record whether the entry call was serviced or not, so that the statements following the entry call or the delay statement can be executed as appropriate after the asynchronous `select` statement.

Implementing a selective wait with delay alternatives is more complex, since it cannot be used as a trigger in an asynchronous **select** statement. The selective wait must be the abortable part of an asynchronous `select` statement with a delay trigger, so that it can be interrupted by the timeout. However, a sufficiently short timeout can result in the selective wait being interrupted before it can accept an entry call, even if there are calls waiting. This requires a conditional selective wait to be used to check for this case before the timeout is started.

We now explain the implementation of the `System.Real_Time.Delay_Object`, defined for `Time_Span` in the Ada 9X Real-Time Annex; implementation of delay objects for the time types is similar, and can use the same or different underlying time services. `Delay_Object` is defined in the predefined package `System.Real_Time`, a partial implementation of which is sketched below.

```
with RTS_Time;
package System.Real_Time is
  type Time is private;
  ...
  type Time_Span is private;
  protected Delay_Object is
    entry Wait(TS : Time_Span);
  private
    ...
  end Delay_Object;
  ...
private
  type Time is new RTS_Time.Time;
  type Time_Span is new RTS_Time.Time;
  ...
end System.Real_Time;
```

`RTS_Time.Time` is a low-level time type, used (for simplicity) to implement both `Time_Span` and and `Time` (as a duration from an epoch).

The `Delay_Object.Wait` operation converts its `Time_-Span` argument to an absolute time. It then requeues itself on an entry in another object that waits until this absolute time arrives.

```
with RTS_Time_Services;
package body System.Real_Time is
  ...
  protected body Delay_Object is
    entry Wait(FD : Time_Span) when True is
    begin
      requeue RTS_Time_Services.Timer.Enqueue
        (RTS_Times.Time(Clock+FD)) with abort;
    end Wait;
  end Delay_Object;
  ...
end System.Real_Time;
```

The `Timer.Enqueue` entry is part of a protected object that queues and services timeout requests. This requires that the entry calls queued waiting for timeouts be available in chronological order. Since Ada protected entry calls are not queued in this order, an explicit queue must be maintained by the `RTS_Time_-Services` package. Each element of this queue contains a protected object of type `QEL_Type` (full declaration given further below) on which the task waiting for the corresponding timeout waits. Only one task waits on each of these protected objects, so it can be awakened when the timeout arrives.

The `Enqueue` operation needs to be a protected entry call because queuing the timeout and servicing it must be atomic operations. The `Service` routine, which wakes up a task on the timeout queue is part of the same protected object, so it and the Enqueue operation execute in mutual exclusion. The Enqueue operation completes by requeuing its call to the protected object in the queue element, without abort, so the Service routine can rely that there is a task queued on each element in the queue.

To do the actual timing, a timer service task Timer_-Server is defined. This uses the GNULLI Cond_-Timedwait operation to wait for the next timeout, which is the one at the head of the timer queue. When this time arrives, the timer server calls the protected procedure Service to service all the entries that are due to time out. Service consists of removing the queue element, waking up the task by calling a protected procedure of the queue element's QEL component, and then disposing of the queue element. The Cond_Timedwait operation can also be awakened by another task using the Cond_Signal operation. The Timer.Enqueue operation does this to force Timer_Service to reset the next timeout to an earlier time.

```
with RTS_Time; use RTS_Time;
with GNULLI;
with RTS_QEL; use RTS_QEL;
package body RTS_Timer_Services is
  ...
  type Q_Rec;
  type Q_Link is access Q_Rec;
  type Q_Rec is record
    QEL :  QEL_Type;
    T :    Time; -- wake-up time
    Next : Q_Link;
    end record;
  Timer_Condition : GNULLI.Condition;
  ...
  task Timer_Server;
  protected Timer is
    entry Enqueue(T : in Time);
    procedure Service(T : out Time);
  private
    Q_Head, Avail : Q_Link;
  end Timer;
  protected body Timer is
    entry Enqueue(T : in Time) when True is
      N : Q_Link;
    begin
      N :=  ...allocate a new Q_Rec  ...;
      N.T := T;
       ...insert N  into time-ordered queue...;
      if Q_Head  changed then
        GNULLI.Cond_Signal(Timer_Condition);
      end if;
      requeue N.Wait;
    end Enqueue;
    procedure Service(T : out Time) is
    begin
      while Q_Head/=null and then
          Q_Head.T<=Clock loop
         Q_Head.QEL.Signal;
          ...remove node from Q_Head...
      end loop;
      if Q_Head=null then T := Time_Last;
      else T := Q_Head.T;
      end if;
    end Service;
  end Timer;
  task body Timer_Server is
    Next_Wakeup_Time : Time := Time_Last;
  begin
    loop
      GNULLI.Cond_Timedwait(Timer_Condition,
        Next_Wakeup_Time);
      Timer.Service(Next_Wakeup_Time);
    end loop;
  end Timer_Server;
  ...
end RTS_Timer_Services;
```

The QEL_Type protected type defines a simple protected object containing an entry Wait on which a task can wait, and a protected procedure Signal which wakes up the waiting task.

```
with RTS_Time;
package RTS_QEL is
  protected type QEL_Type is
    entry Wait;
    procedure Signal;
  private
    Closed : Boolean := True;
  end QEL_Type;
end RTS_QEL;

with RTS_Time;
package body RTS_QEL is
  protected body QEL_Type is
    entry Wait when not Closed is
      null;
    end Wait;
    procedure Signal is
       Closed := False;
    end Signal;
  end QEL_Type;
end RTS_QEL;
```

The Wait entry has no body, and only one task needs to be able to wait on it at a time. Therefore, it would be a good candidate for implementation as a special case using low-level operations. This may prove necessary to get adequate performance. If so, we hope to do it via an optimization of degenerate protected types that will also be accessible to ordinary applications.

8 Status

GNARL has been implemented, including support for all Ada 9X features. Though the GNARL is primarily intended to support the GNAT compiler, at the time of this writing (May 1994) the testing under GNAT has been limited, since GNAT is still under development and is not yet able to translate all Ada 9X tasking constructs into GNARLI calls. Some preliminary testing of GNARL has been done, using the Verdix Ada compiler and glue code that implements the RTS calls generated by the Verdix compiler using GNARLI calls. Operating in this mode, the GNARL was able to pass the Chapter 9 portion of ACVC 1.11. This has tested some Ada 9X tasking features, since the glue code for rendezvous with timeout is implemented using delay protected objects and ATC. Testing of some Ada 83 features of GNARL has also begun using the GNAT compiler.

9 Conclusions and Future Work

Ada 9X is an emerging standard that adds considerable functionality to the Ada tasking model. This includes lightweight synchronization in the form of protected types and lightweight cancellation in the form of ATC. How lightweight these constructs will actually be is the subject of speculation. By providing an early implementation, the GNARL may help to resolve this issue. There is also some uncertainty over the degree to which support for asynchronous task abortion imposes distributed overhead on other tasking features.

We plan to devise some tests to measure the performance of Ada 9X features, in absolute terms. We can also compare the GNARL's performance against our own Ada 83 RTS for the Verdix compiler, which may give some indication of the relative amount of overhead between the two forms of Ada tasking. However, we do not expect the results of such tests to necessarily be conclusive, since we know we have resolved many design decisions in favor of portability over performance.

The proposed POSIX threads interface is also an emerging standard, offering the possibility of a widely available lightweight process model on which Ada tasking can be based. This offers a potential for increased flexibility to the applications programmer and for simplified runtime implementation to the compiler developer. One objective of this research is to demonstrate the practicality of implementing Ada 9X tasking using Pthreads. By developing the GNARL, a complex application of Pthreads, this research should help verify the degree to which the current draft of Pthreads fulfills this potential, as well as to what degree it is suitable for the implementation of Ada.

So far, there seem to be no insurmountable problems, except that our usage of *longjmp()* for ATC may not be strictly compliant with the proposed Pthreads standard. However, some aspects of Pthreads, have made the implementation difficult and seem likely to hurt performance. One example is the need to provide abortion protection around all threads operations, because of async-unsafety. Another example is that Pthreads condition variables allow multiple tasks to be queued waiting for a condition, but the queuing disciplines provided are not adequate to support Ada entry queuing semantics. Any distributed overhead in the Pthreads implementation due to such features not used by the Ada RTS will impact the performance of GNARL implemented over Pthreads, relative to an implementation designed just to support Ada. We hope to be able to produce a non-Pthreads implementation of GNULLI, so as to be able to estimate the performance penalty for using Pthreads.

Once GNAT is complete, and all the GNARL functionality has been well tested, we hope to be able to tune the interface and the implementation for performance. Besides experimentation with a non-Pthreads based lower level, we plan to look for ways to enable better in-line optimization of RTS calls.

References

[1] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual: Draft Version 4.0*, September 1993. Available by anonymous FTP from ajpo.sei.cmu.edu.

[2] Ted Baker. Requirements specification for the Common Ada Run-Time System. Prepared for Westinghouse Electric Corporation, February 1991.

[3] Ada Runtime Environment Working Group: MRTSI Task Force. A model runtime system interface for Ada. Technical report, ACM SIGAda, 1990.

[4] E.W. Giering and T.P. Baker. POSIX/Ada realtime bindings: Description of work in progress. In *Proceedings of the Ninth Annual Washington Ada Symposium*. ACM, July 1992.

[5] E.W. Giering and T.P. Baker. Using POSIX threads to implement Ada tasking: Description of work in progress. In *TRI-Ada '92 Proceedings*, pages 518–529. ACM, November 1992.

[6] E.W. Giering, Frank Mueller, and T.P. Baker. Implementing Ada 9x features using POSIX threads: Design issues. In *TRI-Ada '93 Proceedings*, pages 214–228. ACM, September 1993.

[7] IEEE. *Threads Extension for Portable Operating Systems (Draft 6)*, February 1992. P1003.4a/D6.

[8] E.W. Giering III and T.P. Baker. Ada 9X asynchronous transfer of control: Applications and implementation. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994. To appear.

[9] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[10] Ada 9X Mapping/Revision Team. Ada 9X implementation module 1: Protected records, operator visibility, child packages. Technical report, Intermetrics, Inc., January 1991. Available by anonymous FTP from ajpo.sei.cmu.edu.

[11] United States Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*, February 1983. ANSI/MIL-STD-1815A-1983.