IMPLEMENTING ADA PROTECTED OBJECTS—INTERFACE ISSUES AND OPTIMIZATION

E.W. Giering T.P. Baker[1]
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019
Internet: giering@cs.fsu.edu Phone: (904)644-3441

ABSTRACT

Ada 95 protected objects are intended to provide a synchronization mechanism that is lower-level and more efficient than rendezvous. However, in their full generality they can require a rather complex implementation. To realize the original goal of providing a low-cost synchronization mechanism, compilers should recognize simple cases and optimize them.

The interface between the code generated by the compiler and the runtime system affects the feasibility of such optimizations, as well as the quality of code in general. One fundamental choice that must be made in the design of this interface is between an "inline" model, where procedure calls only go from the application code into the runtime system, and a "callback" model, in which the runtime system also makes calls out to the application code, via procedure pointers. The comparative merits of these two models for the implementation of protected objects is sufficiently unclear that we have undertaken to experiment with implementations of both models in the Gnu/Ada Runtime Library (GNARL) and GNAT compiler.

This paper describes possible optimizations of the implementation of protected objects for some special cases. It also outlines an inline version and a callback version of a runtime system interface for protected objects, and discusses how the differences affect code generation and optimization. Finally, it compares the actual efficiency of these two interfaces using benchmark tests run on the two implementations.

[1]Partially funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through the U.S. Army CECOM.

1 INTRODUCTION

Ada requires a more complex runtime environment than most other languages; in particular, it requires support for multiple tasks. The GNu Ada Runtime Library (GNARL) is being developed at Florida State University to provide tasking support for the Gnu NYU Ada Translator (GNAT), an Ada 95 [3] compiler and associated runtime system (RTS) being developed separately at New York University [5]. This separate development has been facilitated by a well-defined procedural interface between code generated by the compiler for tasking constructs and the GNARL, described in the GNARL interface document [1].

GNAT adds to the suite of languages supported by the the Gnu C Compiler (`gcc`), which already includes compilers for C, C++, and Objective C. Like the other tools, GNAT and its associated RTS, including the GNARL, are provided in source without charge, and can be freely copied under the terms of the Gnu software license agreement.

The GNAT strategy for implementing tasking is based on tree-to-tree translation, converting tasking constructs in the intermediate syntax tree representation into equivalent sequential Ada language constructs, with interspersed RTS service calls. Thus, the GNAT interface should be expressible in terms of ordinary Ada packages whenever possible. In what follows, the use of an RTS interface is described in terms of transformation of tasking constructs into non-tasking Ada code, using RTS types and subprograms.

Most of the tasking features have now been implemented, and the focus of this work has changed to debugging and improvements in efficiency. One tasking feature for which efficiency is particularly important is the protected object. Protected objects contain data and protected operations on that data. All protected operations operate on the data of the protected object in mutual exclusion, providing a critical section without the separate thread of control required for ren-

dezvous. The mutually exclusive portion of this execution is called a *protected action*. Protected entries also provide for the queuing of an operation by providing a *barrier*, a Boolean expression that must be true before a protected entry call can proceed. Note that queuing the operation may or may not block the calling task, since it may be part of an asynchronous select statement.

To provide for maximum portability, the GNARL has been implemented in terms of the C-language binding to the POSIX threads extension (Pthreads) [4]. The basic means of providing mutual exclusion in Pthreads is the *mutex* object, which is owned by at most one thread of control, that which most recently locked it. The owner must unlock the mutex before any other thread can lock it. A mutex is associated with each protected object; this mutex is locked before attempting any protected operation on that object.

Protected subprograms have proven relatively straightforward to implement in this way; the compiler simply transforms the protected subprogram so that it locks the mutex at the beginning of the subprogram and unlocks it before returning. This procedure can then be called like a normal procedure[2].

Protected entry calls are more complex. After locking the mutex, the barrier for the call must be checked. If it is `True`, the associated sequence of statements is executed; otherwise, the call is queued. In either case, there may be queued entries on the same object whose barriers become `True`, either because executing the entry call changed the object's components, or because queuing the call changed the `'Count` attribute of the associated entry. The barriers of all queued entry calls must now be reevaluated and, if `True`, the waiting calls executed. This must be done atomically; no other operations on the object can be allowed until all entry calls are either executed or waiting on `False` barriers. This can potentially mean evaluating each barrier multiple times, since each entry call can further alter barrier values. Protected procedures can also affect barrier values (protected functions cannot), and so they must also perform entry service before unlocking the mutex.

This adds considerable overhead to protected entry calls. Our investigations of potential optimization of protected objects has therefore concentrated on protected entries, and in particular elimination of redundant entry barrier evaluation and checking.

There is some also question as to whether the interface design itself has impeded efficiency. One design choice in particular, inline code generation, has been particularly controversial. The original GNARL assumes that all application code will be executed inline, that is, no application code is called from the RTS. Instead, protected object constructs are compiled into code with interspersed RTS calls. These calls maintain a runtime state of the protected object and perform the required intratask operations. RTS calls are necessary at any point where the task might suspend, or where it might influence the execution of another task.

An alternative approach is the *callback* interface. In this interface, segments of code whose execution is influenced by the RTS are packaged into procedures. Pointers to these procedures are passed into the RTS, which can then call the procedures at the appropriate time and in the appropriate environment.

An interface may use both techniques, implementing some constructs inline and some using callbacks. In Ada 83, a callback interface makes sense for executing tasks and accept statements. Ada 95 adds to these the execution of protected subprograms, entry bodies, and entry barriers. Our inline implementation of entry bodies is quite complex, in particular in the code that must be generated for entry bodies and calls, and we have found no way to reduce this complexity within the inline model. For this reason, we have developed a version of the GNAT and GNARL around an interface that executes entry bodies and their barriers via callback from the RTS.

The choice of inline or callback interfaces also effects the difficulty of different optimizations of protected objects. For example, there is no need to evaluate an entry barrier if no call is currently waiting on it. However, the entry queues on which calls wait tend to be hidden within the RTS. The callback interface does barrier evaluation within the RTS, and so can skip it for entries on which there are no queued calls. The inline interface could not do this unless interfaces to access RTS information about call queuing are provided to the compiler.

The inline expansion model was chosen for the initial GNARL implementation of protected entries, because it most closely fits a vision of protected entries first expressed by the Ada 95 language designer. During the language design process, when concerns were expressed about the potential overhead of barrier expression evaluation, the response was that compilers would be able to use contextual information to avoid reevaluating barriers unnecessarily.

In the original GNARL design, the inline approach seemed quite tractable. However, the original design did not take into account all the subtleties of interactions with other language features, such as priority-order entry queuing, ceiling checks, dynamic priority changes, and requeue operations. As more and more of these were all taken into account, the complexity of the compiler generated code and the RTS interface grew, disturbingly. This led us to study whether it would be

[2] Note that protected operations must also have access to the data of a protected object; the subprograms and calls are modified to pass this object as an additional implicit parameter.

better to stay with the inline interface, or convert to a callback interface.

Further details are provided below. Section 2 discusses means of optimizing both general and restricted protected objects. Section 3 provides a more detailed description of GNARL inline and callback interfaces for protected objects. Section 4 compares the two interfaces in terms of inherent complexity and their effect on various optimizations, and presents some preliminary benchmarks. Section 5 provides a summary of the work so far.

2 OPTIMIZATIONS

The need to service the entries of a protected object whenever a protected action might have changed the state of the object adds execution overhead to protected procedures as well as entries. Our approach is to call to some kind of entry service routine on completion of a protected procedure call, immediately before unlocking the associated mutex. Note that this must be done before leaving the subprogram for any reason, including exception propagation. One simple optimization is the elimination of this call in objects which contain no entries, which provides a simple and efficient critical section. This very straightforward optimization has always been provided by the GNAT implementation of protected objects.

For protected objects which have entries, many optimizations hinge on the exact nature of the barriers. In general, barriers must be reevaluated after any protected action, including queuing an entry call. However, if it can be determined that a barrier does not depend on the `'Count` attribute of an entry, it need not be reevaluated when an entry is queued—only when an entry or protected procedure call is executed.

The GNARL has complete responsibility for managing entry queues, so it needs access to the barriers to determine if a call should be queued or not. One way to do this is to place the values of all barriers in a *barrier vector*, a Boolean array with an element corresponding to each entry. Part of the interface between the compiler and RTS is the mapping of entries into elements of this vector.

In many cases, a barrier is a simple Boolean component. For example, consider the following protected type, used for simple suspension and resumption of a task:

```
protected type Signal_Object is
   entry Wait;
   procedure Signal;
private
   Open   : Boolean := False;
end Signal_Object;

protected body Signal_Object is

   entry Wait when Open is
   begin
      Open := False;
   end Wait;

   procedure Signal is
   begin
      Open := True;
   end Signal;

end Signal_Object;
```

A task can suspend itself by calling the `Wait` entry of an object of this type; another task can wake it up by calling Signal. Here the barrier is the Boolean component `Open`. If a barrier vector is permanently associated with the protected object, this kind of barrier can be simply mapped over the corresponding vector location. All accesses to the Boolean component would be translated into access to the vector component, assuring that the barrier component would always be current. In such cases, there is no separate barrier evaluation step.

If a vector of barrier values is kept in the protected object, it may be possible to avoid evaluating some barriers between protected actions. When a protected action is completed, all of the barriers on which entries are queued are valid, and are false. It may be advantageous to evaluate the barriers that are not awaited at this point as well. All of the barriers would then be valid at the next entry call, meaning that the initial barrier evaluation could be skipped. In the case where the barrier did not depend on a `'Count` attribute, a call could be queued without evaluating barriers at all. Of course, this is traded off against the need to evaluate barriers that are not being awaited. However, as pointed out above in Section 1, an inline interface may not have access to the entry queues at the point of barrier evaluation, so that all the barriers must be evaluated. In many cases, barriers may be simple enough that checking them is less expensive than checking the corresponding queues, which also argues for evaluating all barriers.

Depending on the exact arrangement the code that evaluates barriers, it may be possible for the compiler to remove it using static analysis. For example, if code evaluating the barriers follows entry body or procedure code immediately, such that there is no execution path from any other entry body or procedure to the barrier evaluation, the compiler may be able to eliminate evaluation of some or all of the entry barriers that were not affected by that particular protected operation.

3 IMPLEMENTATION

The overall strategy for implementing protected types is to define a record to represent the state of the object and a set of subprograms which implement the associated protected operations on that record. The record contains components representing the private data of the protected object and a component of type `Protection` representing the RTS state of the object. `Protection` is a type private to the GNARL, and contains the entry queues and the status of the current entry call on the corresponding object, if any. This record is passed as an **in out** parameter to the subprograms implementing protected procedures and entries, and as an **in** parameter to those implementing protected functions.

The subprograms which implement the protected entry bodies, barriers, and protected subprograms depend on the GNARL to enforce the required mutual exclusion and intertask synchronization. The exact division of labor between the compiler-generated code and the GNARL differs for the callback and inline interfaces. With the inline interface, the subprograms implementing protected operations call GNARL operations as required to provide the correct intertask environment before executing application code. With the callback interface, the RTS creates the correct environment and then calls the application code through subprogram pointers passed to it by the compiler-generated code.

3.1 THE INLINE INTERFACE

The GNARL inline interface transforms protected procedures and functions into procedures and functions, as described above, which have additional code to defer abortion and lock the associated mutex before entering the user-provided body and to unlock the mutex and undefer abortion before leaving. The interface contains the `Lock` and `Unlock` procedures, which act on an object of type `Protection`, and the `Defer_Abortion` and `Undefer_Abortion` procedures, for this purpose. Protected procedures may also change the state of the protected object and therefore of entry barriers, and so must check for queued entry calls that have become eligible for execution before unlocking. This is done using a call to an entry service procedure, described below. Protected subprogram calls are implemented as ordinary calls to the corresponding transformed subprograms.

Entry bodies are more difficult to translate inline. After each protected entry or procedure call, all barriers must be checked and any eligible entry calls executed. If the entry code is not to be called directly by the RTS, the iteration over entry barriers must be done by compiler-generated code. We have chosen to do this by generating an *entry service procedure* for each protected type. The entry service procedure contains the code for all entry bodies and barriers for that type. This procedure iterates over all entries, executing any that are waiting on an open barrier. It is called at the end of any protected action that might result in a change of state in the object.

As an example, we give the translation of the `Signal_Object` type defined above in Section 1 as performed by the the GNAT 2.04 compiler, simplified and edited for clarity.

The protected type is translated into a record type representing the object state.

```
type Signal_Objectv is record
   _object : aliased Protection (1);
   _service : aliased Service_Record;
   Open : Boolean := False;
end record;
```

This record contains a field for each private component of the protected object (here `Open`) to represent the object state explicitly defined by the application. The object also has an implicit runtime state (e.g. its entry queues). A component of the RTS interface type `Protection` (here `_object`) is used to hold the RTS state. `_object` is discriminated with the number of entries declared by the protected type, to allow for the sizing of RTS data structures.

The `_service` component contains information that allows the entry service procedure of this type to be called; it's initialization is not shown. This is required in certain circumstances where a protected operation is performed on an object as the result of an entry call, where the entry call might have been requeued. The object on which the call is actually queued cannot be known staticly, so the `_service` object is used to call it dynamically. This is arguably a portion of the interface that uses callbacks, though its use would not require RTS support. The `service` component does not really reflect the state of the protected object, and could be kept as a separate per-type data structure. It is kept in the per-object record as a convenience.

The `Signal` procedure is rewritten to take an **in out** parameter of the protected type record (a function would take an **in** parameter), and to lock the object and defer abortion during the execution of its sequence of statements.

```
procedure Signal (_object : in out Signal_Objectv) is
begin
   declare
      Pending_Serviced : Boolean;
   begin
      Defer_Abortion;
      Lock (_object._object'Access);
      _object.Open := True;
```

```
   at end
      Signal_ObjectS (
        _object'Address, Pending_Serviced);
      Unlock (_object._object'Access);
      Undefer_Abortion;
   end;
end Signal;
```

References to the protected component within the body are translated into references to the corresponding record component. The **at end** construct is an extension to Ada which exists only in the intermediate code; it indicates a code sequence to be executed on any exit from the associated block. Note that this cleanup processing includes a call to the entry service procedure `Signal_ObjectS`.

```
procedure Signal_ObjectS
  (O : Address;
   P : out Boolean) is
   ... local declarations ...
begin
   loop
       ... entry service loop ...
   end loop;
   P := Accumulate_Serviced;
end Signal_ObjectS;
```

The entry service procedure `Signal_ObjectS` takes an `Address` parameter to access the object being serviced. This allows indirect access to an arbitrary entry service procedure in the translation of protected entry calls, as described below. This address is converted into a locally declared access value referencing the object, as shown below. (The example uses "!" as an internal shorthand notation for unchecked conversion.)

```
type Signal_ObjectVP is access Signal_ObjectV;
_object : Signal_ObjectVP := Signal_ObjectVP!(O);
```

The compiler maps the entries of each protected type, in one-to-one correspondence with a range of integers starting with the value one[3]. It passes the upper bound of this range, which is the number of entries, to the RTS through the discriminant of the `_object` component. In the example we are using here, there is only one entry, with an entry index of 1. The entry service routine declares a local Boolean array, with a component for each entry, to hold the barrier values.

```
Barriers    : Barrier_Vector
  (1 .. _object._object.Num_Entries);
Entry_Index : Protected_Entry_Index
  range 0 .. _object._object.Num_Entries;
```

Some other local variables are needed by the entry service procedure for calling GNARL operations and for internal calculations.

[3] Each member of an entry family is considered to be a separate entry for this purpose.

```
Pending_Serviced    : Boolean := False;
Accumulate_Serviced : Boolean := False;
Parameters          : Address;
```

The entry service loop serves entry calls until no more are available for service. In our example, this loop has the following form.

```
loop
   Barriers (1) := _object.Open;
   Next_Entry_Call (
     _object._object'Access,
     Barriers,
     Parameters,
     Entry_Index);
   begin
      case Entry_Index is
         when Null_Protected_Entry =>
            exit;
         when Queued_Protected_Entry =>
            null;
         when 1 =>
            _object.Open := False;
         when others =>
            raise Program_Error;
      end case;
      Complete_Entry_Body (
        _object._object'Access,
        Pending_Serviced);
   exception
      when others =>
         Undefer_Abortion;
         Exceptional_Complete_Entry_Body (
           _object._object'Access,
           Pending_Serviced,
           Current_Exception);
   end;
   Accumulate_Serviced :=
     Accumulate_Serviced or Pending_Serviced;
end loop;
```

The body of the loop first evaluates all barriers, in this case simply copying the value of `Open` into the only component of the barrier vector. It then checks to see which call it should service next, by calling the RTS routine `Next_Entry_Call` with the `Protection` component and the barrier vector. `Next_Entry_Call` will return, via the parameter `Entry_Index`, the index of the next entry to be serviced according to the queuing discipline in effect. If there is no entry call ready to be served, it returns the distinguished value `Null_Protected_Entry`. `Next_Entry_Call` may also queue a call, as described below, in which case it returns the distinguished value `Queued_Protected_Entry`. Since barriers can in general depend on the `'Count` attribute of the entry on which the call was just queued, they must be reevaluated in this case. If the entry call has any parameters, a pointer to a block of storage containing the entry parameters is returned, via the parameter `Parameters`.

The entry index returned by a call to `Next_Entry_Call` is used to decide what to do next, using a case statement. For each entry index there is a corresponding case

alternative, which contains the declarations and statements of the entry's body. Following the entry body statements there is a call to `Complete_Entry_Body`, to inform the RTS that the call has completed execution. The effect of `Complete_Entry_Body` depends on the kind of entry call being completed. For a simple entry call, it will wake up the caller; for an asynchronous entry call, it will attempt to abort the corresponding abortable part. Once `Null_Protected_Entry` is returned, no further entry calls can become eligible to execute until the next protected action, so this case exits from the entry call processing loop. A return of `Queued_Protected_Entry` requires a reevaluation of the entry barriers, so the alternative for this value does nothing, allowing the loop body to execute again.

`Signal_ObjectS` returns a Boolean value indicating whether an entry call was serviced immediately, without being queued. This is used in conjunction with `Protected_Entry_Call` when initiating a new entry call, as explained below. The RTS keeps track of whether a call has ever been queued, and reports back when a call is completed in the `Pending_Serviced` parameter of `Complete_Entry_Body`. It must be reported by `Complete_Entry_Body` rather than by `Next_Entry_Call`, since the entry body may requeue the call to another object. It is not therefore known that a call will execute without being queued until it is complete. Since the loop iteration which executes the pending call is not distinguished from those executing queued calls, the `Accumulate_Serviced` flag is used to remember whether the first call to `Complete_Entry_Body` returned `True` for `Pending_Serviced`.

Calling a protected entry is somewhat more complicated than calling a protected subprogram. Consider a simple entry call to the `Wait` entry of `S`, an object of type `Signal_Object`.

```
S.Wait;
```

After deferring abortion and locking `S`, the RTS operation `Protected_Entry_Call` is called to inform the RTS that a new entry call is being attempted on `S`:

```
declare
   Pending_Serviced : Boolean;
   Block : Communication_Block;
begin
   Defer_Abortion;
   Lock (S._object'Access);
   Protected_Entry_Call (
     S._object'Access,
     1,
     Null_Address,
     Simple_Call,
     Block);
```

This does not actually execute the entry call, since the inline RTS has no access to the entry body code, nor does it queue the call, since a call that can be executed immediately should not be queued. Instead, it *pends* the call on the object. In addition to the `Protection` object and the index of the entry to be called, `Protected_Entry_Call` is called with a pointer to a record containing the actual entry parameters (here null), the constant `Simple_Call` indicating that this is a simple entry call (as opposed to a conditional or asynchronous call), and a `Communication_Block` object to tie this operation together with the other RTS operations making up this entry call. This `Communication_Block` object allows the RTS to pass values between related operations that might be expensive to recompute.

Once the call is pended, the entry service procedure `Signal_ObjectS` defined above is called to either execute or queue the pended call:

```
Signal_ObjectS (S, Pending_Serviced);
```

The call to `Next_Entry_Call` made by `Signal_ObjectS` gives preference to the pended call over queued calls, and will return its index if its barrier is open. If not, it will queue it and return `Queued_Entry_Call`. In either case, barrier values may change, and so `Signal_ObjectS` may execute a number of queued calls before returning.

After all affected entries have been serviced, the protected object is unlocked. At this point, it is necessary to wait for the call to complete unless it was serviced immediately. The `Wait_For_Completion` RTS call is used to do this, conditional on the value of the `Pending_Serviced` flag described above. The wait may be terminated by cancellation of the call, e.g. by abortion of the calling task. Since cancelling a call is a protected operation that may change barrier values, cancellation requires that the affected object be serviced. This cannot be done using `Signal_ObjectS`, since the call may have been requeued to another object or task. This is the function of the `Service_Cancelled_Call` operation. It calls the entry service procedure of the object through a pointer which is kept by the RTS. This is one form of callback which apparently cannot be avoided in implementing Ada 95 protected entries.

Both `Wait_For_Completion` and `Service_Protected_Entries` must be executed if the call has been queued, and must not be interrupted by exceptions or abortion. Abortion therefore remains deferred during these operations, though abortion will cause `Wait_For_Completion` to return:

```
Unlock (S._object'Access);
if not Pending_Serviced then
   Wait_For_Completion (Block);
   Service_Cancelled_Call (Block);
end if;
```

Any exceptions raised by these operations (e.g. an exception raised by the execution of the corresponding

entry body) are deferred until they are complete. If the task has been aborted, it will take effect when `Undefer_Abortion` is called; if any exceptions were deferred, they will be raised by `Raise_Pending_Exception`:

```
      Undefer_Abortion;
      Raise_Pending_Exception (Block);
   end;
```

3.2 THE CALLBACK INTERFACE

The complexity of the generated code for the inline interface, and in particular for entry calls, led to the reconsideration of a callback interface as a possible means of increasing the performance of protected operations within GNARL. In particular, the inline interface seems to require execution to move in and out of the RTS frequently, as evidenced by the numerous calls needed to implement a protected entry call (above). Other constructs, such as asynchronous calls, are even more complex. Intuitively, the callback interface should limit interaction between the compiler-generated code and the RTS for a protected entry call to one call to the RTS, plus a call back to evaluate the barrier and one to execute the entry body itself. Balanced against this is overhead due to the extra levels of indirection required for a callback.

As a design exercise, and to gain some empirical data on the relative performance of the two interfaces, we modified GNARL to implement a callback interface for protected entry calls, and modified GNAT to use this interface. Other interfaces remained unchanged. In particular, no callbacks were used for protected subprograms since we could not see any possible advantage to executing them through a callback.

With the callback interface, in order for the RTS to evaluate barriers and execute entry bodies, both barriers and entry body declarations and statements are packaged by the compiler into subprograms. Each barrier expression is transformed into a Boolean *barrier function*, and the balance of the entry body is transformed into an *entry action procedure*. Since the RTS must be able to call any arbitrary barrier function or action procedure, they must all have the same profile, using untyped parameters to access the associated protected object and parameters.

For example, consider the `Signal_Object` type which was translated using the inline interface in Section 3.1 above. To allow the barrier to be evaluated from within the RTS, the barrier expression is encapsulated in a function:

```
function Signal_Object__WaitB
  (O : Address;
   E : Protected_Entry_Index)
return Boolean
is
   type Signal_ObjectVP is access Signal_ObjectV;
   _object : Signal_ObjectVP := Signal_ObjectVP!(O);
begin
   return _object.Open;
end Signal_Object__WaitB;
```

To allow the sequence of statements within the entry body to be called from within the RTS, it is similarly encapsulated in a procedure:

```
procedure Signal_Object__WaitE
  (O : Address;
   P : Address;
   E : Protected_Entry_Index)
is
   type Signal_ObjectVP is access Signal_ObjectV;
   _object : Signal_ObjectVP := Signal_ObjectVP!(O);
begin
   _object.Open := False;
   Complete_Entry_Body (_object._object'Access);
exception
when others =>
   Exceptional_Complete_Entry_Body (
     _object._object'Access, Current_Exception);
end Signal_Object__WaitE;
```

The `Complete_Entry_Body` and `Exceptional_Complete_Entry_Body` RTS operations are similar to the inline equivalents described under the inline interface in Section 3.1 above.

Note that these subprograms take an `Address` parameter that points to the associated protected object record, which they convert to the access value `_object`. This is to allow all barrier function and protected entry procedures to have the same profile, allowing all such subprograms to be referenced by subprogram pointers of the same type. Pointers to these routines are loaded into an array:

```
Signal_ObjectA : aliased constant
  Protected_Entry_Body_Array (1 .. 1) :=
  ((Barrier => Signal_Object__WaitB'Access,
    Action => Signal_Object__WaitE'Access));
```

Note that this array is per-type, not per-object, so a pointer to the array is put in each object of type `Signal_ObjectV`, the record type generated by the compiler to represent the object state for convenience:

```
type Signal_ObjectV is record
   _object : aliased Protection (1);
   _entry_bodies : Protected_Entry_Body_Access :=
     Signal_ObjectA'Access;
   Open : Boolean := False;
end record;
```

We will omit the callback version of the translation of the protected procedure, since it is almost the same as for the inline interface. The one difference is that the call to the entry service procedure upon completion of the protected procedure is an RTS call.

The compiler translation for a protected entry call is

much simpler than with the inline interface. The compiler no longer generates inline code to select and serve the next entry call. That code has migrated into the RTS. For example, compare the callback version of the translation of an entry call, below, to the inline version, given in Section 3.1.

```
declare
   X     : protected_entry_index := 1;
   Block : Communication_Block;
begin
   Protected_Entry_Call (
     S._object'Access,
     X,
     Null_Address,
     Simple_Call,
     Block);
end;
```

Note that only one RTS call is now required for the entry call, including the wait for completion. The parameters are the same as for the inline version of `Protected_Entry_Call`. The `Block` parameter is now a dummy for this simple form of entry call, but it is retained because it is useful in the implementation of asynchronous entry calls. The interface could be streamlined further by providing separate interfaces for the different types of protected entry calls, and eliminating this parameter for the simple call case.

4 INTERFACE COMPARISON

While the callback interface requires less complex compiler translation of protected constructs, this does not necessarily result in better performance. The basic mechanisms remain the same; they have simply been moved into the RTS. Any savings in execution time must be gained by elimination of procedure calls and control structures used to mediate between the RTS and code generated inline.

To get some indication of the relative performance of the two interfaces, we modified the PIWG benchmarks for Ada 83 rendezvous, simply replacing accept and select statements with entry bodies and task entry calls with protected entry calls [2]. These are simple benchmarks to determine the execution time of tasking constructs in isolation. The results of these benchmarks are shown in Table 1.

The improvement shown by the callback interface, while not dramatic, is encouraging, in particular since there are other improvements in progress which will make the difference proportionally greater.

Besides the raw differences in overhead of these two interfaces, one is interested in their comparative effects on optimizations, including those discussed in Section 2. Though we have not had the resources needed to implement these optimizations for both interfaces, so as to obtain quantitative measurement comparable to those above, we believe it is possible to draw some conclusions from the implementation and analysis we have done.

Table 1: Modified PIWG Benchmarks for Protected Entry Call (in μ seconds)

Category	Inline	Callback
P000001	214	199
P000002	217	192
P000003	215	191
P000004	218	192
P000005	211	195
P000006	225	202
P000008	969	913

Unless a barrier depends on the `'Count` attribute of an entry, reevaluation of barriers after queuing an entry call can be avoided. The inline interface could take advantage of this by only evaluating barriers depending on `'Count` when an entry is queued, evaluating the rest only when some entry is actually executed. This can be done with no additional execution overhead by duplicating entry barrier evaluation code in each case alternative of the entry service procedure, and putting only the barriers dependent on `'Count` in the case for `Queued_Entry_Call`:

```
...
   --  Evaluate all barriers
   loop
      Next_Entry_Call (... ,X, ...);
      case X is
      when Null_Protected_Entry => null;
      when Queued_Protected_Entry =>
         --  Evaluate 'Count dependent barriers
      when 1 =>
         --  Statements for entry body 1
         Complete_Entry_Body (...);
         --  Evaluate all barriers
      when 2 =>
         --  Statements for entry body 2
         Complete_Entry_Body (...);
         --  Evaluate all barriers

      ...

      when n =>
         --  Statements for entry body n
         Complete_Entry_Body (...);
         --  Evaluate all barriers
   end loop;
...
```

Code duplication has the potential to generate more object code, but this may be mitigated by static analysis. In the callback interface, these decisions are all made within the RTS, so the fact that a barrier did not depend on `'Count` would have to be passed into the RTS, and RTS control structures would have to be used to

avoid their evaluation. This is likely to involve additional overhead in all cases.

The inline interface seems to favor static optimization by the compiler. In the callback interface, all entry bodies are executed at the same place, via an indirect call from the RTS. This means that the compiler cannot use special knowledge about the relationship between an entry body and its execution environment to aid in optimization. With the inline interface, the entry body code is not called, but executed inline within the entry service procedure. This allows the compiler to perform optimizations based on the relationship between the protected entry body and the surrounding code.

This is illustrated by the example shown above, where the statements of each entry body are followed immediately by the evaluation of all barriers. The compiler may be able to infer by flow analysis that the executing of a particular entry body does not effect the value of a particular barrier. The code to evaluate this barrier can be eliminated after the execution of that body, helping to limit the duplication of barrier evaluation code and increasing performance. Note that this would require some rather sophisticated interprocedural analysis, since the compiler has to be able to determine that the call to `Complete_Entry_Body` (a nontrivial procedure, which uses pointers and contains calls to other procedures) cannot modify the value of barrier expression, before the expression can be eliminated.

The optimizations that involve saving a Boolean vector of barrier values, and mapping single-component barriers into it, tend to favor the inline interface, which requires such a vector in any case. The inline interface also evaluates all barriers after each protected action, meaning that the final value of the barrier vector will be valid for the next protected action. Providing a barrier vector and keeping it up to date could therefore involve additional overhead with the callback interface.

Note that in the implementation of concurrent programming constructs via source-to-source transformations may require special information be passed to the compiler back end, to prevent incorrect optimizations. This is especially true if we expand runtime system calls inline. In the absence of any warnings, an aggressive back-end optimizer may perform code-movement that invalidates concurrency assumptions between tasks. For example, consider the use of a simple spin lock to implement protected object locking. The body of a protected procedure might look something like the following.

```
while not Test_And_Set (Flag) loop
   null;
end loop;

<critical section>

Flag := False;
```

If the code in `<critical section>` does not modify or use the value of `Flag`, as it generally would not, an aggressive optimizer might rearrange this to give the following.

```
while not Test_And_Set (Flag) loop
   null;
end loop;

Flag := False;

<critical section>
```

Some modern compilers might make such rearrangements to schedule instructions in the pipeline or cache. This of course violates the intent of the flag, something that is not apparent without knowing that this is the translation of a concurrency construct. With modern pipelined processors, this might even occur at runtime, as the hardware schedules instructions in multiple pipelines. To avoid this, barriers to code motion must be provided to the back end, based on the semantics of the tasking constructs being translated.

5 CONCLUSIONS AND FUTURE WORK

We have implemented protected entry calls using two interfaces, one in which all entry body code is executed in line, and one in which it is executed by the RTS through subprogram access values. Our results indicate that the callback interface allows for much simpler translations of protected structures and operations, and eliminates some of the overhead inherent in the inline interface's frequent alternation between the RTS and the application code. However, the inline interface still appears to hold more promise for further optimization.

Since we could not maintain both callback and inline versions of GNAT and GNARL indefinitely, we were forced to choose one version as the basis for continue development. In the experimental results reported above, the callback version had a slight advantage in performance. It also had a big advantage in the simplicity and understandability of both the generated code and the internal logic of the compiler. Given that GNAT is free software, intended to be modifiable and maintainable by the user community, this is a big virtue. While we expect GNAT will be extended to provide some tasking optimizations in the semantic-tree-to-semantic-tree transformation phase, the back-end technology is that of `gcc`, which apparently does not perform the kind of flow analysis that would be needed to take good advantage of the inline model. Therefore, we chose the callback version.

Of course, our reasons for choosing the callback version are peculiar in some respects to GNAT. Perhaps some other compiler will be able to make better use of

the inline approach.

References

[1] T.P. Baker and E.W. Giering. PART/GNARL interface definition. Technical report, Florida State University, July 1994. Available by anonymous FTP from ftp.cs.fsu.edu.

[2] Performance Issues Working Group. The piwg test suite. Available through anonymous ftp from ajpo.sei.cmu.edu.

[3] International Organization for Standardization, International Electrotechnical Commission. *Ada Reference Manual: Version 6.0*, December 1994. ISO/IEC 8652:1995(E).

[4] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C Language] (Draft 9)*, May 1994. P1003.1c/D9.

[5] New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.