The Rendezvous is Dead – Long Live the Protected Object [*]

Dragan Macos and Frank Mueller
Humboldt-Universität zu Berlin, Institut für Informatik
10099 Berlin (Germany)
e-mail: {macos,mueller}@informatik.hu-berlin.de
phone: (+49) (30) 20181-276 fax: -280

Abstract

This paper outlines the short-comings of rendezvous and the advantages of protected objects as a means of synchronization in Ada 95. A common Ada benchmark suite, ACES, gives the misleading impression that protected objects are cheaper *per se*. This work shows that the main benefit of protected objects stems from the potential to reduce the number of context switches. Based on the latter observation, a general model to translate rendezvous into protected objects is developed. This model is further refined to replace entire server tasks that make calls to or accept rendezvous. A quantitative evaluation shows the benefits of protected objects for each approach and illustrates that the number of context switches can be reduced at least by 50% when tasks with rendezvous are replaced by protected objects resulting in significant execution time savings for applications requiring many synchronizations.

1 Introduction

In a concurrent language entities for threaded execution (tasks) and methods of synchronization allow the expression of potential parallelism. Methods for synchronization include mutual exclusion with conditional synchronization, semaphores, monitors, remote procedure calls, message passing and rendezvous, just to name the most common features [1]. Ada tasks traditionally employed rendezvous as the only means of synchronization. Cooperating tasks could use rendezvous to directly communicate with each other, much alike synchronous, bi-directional message-passing style. However, if the access to a shared resource had to be arbitrated, a single server task was used that responded to lock and unlock entry calls. The creation of a server task would not have been necessary had other synchronization mechanisms been adopted for this case. Ada 95 fixed this shortcoming by providing protected objects as another option for synchronization. Protected objects were added to the language standard in order to provide for more efficient synchronization and natural expression of synchronization methods at the language level. They offer the expressiveness of monitors for implicit mutual exclusion. Yet, while monitors provide explicit synchronization, protected objects use implicit synchronization through entries with barriers. An entry call can be executed in the context of any task when the barrier evaluates to true so that a task may serve as a proxy for the actual caller. This proxy model results in an optimized runtime behavior. The Ada 95 Rationale [9, section 9.1.3] already points out the potential savings in terms of context switches. This observation poses a number of questions:

- How much more efficient are protected objects?
- Where do the savings come from?
- How can rendezvous be avoided?
- Should existing Ada 83 rendezvous be replaced by protected objects?
- Is there a translation scheme for replacing rendezvous with protected objects?
- Should rendezvous always be avoided?

This paper answers the above questions based on performance measurements for the different synchronization methods. It offers a general translation scheme to replace rendezvous by protected objects. Two additional, more specific translation schemes are derived for eliminating server tasks. Finally, measurements for a number of applications with rendezvous are given and compared with semantically equivalent translations into protected objects.

2 Related Work

Discussions of protected objects can be found in the Ada 95 Rationale [9] and Ada textbooks [4, 5]. Performance considerations are mentioned but no quantitative analysis is presented and no translation schemes are offered. Prior to the availability of Ada 95, a study by Ledru gave a translation scheme for expressing protected types in terms of Ada 83 enhanced by operating-system level semaphores [10]. Baker *et. al.* report performance gains for optimizing the Gnat runtime system [3]. Their focus is centered around improvements relative to earlier versions of the runtime system. Another study by the same group reports performance improvements for coarse-grained locking policies within the

[*]This work was supported in part by Deutsche Telekom AG, contract # 4160/75004.

runtime system, even for multiprocessors, and special versions of the runtime system with reduced functionality [7]. In contrast, the objective of this paper is to develop and evaluate a scheme to replace rendezvous by protected objects.

3 Motivation

Ada 95 offers two basic options for synchronizing tasks: rendezvous and protected objects. Disregarding semantic differences, the choice between these two language features should be driven by their associated performance overhead.

In the following, the Ada Compiler Evaluation System (ACES) [6] was used to obtain performance measurements. ACES serves as a test system to analyze the functionality and the performance of Ada compilers. It may be used as a regression test suite during compiler development, as part of a compiler validation process, and for comparing different compilers.

The performance tests included task creation, simple rendezvous and protected operations. They were conducted on a SPARCengine Ultra 1/170E Creator (167 MHz processor) under Solaris 2.6 using Gnat 3.10 with the runtime library built upon native Solaris threads. The measurements in Table 1 show for certain Ada features (column 1) the corresponding test from the ACES test suite (column 2), the mean execution times (column 3) and the variance of the latter (column 4). A divisor in column 2 indicates that the reported time was adjusted to represent a single operation, not multiple operations.

Ada feature	ACES test	mean [μs]	var.
create/terminate	tk_lf01 / 10	398.93	2.6%
entry/accept	tk_lf03 / 20	110.41	0.6%
protected function	pt_po04	4.22	0.6%
protected procedure	pt_po05	7.19	0.2%
protected entry	pt_po06	10.30	0.6%

Table 1: Overhead of Ada Tasking Constructs

Task creation together with termination is almost an order of a magnitude slower than a rendezvous. As a result, Ada tasks are mostly used for coarse-gained parallelism and tasks are often created during program initiation, only. The measurements also seem to indicate that protected entries are an order of a magnitude faster than rendezvous. However, these numbers are misleading. Rendezvous tests with an entry call and accept pair *include the overhead of a context switch* between the calling and accepting tasks. The measurements for protected entries cover the overhead for gaining exclusive access to the protected object in the absence of contention, *i.e. no context switches are taking place.* The measurements also show further savings when protected procedures or even functions can be used instead of protected entries. Another test program should be added to ACES to cover the contention case similar to the rendezvous tests.

As mentioned before, the primary focus of this paper is on protected entries since they provide the means of synchronization between tasks. The remainder of this paper evaluates the question if and when protected entries should be used instead of rendezvous based on the potential savings in performance.

4 A Simple Example

The following example in Figure 1 depicts a set of client tasks that need to access a common resource in mutual exclusion, *i.e.* an exclusive lock is acquired before using the resource and released afterwards. This lock can be implemented *via* a server task that serializes acquire and release lock pairs and thereby governs the access to the resource. Alternatively, the lock may be realized *via* a protected object that ensures through barrier conditions that after an acquire a release has to be performed before the next acquire completes.

The purpose of this example is to illustrate both the syntactical and semantic similarities between rendezvous and protected objects, rather than to depict an efficient implementation of mutually exclusive access. It also underlines the potential to replace an active task by a passive synchronization mechanism. In addition, the use of protected objects reduces the number of context switches. Rendezvous require a context switch at every call/accept pair.[1] Protected entries are executed within the context of a client task, which is either the calling task if the barrier evaluates to true upon the initial call or possibly in a proxy task (another client) that modified variables of the barrier expression.

5 From Rendezvous to Protected Objects

At first, an attempt to formulate a general approach for replacing rendezvous with protected objects in Ada applications shall be made. This approach does not aim at eliminating tasks. The main consideration for this model lies in the data flow of rendezvous (see Figure 2a). When a call to a rendezvous is made, input parameters are sent to the acceptor. The acceptor receives the parameters and executes the accept body. When completing the accept body, parameters may be sent back to the caller, *i.e.* the caller receives output parameters upon completion of a rendezvous.

The data flow of a rendezvous can be resembled by a protected object that serves as a temporary storage to hold input and output parameters (see Figure 2b). The call to a rendezvous is replaced by an initial call of a protected entry `Call` to store input parameters in the protected object. Instead of accepting the rendezvous, the acceptor task calls the protected entry `Accept_Begin` and extracts the original input parameters. Next, the accept body is executed followed by a protected entry call `Accept_End` to store result parameters in the protected object. A final protected entry `Queue` transfers the output parameters back to the calling task.

After preserving the data flow, the control flow of a rendezvous has to be resembled as well. First, the sequence of stages of a rendezvous (call, start accepting, complete accepting, complete rendezvous) has to be ensured for the protected object (Call, Accept_Begin, Accept_End, Queue). Figure 3 depicts a protected type `Rendezvous` that preserves the control flow by constraining the protected entries into the desired sequence *via* barrier expressions. A stage counter guards each entry. When a barrier evaluates to true and an entry call is executed, this stage counter is incremented to open the barrier of the next entry. Notice that the actual parameters of the calls are denoted by `Some_Type`.

[1] Optimizations for simple rendezvous without accept body may actually avoid certain context switches. However, this paper concentrates on the general case.

```
task type Client;                 task Resource is               protected Resource is
P : array (<>) of Client;           entry Acquire;                 entry Acquire;
                                    entry Release;                 entry Release;
                                                                 private
                                                                   Empty : Boolean := True;
                                  end Resource;                  end Resource;

task body Client is               task body Resource is          protected body Resource is
begin                             begin                            entry Acquire when Empty is
  loop                              loop                           begin
     Calculate(Result);                                              Empty := False;
     Resource.Acquire;                accept Acquire;              end Acquire;
     Access_Resource;                                              entry Release when not Empty is
     Resource.Release;                accept Release;              begin
     exit when Done;                  exit when Done;                Empty := True;
  end loop;                         end loop;                      end Release;
end Client;                       end Resource;                  end Resource;

(a) Client Tasks                  (b) Accepting Server           (c) Protected Object
```

Figure 1: Mutual Exclusive Access to a Resource

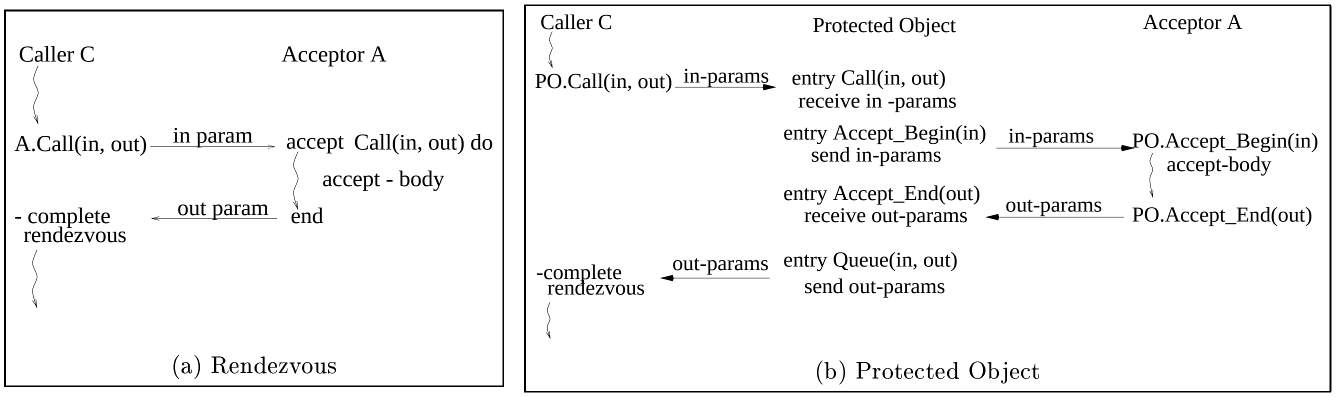

Figure 2: Data Flow for Rendezvous and Protected Object Translation Scheme

The second issue concerns the origin of the call to the entry `Queue`. One could simply require the call of the original rendezvous to be replaced by a sequence of the two calls `Call` and `Queue` containing input and output parameters, respectively, in separate calls. However, a more elegant solution is outlined in Figure 3. The protected entry `Call` is requeued on the protected entry `Queue` to await completion of the accept body. This approach requires only the original `Call` of the rendezvous to be present on the caller's side.

The semantics of the rendezvous is preserved through this translation since entry parameters are governed by the same rules for rendezvous and protected objects. The initial call is requeued with abort enabled to provide the equivalence of abortion of the calling task at synchronization points. Exceptions are handled correctly even if they were raised during a proxy execution since Ada requires the exception to propagate into the caller, regardless of who executes an entry body. However, accept statements have been replaced by protected entry calls, which poses problems with the translation of select statements. An acceptor may perform selective accepts with multiple accept alternatives. A direct transformation into conditional entry calls is not possible since only one entry call is permitted.

A single conditional entry in a select may be sufficient to determine later which entry call had actually been called. This approach follows the continuation-passing style of translation [2]. A parameter of the conditional call indicates which protected object should be called as depicted in Figure 4. The expense of this method lies in another protected entry call whereas less general approaches can cope without the continuation-passing style and the associated overhead.

Default and delay alternatives are legal both for accepting rendezvous and calling entries in select statements. Terminate alternatives are only permitted in selective accepts. A replacement cannot easily be found but additional logic can be added to abort tasks with terminate alternatives when all client tasks, *i.e.* the original callers of rendezvous, have terminated.

The translation scheme in Figure 3 even handles the translation of requeue statements originally used with rendezvous entries. Such a requeue statement is replaced by a call to `Accept_End` with the protected object as a first parameter, where the object corresponds to the entry of the original rendezvous. Again, continuation-passing style is utilized for providing the correct control flow.[2]

[2]The optional `with about` is only sketched here, another parameter to `Accept_End` and two alternatives in `Select_Accept` would be required.

```
type Rendezvous;
type Rendezvous_PO is access Rendezvous;
protected type Rendezvous is
  entry Call(X : Some_Type; Y : out Some_Type);
  entry Accept_Begin(X : out Some_Type);
  entry Accept_End(Requeue_Object : Rendezvous_PO;
                   Y : Some_Type);
  entry Queue(X : Some_Type; Y : out Some_Type);
private
  Stage : Integer := 0;
  In_Param, Out_Param : Some_Type;
  Req_Object : Rendezvous_PO := null;
end Rendezvous;

protected body Rendezvous is
  entry Call(X : Some_Type; Y : out Some_Type)
    when Stage = 0 is
  begin
    In_Param := X;
    Stage := Stage + 1;
    requeue Queue with abort;
  end;
end Rendezvous;
```

```
entry Accept_Begin(X : out Some_Type)
  when Stage = 1 is
begin
  X := In_Param;
  Stage := Stage + 1;
end;

entry Accept_End(Requeue_Object :
  Rendezvous_PO; Y : Some_Type)
  when Stage = 2 is
begin
  Out_Param := Y;
  Req_Object := Requeue_Object;
  Stage := Stage + 1;
end;

entry Queue(X : Some_Type;
   Y : out Some_Type) when Stage = 3 is
begin
  Y := Out_Param;
  Stage := 0;
  if Req_Object /= null then
    requeue Req_Object.Call [with abort];
  end if;
end;
```

Figure 3: Rendezvous as a Protected Type

The sources for optimizing synchronizations for rendezvous are similar to optimizations for their translation into protected objects. A simple rendezvous without accept body may not result in a context switch. If the acceptor is waiting, the caller can complete the rendezvous right away. Conversely, if the caller is waiting, the acceptor can complete the rendezvous and proceed. Following the above translation scheme, a call to `Accept_End` can be eliminated for empty accept bodies. If the acceptor is waiting, the caller can complete the calls `Call`, `Acceptor_Begin` (as a proxy) and `Queue`. Conversely, if the caller is waiting, the acceptor can complete `Acceptor_Begin` and `Queue` (the latter as a proxy).

The general translation scheme of rendezvous into protected objects does not handle a variable number of parameters. It clearly comes at the expense of multiple entry calls on both sides, even if the second call on the caller's side is hidden through a requeue statement. If the performance numbers of the ACES test suite reported in section 3 were the whole truth, the translation scheme might prove to be useful. Unfortunately, this is not the case, as discussed in more detail in section 7. Before turning to a quantitative comparison, other options for a translation shall be considered and included in the later evaluation.

6 Eliminating Server Tasks

A number of standard patterns for the utilization of Ada tasks have been described in literature and used in practice. For instance, a single server task is often employed to manage resources or to communicate data. In particular, servers accepting rendezvous and those making calls to rendezvous can be distinguished.

6.1 Accepting Servers

Figure 1 depicted an accepting server guarding access to a common resource. Had the general translation scheme of section 5 been applied to part (b) of the figure, each rendezvous entry would have been replaced by a protected object with four entries while preserving the task structure. Instead, part (c) of the figure depicts a translation into one protected object with 2 entries that replaces the server task.

Accepting server tasks may be replaced by a protected object following the translation scheme in Figure 5. The figure omits the declaration of variables `vars` that can be realized in a `declare` block surrounding the declaration of the protected server. The barriers of accepts translate into barriers of protected entries enhanced by a check on the stage counter to ensure the proper sequence of synchronizations. Accepts of the same select statement check for the same value of the `stage` counter, consecutive accepts check for increasing values to the point of wrap-around at the last synchronization point in the server loop.

The main advantage of this translation scheme lies in the elimination of a task. Rendezvous with a server task result in context switches while calls to protected entries may be executed as proxies within the context of the caller.

The main disadvantage of this translation scheme is caused by semantic restrictions of protected entries. First, potentially blocking operations must not be called from protected entries. This implies that the translation scheme can only be applied if accept bodies do not contain such calls. In particular, nested rendezvous cannot be handled this way. Second, statements outside the accept bodies in the server task cannot easily be integrated. For example, `stmt2` and `stmt3`, if present in the original code, had to be placed in `e1`, `stmt1` would be part of the last entry `e3`. An additional entry `e0` with a distinct barrier condition would contain `stmt0` and

```
protected Selector is
  entry Call(PO : Rendezvous_PO; Params ...);
  entry Acceptor(PO : out Rendezvous_PO);
private
  Selected : Rendezvous_PO;
  Called : Boolean := False;
end Selector;

protected body Selector is
  entry Call(PO : Rendezvous_PO;
             Params ...)
    when not Called is
  begin
    Called := True;
    Selected := PO;
    requeue PO.Call;
  end;
```

```
  entry Acceptor(PO : out Rendezvous_PO)

    when Called is
  begin
    Called := False;
    PO := Selected;
  end;
end Selector;
```

```
(a) protected object to dispatch multiple alternatives in select
```

```
Selective_Accept.Entry1(Params...);

task body Selective_Accept is

  ...
  select
    accept Entry1(Params...) do
      -- accept body;
    end Entry1;
  or
    accept Entry2(Params...) do
  ...

(b) call and selective accept
```

```
Entry1, Entry2 : Rendezvous_PO;

Selector.Call(Entry1, Params...);

task body Selective_Accept is
  PO : Rendezvous_PO;
     ...
  Selector.Acceptor(PO, Params);
  if PO = Entry1 then
    Entry1.Call(PO, Params);
    -- accept body;

  elsif PO = Entry2 then
  ...

(c) equivalent for POs
```

Figure 4: Translation Scheme for Selective Accepts

`stmt1` to be called upon initialization. Other statements can be handled analogously. Notice that global variables modified in these statements do not pose any problems as long as output parameters are evaluated before executing these statements. However, this scheme is also subject to absence of potentially blocking calls in any of the statements surrounding entry calls to rendezvous in the server task.

In practice, the first constraint may not necessarily pose a problem. Nested rendezvous can typically be translated into distinct protected entries with consecutive stage counter checks as barriers. The else part of a select (default part/selective delay) can be handled within the protected entry of the outer rendezvous by checking on the original barrier of the inner one. The `Count` attribute allows the selection of the proper inner action. A complete translation scheme has to be omitted due to its length.

6.2 Calling Servers

Another commonly used pattern of rendezvous involves a call from a server to a number of client tasks to communicate distinct messages to each of them. For example, in data-oriented parallel execution, a task may operate on a subspace of the problem. If this subspace is statically assigned, an initial call to an entry of the task communicates the location of this subspace.

The task of the calling server can be replaced by a protected object with an inverted direction of the data-flow of entry parameters. The accept statements of client tasks are replaced by calls to protected entries whose barriers are enhanced by a stage check as seen before. Select alternatives with multiple entries can be dealt with in a continuation-passing style similar to the general translation scheme of section 5. Statements within the server between calls may be moved into protected entries but are subject to the same constraints as for accepting servers, *i.e.* potentially blocking calls are not permitted.

Apart from the inverted direction of data flow and the replacement of accept statements with calls, the translation is similar to that depicted in Figure 5. However, the replacement of the caller through a protected object may not be trivial as seen in the example in Figure 6. The loop structure of the caller in (a) cannot be embedded into a protected entry call. This problem may also occur when accepting servers are replaced. It can be handled by first transforming `for` and `while` loops into `exit when` loops seen in (b) and then translating the resulting code into a protected object such as (c). Other code sequences such as nested loops can be handled similarly but are subject to the same constraints as before, *e.g.* potentially blocking calls (besides

```
task body Server is
  vars;
begin -- stmt0;
  loop -- stmt1;
    select
      accept e1(args1) when cond1 do
        body1;
      end; -- stmt2;
    or
      accept e2(args2) when cond2 do
        body2;
      end;
    end select; -- stmt3;
    accept a3(args3) when cond3 do
      body3;
    end;
  end loop;
end Server;

(a) Accepting Server
```

```
protected Server is
  entry ...;
private
  stage : integer := 0;
end Server
protected body Server is
  entry e1(args1) when stage = 0 and Cond1 is
  begin
    body1;
    stage := stage + 1; -- stmt2; stmt3;
  end;
  -- ditto for e2
  entry e3(args1) when stage = 1 and Cond2 is
  begin
    body2;
    stage := 0; -- stmt1;
  end;
end Server;

(b) Protected Server Object
```

Figure 5: Translation Scheme for Accepting Servers

```
                         i : Integer := range'First;      ... private x : Integer ...
                         ...                              entry call(i : out Integer)
for i in range loop      loop                                when true is
                           exit when i > range'Last;      begin
  call(i);                 call(i);                         i := x;
                           i := i + 1;                      x := x + 1;
end loop;                end loop;                        end call;

(a) Calling Server       (b) Transformed Loop             (c) Protected Server Object
```

Figure 6: Transforming Calling Servers

the rendezvous call subject to replacement) must not occur in the original code. The elimination of servers restricts the semantics of rendezvous with arbitrary selection of open entries to a FIFO ordering for protected objects, which is a legal specialization.

7 Quantitative Analysis

A quantitative evaluation of the different approaches was performed. They were again conducted on a SPARCengine Ultra 1/170E Creator (167 MHz processor) under Solaris 2.6 using Gnat 3.10. At first, the experiments utilized the Gnat runtime library [8] for native Solaris threads. However, the variance of execution times for this library turned out to be very large, *i.e.* a consecutive run of the same test programs took twice as long as the previous run under the same machine load. Closer investigation revealed that the number of context switches also varied significantly between consecutive runs. The reason for this behavior is explained by the round robin scheduling of Solaris threads. Context switches due to expiring time slices cannot be controlled and result in the observed variations. Thus, the measurements were conducted with the FSU Pthreads library [11] to yield reproducible measurements.

The set of test programs included a number of sample applications. An implementation of the dining philosophers uses nested rendezvous and requeues entry calls. This represents a complex case of a selective acceptor as a server task. The timing measurements represent the sum of the times for each task excluding durations during which the task was suspended in a delay statement. A producer/consumer application uses a buffer task with a storage capacity of 10 items to compensate between the speed of a producer that forwards data into the buffer and a consumer that retrieves data from the buffer. Here, a more simplistic selective acceptor represents the server task. The entire program is timed, including I/O operations. A numerical application, an implementation of the Gauss/Seidel algorithm for partial derivation along the lines of Barnes [4], is realized *via* a calling server for client initialization and uses barriers to synchronize iteration steps of the client tasks. The barriers constitute another selective acceptor as a server task. Measurements labeled “Gauss/Seidel” in Table 2 refer to these barriers while “calling server” refers to the client initialization of the program. Barnes' implementation lacks barrier synchronizations, which may result in non-terminating programs when a (subset of) client tasks iterates without letting other tasks regain the CPU. This behavior was observed for strict priority FIFO scheduling with FSU Pthreads, which constitutes a legal Ada 95 option for runtime implementations. Thus, barriers were added to correct Barnes' example. Context switches were counted by comparing the value of `System.Tasking.Self` against a previously obtained value around synchronization points.

Name	Rendezvous	General	Servers
dining	0.009213	0.015780 (+71.28%)	0.003941 (-57.22%)
prodcons	0.885687	2.005755 (+126.46%)	0.543666 (-38.62%)
gauss_seidel	0.005530	0.013927 (+151.84%)	0.003822 (-30.89%)
calling server	0.003067	0.003833 (+24.98%)	0.002618 (-14.64%)

(a) Execution Time [seconds] and Change [%]

Name	Rendezvous	General	Servers
dining	110	116 (+5.45%)	63 (-42.73%)
prodcons	33120	33120 (0.00%)	1382 (-95.83%)
gauss_seidel	238	238 (0.00%)	119 (-50.00%)
calling server	49	49 (0.00%)	17 (-65.31%)

(b) Number of Context Switches and Change [%]

Table 2: Performance of Sample Applications for Translation Schemes

Table 2 depicts the measured execution times in (a) and the number of context switches in (b). The changes of measurements for the test programs (column 1) are shown relative to the rendezvous version of each application (column 2). The following numbers were obtained for the general translation scheme of section 5 (column 3) and the server translation schemes of section 6 (column 4).

The performance measurements indicate that the general translation scheme has a higher overhead than the rendezvous version. This may come as a surprise to someone who had only consulted the ACES results in Table 1 before. Even if the overhead of context switches was considered as well, one would not expect significant increases in execution time for the general scheme while the number of context switches remains the same. This decrease of performance illuminates that four protected entry calls are actually more expensive than a call/accept pair of a rendezvous. Does this mean the general translation scheme is an academic exercise? If the performance loss is due to the overhead of entering the runtime system, then it is likely that runtime optimization would fix this problem. Special code could even be generated for cases when two distinct entries are always executed in sequence due to barrier conditions (see Figure 3). This is underlined by the performance of the optimized general scheme, which almost matches the server translation schemes.

The measurements also indicate a correlation between reduction in context switches and execution time. In general, the actual reduction of context switches depends on the application and on the hardware architecture. For a uniprocessor, context switches due to rendezvous from one client to the server task and then to another client are reduced by eliminating the server. Thus, context switches occur directly between client tasks with protected objects. This cuts the number of context switches in half since two switches of a rendezvous (client/server/client) are reduced to one switch for protected objects (client/client). If the server accumulated several calls before blocking, *e.g.* for an n-element buffered server, then $2n + 4$ switches for rendezvous can be reduced to a single switch for protected objects.

The number of context switches for multiprocessors depends on the ratio of tasks to processors and the assignment of tasks to processors. But when the number of processors is less than the number of tasks, a reduction of context switches results in performance benefits. Even if enough processors are available, server tasks may become a bottleneck. Execution may be suspended on the client side (during calls) and on the server (without incoming calls), and resumption of execution may be a costly operating system operation. Protected objects do not require resumption of execution so that more accesses can be served during the same time. [3]

Having provided a motivation for the overall results, specific numbers can be discussed. For the dining philosophers the number of context switches due to synchronization is 80 and 40 for the first two and the last versions of the application, respectively. This is an example where two context switches for a rendezvous are replaced by one for a protected object. The remaining context switches are due to randomized delays. Slight differences in the order of execution may occur during initialization, as seen for *General*. Execution time savings for *Server* exceed 57% and are due to the heavy use of synchronization in the timed portions of the application. A hand-tuned version of the server translation scheme even showed over 61% savings in time. *General* adds an execution time overhead of over 70%.

The producer/consumer application reduced switches to 1/24th for*Servers*, which was expected since the buffer contains 10 elements, hence every $2 * 10 + 4$ switches were replaced by just one. About 38% of the execution time is saved. The time savings for *Servers* are skewed by the dominating overhead of I/O. *General* becomes very expensive at 126% additional overhead due to another protected object to translate selective accepts (see Figure 4). Without this scheme, the additional execution time overhead was still 55% but busy waiting may occur in the server.

For the accepting server part of the Gauss/Seidel algorithm, the number of context switches was cut in half for *Servers* due to the reasons discussed before. The calling server translation reduces two switches of a rendezvous to none at all since the server task can be completely replaced by a protected object. A constant overhead of 17 context switches is due to switching between the 16 clients and the environment task for all three versions of the program. Execution time savings of about 14% for *Servers* are skewed by the dominating overhead of arithmetic operations. *General* is prohibitively expensive for the accepting server at over 150% additional execution time overhead while the calling server still imposes almost 25% additional overhead.

[3] At the time of the experiments, Gnat did not provide the means to set the thread concurrency level for native threads, which is a requirement under Solaris to exploit multiple processors on a shared-memory multiprocessor. Thus, true parallelism cannot be tested with Gnat under Solaris.

8 Conclusion

This work explores the cost of rendezvous and protected objects, the two synchronization mechanisms of Ada 95. It shows that the main source of performance improvements is a reduction in the number of context switches. First, a general scheme for the translation of rendezvous into protected objects is offered. Programs translated by this scheme experience the same number of context switches as their original counterparts but have longer execution times showing also that protected objects are not cheaper *per se*, as might be expected from ACES benchmarks. Second, schemes to replace accepting and calling servers are offered, which reduce context switches at least by 50% and result in significant performance benefits. These results should be interpreted as follows:

- Tasks should only be used to express parallelism, server tasks should be avoided by all means.
- New Ada applications should use protected objects for synchronization instead of rendezvous unless they communicate data in a message-passing style between peers (and not between client/servers).
- Existing Ada 83 rendezvous should be replaced by protected objects for server tasks following the translation schemes for accepting and calling servers (see section 6).
- In general, it only pays to replace a call/accept pair of a rendezvous with a single protected entry. More protected entries may become prohibitively expensive when additional context switches occur.
- The main performance benefits come from fewer context switches when entire tasks can be replaced by protected objects.
- The general scheme to translate rendezvous into protected objects (section 5) may become feasible as a building block to implement rendezvous when optimizations are applied to merge consecutive runtime calls into one call (runtime and compiler optimizations).

Future work may include the last item and automatic translation according to the server translation schemes as a preprocessor or in the compiler. Also, performance studies for other compilers may be conducted but it can be expected that the findings will be similar as long as protected entries can be executed as proxies.

References

[1] G. R. Andrews. *Concurrent Programming – Principles and Practice.* Benjamin Cummings, 1991.

[2] A. W Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] T.P. Baker, D.-I. Oh, and S.-J. Moon. Low-level Ada trasking support for GNAT – performance and portability improvements. In *Proceedings of the 13th Annual Washington Ada Symposium*, July 1996.

[4] J. G. P. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.

[5] A. Burns and A. Wellings. *Concurrency in Ada.* Cambridge Press, 1995.

[6] Ada Information Clearinghouse. Ada compiler evaluation system (ACES). Available from http://sw-eng.falls-church.va.us/AdaIC/compilers/aces/.

[7] D.-I. Do and T. P. Baker. Optimization of Ada'95 tasking constructs. In *TRI-Ada '97 Proceedings.* ACM, 1997.

[8] E.W. Giering, Frank Mueller, and T.P. Baker. Features of the gnu Ada runtime library. In *TRI-Ada*, pages 93–103, November 1994.

[9] Intermatrics. *Ada 95 Rationale*, February 1995.

[10] P. Ledru. Translation of the protected type mechanism in Ada 83. *ACM Ada Letters*, 15(1):64–69, January 1995.

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