Implementing Ada Protected Objects—Interface Issues and Optimization

Extended Abstract

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

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

Implementing Ada Protected Objects—Interface Issues and Optimization

Extended Abstract

ABSTRACT

Ada 95 protected objects are intended to provide a low-level, inexpensive synchronization mechanism as compared to Ada rendezvous. However, protected objects in their full generality provide a large number of well-defined, abstract features that require a complex implementation on many platforms. Some of these features, such as Asynchronous Transfer of Control (ATC) and requeue appear to be relatively independent of protected objects, but greatly impact their implementation. Some form of optimization of special cases is likely to be needed to provide low-cost synchronization.

The choice of interface between the code generated by the compiler from an application program and the runtime system code can influence the ease of such optimizations, as well as the quality of code in general. One interface design choice in particular, that between the inline execution of application code versus its execution through procedure pointers from within the runtime ("callbacks") is sufficiently unclear that we have undertaken to implement both interfaces in the Gnu/Ada Runtime Library (GNARL), the tasking support for the GNAT compiler.

This abstract describes optimizations for some protected object special cases. It also outlines the inline and callback interfaces for protected object runtime services and discusses how they effect code generation and optimization. It also gives some preliminary results from the implementation and testing of these two interfaces.

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 9X compiler and associated runtime system (RTS) being developed separately at New York University [3]. 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].

The Gnu NYU Ada Translator (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 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 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 rendezvous. 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) [2]. 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,

[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.

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 current 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 in particular 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 prototype 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 information on queued calls tends 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.

Further details are provided below. Section 2 discusses means of optimizing both general and restricted protected objects. Section 3 and Section 4 provide a more detailed description of the GNARL inline and prototype callback interfaces for protected objects, respectively. Section 5 compares the two interfaces in terms of inherent complexity and their effect on various optimizations, and presents some preliminary benchmarks. Section 6 provides a summary of the work so far.

2 OPTIMIZATIONS OF PROTECTED OPERATIONS

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, forcing it to evaluate all barriers. 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 THE INLINE INTERFACE FOR PROTECTED OBJECTS

The GNARL inline interface transforms protected procedures and functions as new procedures and functions which take the protected object as an implicit parameter, defers abortion and locks the associated mutex at the start of execution, and unlocks the mutex and undefers abortion before leaving. The interface contains the `Lock` and `Unlock` procedures, which act on an object of type `Protection` (used to represent the state of the object within the RTS), 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.

For example, consider the `Signal_Object` type defined above in Section 1. This type is translated as follows:

```
type Signal_Objectv is record
   _object : aliased Protection (1);
   _service : aliased Service_Record;
   Open : Boolean := False;
   _priority : Integer := 30;
end record;

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;
```

```
procedure Signal_ObjectS
  (O : Address;
   P : out Boolean)
is
   type Signal_ObjectVP is access Signal_ObjectV;
   _object : Signal_ObjectVP := Signal_ObjectVP!(O);
   Barriers : Barrier_Vector
     (1 .. _object._object.Num_Entries);
   Parameters : Address;
   Entry_Index : Protected_Entry_Index
     range 0 .. _object._object.Num_Entries;
   Pending_Serviced : Boolean := False;
   Accumulate_Serviced : Boolean := False;
begin
   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;
   P := Accumulate_Serviced;
end Signal_Objects;
```

This is based on the output of the GNAT 2.04 compiler, simplified and edited for clarity. The protected type has been translated into a record type representing the object state. This includes the application component `Open`, as well as `_object`, a component of the RTS interface type `Protection`, which is used to hold the RTS object state (e.g. the entry queues). `_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 the service structure is only visible to compiler-generated code.

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. 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`.

The compiler assigns an integer value between one and the number of entries that it reports to the RTS through the discriminant on the `_object` component. This is used by the RTS to identify the entry[3]. In this case, there is only one entry, with an entry index of 1. The entry service routine declares a boolean vector with a component for each entry to hold the barrier values.

The entry service procedure `Signal_ObjectS` takes an `Address` parameter to access the object being serviced. This is to allow indirect access to an arbitrary entry service procedure in the translation of protected entry calls, as described below. This address is converted into an access value referencing the object; using "!", an internal shorthand notation for unchecked conversion. The statements of the entry service routine are mostly within a loop. This loop continues to execute entry calls until no more are eligible for execution. It first evaluates all barriers, in this case simply copying the value of `Open` into the only component of the barrier vector. It then calls the RTS routine `Next_Entry_Call` with the `Protection` component and the barrier vector to see which call to service next. `Next_Entry_Call` will return the index of the next entry to be serviced, according to the queuing discipline in effect, in the parameter `Entry_Index`; if there is no such call, it returns a distinguished `Null_Protected_Entry` index. `Next_Entry_Call` may also queue a call, as described below, in which case it returns the distinguished `Queued_Entry_Call` index. Since barriers can depend on the `'Count` attributes of this queued entry call, they must be reevaluated in this case. If the entry call has any parameters, a pointer to them is returned in the parameter `Parameters`.

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

The call to Next_Entry_Call is followed by a case statement on the entry index that it returns. The declarations and statements from each entry body are placed in a case alternative for the corresponding entry index value in this case statement, so that the appropriate entry body processing will be performed. When the entry body statements are complete, a call is made to Complete_Entry_Body to inform the RTS that the call is complete. The effect 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_Entry_Call 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 whether a call will be queued until it is complete. Since the call to the entry service procedure that service the entry call without queuing it may go on to execute other calls that become eligible as a result, the flags from all of the calls to Complete_Entry_Body are accumulated in Accumulate_Serviced, so that if any one if them is True, True will be returned in P.

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;
```

This is translated as follows:

```
declare
   X : Protected_Entry_Index := 1;
   Pending_Serviced : Boolean;
   Block : Communication_Block;
begin
   Defer_Abortion;
   Lock (S._object'Access);
   Protected_Entry_Call (
     S._object'Access,
     X,
     Null_Address,
     Simple_Call,
     Block);
   Signal_ObjectS (S, Pending_Serviced);
   Unlock (S._object'Access);
   if not Pending_Serviced then
      Wait_For_Completion (Block);
      Service_Cancelled_Call (Block);
   end if;
   Undefer_Abortion;
   Raise_Pending_Exception (Block);
end;
```

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. 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 difficult to recompute.

Once the call is pended, the entry service procedure Signal_ObjectS is called to either execute or queue the pended call. Next_Entry_Call 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_Object 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. 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`.

4 THE CALLBACK INTERFACE FOR PROTECTED OBJECTS

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 have prototyped a callback version of protected entry calls in the GNARL. Protected subprograms remain unchanged, since there seemed little advantage in executing them through a callback.

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 above. For use with the callback interface, this type is translated as follows:

```
procedure Signal_Object__WaitE
  (O : address;
   P : address;
   E : protected_entry_index);
function Signal_Object__WaitB
  (O : address;
   E : protected_entry_index)
   return boolean;

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

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

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;

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;
```

The translation of the protected procedure is as for the the inline interface, with the exception that the call to service entries upon completion of the procedure is an RTS call. This translation has therefore been omitted. The entry body `Wait` has been translated into the barrier function `Signal_Object_WaitB` and `Signal_Object_WaitE`. Pointers to these routines are loaded into an array which is used by the RTS to call them. 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. Both of these routines convert the `Address` parameter to an access value to reference the object record, similar to the compiler-generated entry service procedure described for the inline interface. 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 above.

The logic for choosing and executing the entry call described for the inline interface has now largely migrated into the RTS implementation. The compiler translation for an entry call is correspondingly simpler. The entry call described in Section 3 is translated as follows:

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

Only one call to the RTS is now required to complete the entry call, including the wait for completion. The parameters are the same as for the inline version of `Protected_Entry_Call`, though the `Block` parameter is now a dummy. It is still included in this interface, since it is useful in the implementation of asynchronous entry calls. Multiple interfaces for the different types of protected entry calls could be provided to eliminate this parameter for the simple call case, further streamlining the interface.

5 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. 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.

The other thrust of this investigation is comparison of these two interfaces with regard to the ease of optimization. Here we discuss the optimizations outlined in Section 2 with respect to the two interfaces.

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

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

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, a loop within the RTS iterates over the entry bodies of all protected objects, so few assumptions can be made about the contents of those bodies. With the inline interface, the entry body code is exposed at the point of execution, and more can be inferred about it in relation to the evaluation loop.

Consider 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.

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.

6 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 preliminary 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 appears to hold more promise for further optimization.

We plan to further investigate performance improvements for these implementations, and to perform further benchmarks on the results. The final version of this paper will report on the results of these investigations.

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] 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.

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