Ada*Tasking: from Semantics to Efficient Implementation†

T. P. Baker ‡
G. A. Riccardi
Department of Computer Science
the Florida State University
Tallahassee, Florida 32306

(regenerated from March 1985 LaTeX sources)

Abstract

The semantics of Ada tasking are reviewed, and used as the basis for a discussion of implementation techniques. It is demonstrated that Ada tasking can be implemented for a single shared processor by simple, straightforward, and efficient algorithms. Factors affecting execution times of tasking operations are explained.

Details that do not significantly affect efficiency are mostly avoided, as are those which are likely to depend on a particular machine architecture. Some algorithms used by the authors in a functional implementation of the full Ada language are described.

1 Introduction

Ada is a the standard programming language of the United States Department of Defense[Ada83]. It is intended to be suitable for a variety of computer applications, including real-time embedded systems, such as autopilots in missiles and aircraft. Ada is also viewed as a potentially useful language for systems programming and industrial process control. An important feature of the language

*Ada is a trademark of the U.S. Department of Defense (AJPO).

†This work supported in part by U.S.A.F. Contract F08635-82-C-0122, Armament Laboratory, Eglin A.F.B.

‡On leave at the Department of Computer Science, University of Washington, Seattle, Washington 98195.

is tasking, which provides the capability of programming multiple concurrent processes, called tasks. Tasks execute in parallel, and may communicate with one another.

In order to support the concurrent execution of tasks, an Ada implementation must provide for storage allocation, task scheduling, and intertask communication. These are functions that are typically performed by the kernel of an operating system. Ada is so specific in its requirements, however, that it is extremely unlikely that a given existing operating system will make such services available in a form that they can be directly used by Ada programs. One of the tasks of an Ada implementor is therefore to develop a runtime environment for Ada tasks.

We call the collection of code and data structures that provide the runtime environment for Ada tasks the *tasking supervisor*. The tasking supervisor may be implemented directly, on a bare machine, or may be implemented as a subsystem, under another host operating system. From the point of view of interaction between an Ada task and the tasking supervisor, this should make little difference, since Ada tasks still need interface directly only with the supervisor. The code of the supervisor is likely, for reasons of compactness and compilation efficiency, to be mostly shared, though some pieces of it may be generated in-line within tasks by the compiler. Even if generated in line, we will consider such code logically a part of the tasking supervisor.

The tasking supervisor must be implemented in such a way as to as to insure *mutual exclusion* between operations that access common data structures or allocate common resources. That is, it must not be possible for parallel execution of supervisor code by different Ada tasks to result in incorrect operation of the system, through interleaved access and update of common data structures. Supervisor operations should not be interruptable by an Ada task. That is, if a task requests action by the supervisor no Ada task should be permitted to execute on the same processor until the supervisor has completed its work.

For most supervisor requests it must be assumed that the supervisor will not necessarily return control of the processor to the requesting task. In particular, as a result of the supervisor action, Ada's semantics may require blocking the task requesting service, or preempting the processor in favor of a higher priority task that has become unblocked.

The actions performed by the supervisor may be rather complex, and consequently may take a significant amount of time to execute. A programmer who is concerned with execution speed or predictable execution timing therefore needs to have some appreciation of the work of the tasking supervisor, and how it is likely to affect the execution timing of a program. This paper is intended to help fill this need. In the paper we try to give a simple and and understandable presentation of the tasking features of Ada, and how they may be implemented on a single processor system, with attention to how the efficiency of operations involved in tasking is likely to be related to factors under a programmer's control. Some previous knowledge of Ada on the part of the reader is desirable,

though the paper does review most of the important concepts of tasking.

For the reader interested in knowing more about Ada tasking, there is a sizeable body of literature. An operational semantic definition for Ada using the Semanol language is given in [BBH80]. Lovengreen and Bjorner [LB80] use the Vienna Definition Language to present a formal model of tasking. Clemmensen [Cl82] uses a more denotational approach and also considers a distributed, or multiprocessor, environment for tasking. Falis [Fa82] gives a description of the interface between a tasking supervisor and Ada programs, including descriptions of all necessary supervisor operations. A detailed description of the states of a task and the changes in state induced by various tasking operations is given in [HPT81]. The efficiency of various implementation strategies, as well as the semantics of the tasking operations, are addressed in [St80], [HN80], [Hi82] and [JA82]. These papers are mainly concerned with the rendezvous operations. Stevenson [St80] considers a variety of translations of tasking operations into a lower-level language. An implementation of Ada tasking on a specific machine model is given by Haberman and Nassi in [HN80]. Hilfinger [Hi82] considers certain special case tasks which allow for an efficient implementation strategy which cannot be used for many Ada tasks. Jones and Ardo [JA82] conducted experiments to test implementation strategies in distributed environments. More recent work on distributed implementations is reported in [Co84] and [We84].

The usefulness of some of the above references is limited by the changes made to the Ada language since publication of the preliminary definition [Ada79]. Major changes to tasking, including elimination of the initiate statement and the addition of terminate alternatives for selective wait statements, were made in the first proposed standard [Ada80]. The revised proposed standard [Ada82] introduced the notion of the completion of a declarative unit and made changes in the activation and termination of tasks. The current definition [Ada83] made less significant changes in tasking, but prescribes a different treatment of shared variables and aborted tasks. All these changes have been beneficial, since they helped to make tasking more efficiently implementable. However, due to the extent of the changes, any reports which reference neither [Ada82] nor [Ada83] should be carefully checked against the current standard ([Ada83]) to determine which parts are still valid.

The content of the present paper has been influenced by some of the work cited above, but is mostly the result of an independent analysis of Ada tasking done by the authors during the design of a runtime support system for the FSU Ada cross-compiler. This analysis was done over the period November 1982-January 1983. Because of the changes that were being made in the language definition at that time, the authors relied most heavily on the draft version of the new Ada language definition, [Ada83], and on standard techniques from the field of operating systems. The result of this work is a complete and fully functioning tasking supervisor, consisting of approximately 6KB of Z8002 machine code. This tasking supervisor has been used in internal prevalidation testing of the FSU cross-compiler, and has proved satisfactory for all of the applicable Version

1.4 ACVC tests. The techniques described here are primarily abstractions of those used in this implementation, supplemented by some obvious alternatives. A more detailed description of the FSU implementation can be found in [BR83a], [BR83b], and [BR84].

A number of important implementation issues related to Ada tasking are not covered in this paper. In some cases, this is because they do not have significant effects on the efficiency of tasking operations, or because they characteristics of particular machine architectures. In others, it is just that they are too complex to discuss in a paper of this nature. Among these are issues related to hierarchical storage systems and multiprocessor or distributed implementations, as well as special-case optimizations related to tasking.

The paper is organized around a review of the Ada language features related to tasking. As each feature is discussed, so also is its implementation.

2 Tasks and Task Execution

In Ada, tasks are objects. Each task has a unique type, which is specified in an object declaration or allocator (an expression of the form "NEW ...") that causes the **creation** of the task. Each task type is declared in two separate parts: a task specification and a task body. (See Figure 1.) The specification gives a sequence of entry declarations, which define the communications interface of tasks of that type. The body gives the rest of the description of the task type.

All declarations in Ada need to be *elaborated* at run time. In general, a compiler must generate code for the elaboration of declarations. The declarations within a task body in general require elaboration code. This may be rather lengthy and is executed as part of task activation. Elaboration of a task object declaration also requires elaboration code, and results in creation of a task.

Each task has a lifetime, from the moment it is created until its storage is deallocated. During a portion of its lifetime, the task is executing. The execution of each task proceeds in parallel with that of all other tasks. Each task may thus be viewed as having its own individual thread of control, along which execution of the task proceeds.

Ada requires that "each task can be considered to be executed by a logical processor of its own" [Ada83]. Since an indeterminate number of tasks may be dynamically created, every implementation must make provision for sharing processor time and memory among tasks, in a way that is transparent to the tasks. From time to time the execution of a task may be *blocked*, that is, forced to wait for some event external to the task. Examples of such events are self-imposed time delays, termination of a subordinate task, and completion of the operations involved in intertask communication. A task that is not blocked is said to be *ready*. When a task on a shared processor blocks itself, the supervisor must determine, from among the currently ready tasks waiting for that processor, which one is to execute. This function of the supervisor we call *dispatching*.

```
task type BUFFER is                                        -- task specification
     entry PUT_IN (X: ITEM_TYPE);                          -- entry declaration
     entry TAKE_OUT (ITEM: out ITEM_TYPE);
end  BUFFER;

task body BUFFER is                                              -- task body
     SIZE: constant INTEGER:= 10;                          -- declarative part
     COUNT,NEXT_IN,NEXT_OUT: INTEGER:=0;
     B: array(0..SIZE-1) of ITEM_TYPE;
begin                                                  -- sequence of statements
      loop
           select                                                -- selective wait
                when COUNT < SIZE =>                                    -- guard
                accept PUT_IN(X: ITEM_TYPE) do     -- accept alternative
                     B(NEXT_IN):=X;
                     NEXT_IN:=(NEXT_IN+1) mod SIZE;
                     COUNT:=COUNT+1;
                end PUT_IN;
           or when COUNT > 0 =>
                accept TAKE_OUT(X: out ITEM_TYPE) do
                     X:=B(NEXT_OUT);
                     NEXT_OUT:=(NEXT_OUT+1) mod SIZE;
                     COUNT:=COUNT+1;
                end TAKE_OUT;
           or terminate;                                -- terminate alternative
           end select;
      end loop;
end BUFFER;

B:    BUFFER;                                          -- task object declaration
```

Figure 1: An Ada task specification, body, and declaration.

Ada does not require that an implementation follow any particular dispatching policy. In particular, it does not require that the policy be equitable or "fair". To an application programmer, this means that the correctness of a program should not depend on any dispatching policy, such as round-robin time-slicing. To an implementor, it means freedom to choose a dispatching policy based on efficiency.

An Ada programmer may specify a static priority for a task, via the pragma PRIORITY". In this case, if two tasks with different priorities are both ready and waiting for the same physical processor the one with the higher priority must be dispatched[1] . These priorities do have a dynamic aspect: even though a task's inherent priority is static, its dispatching priority is temporarily raised when it is engaged in rendezvous with a higher priority task.

The dispatching function of the tasking supervisor requires it to keep track of the ready tasks. We call the data structure it uses to do this the *ready queue*. The operations on this queue are: (1) insert a task; (2) delete a task (from any position); (3) choose the task with highest priority. There are a number of data structures suitable for such a queue, representing trade-offs of the cost of one operation against the others. Since insertion and deletion will ordinarily be executed in pairs, the theoretically optimum structure would be a heap of the kind used in the Heap Sort. With this structure, insertion and deletion can be done in logarithmic time with respect to the number of ready tasks, and choosing the task with highest priority can be done in constant time. We doubt, however, that the number of ready tasks will be large enough in practice to justify the overhead of this structure, and so our implementation makes use of a circular doubly-linked list, ordered by decreasing priority. With this structure, the only operation that requires more than constant time is priority-ordered insertion, which must traverse any tasks of higher priority than the task being inserted. We assume that an implementation will support a nontrivial range of priorities. If not, there is no need to order the ready queue, and all operations can be done in constant time.

In addition to maintenance of a ready queue, there is another cost factor involved in dispatching. This is *context switching*, the process of transferring control between tasks. In general, any supervisor action may require two context-switches: the first, to transfer control to the supervisor, and the second to transfer back to some Ada task. The actual cost of these context switches is likely to vary greatly from implementation to implementation, depending mostly on the physical processor and the way memory is organized.

Most processors have a set of registers. When a task transfers control to the supervisor, any of these registers whose values may be needed later must be copied to memory. For the most part, this is the same as must be done for procedure calls, and is not necessarily a problem related to tasking. In the

[1] Note that the interpretation of priority must apply to any other "processing resources", besides the physical processor. This may be interpreted to include busses, processor memory, and peripherals.

authors' implementation on the Z8002, context switching actually takes fewer instructions than an Ada procedure call.[2] Unfortunately, for some architectures context switching may involve much more overhead. In particular, features such as a large register file or multiple banks of registers, which are designed to make procedure switching more efficient, appear to make task switching less efficient. As another example, in a hierarchical storage system, switching control to a task that has been waiting may sometimes require copying a portion of the task's code or data from backup memory to processor memory, which could be very time-consuming.

Since an implementation does have control over dispatching policy, subject to specified priorities, if context-switching is expensive it can choose a dispatching policy that reduces this cost. For example, switching to and from the supervisor can possibly be handled as a special case that is implementable more efficiently than switching between tasks. Thus, by giving preference, among equal priority tasks, to the task which last transferred control to the supervisor, the frequency of expensive task-to-task context switches may be reduced.

3 Storage Management

In order to execute, a task requires storage. This may logically be divided into two main parts: storage for code, and storage for data. It is possible to implement tasks so that all tasks of a single type may share one copy of code. It is not possible to do this with data. Each task must therefore be allocated some data storage of its own.

Some of this storage is for data structures used by the tasking supervisor to keep track of the state of the task. The rest of it, which we call the *working storage* of the task, is needed for the local data belonging to the task. The details of how this storage is laid out and how it is managed can become rather complicated, and are beyond the scope of this paper. It is necessary, though, to consider how the basic functions of storage allocation and deallocation relate to the tasking supervisor.

Ada requires an implementation to support allocation of storage in blocks of arbitrary size. In some instances, the lifetimes of data objects permit them to be allocated storage on a stack. In other instances allocation must be from a pool (or pools) managed according to a general purpose scheme. This is certainly true for the working storage of tasks and for objects created by allocators. Some implementations may also choose to use such a scheme for objects whose size is not statically determinable, since it permits more efficient treatment of block statements. The execution times of allocation and deallocation operations in such pools vary randomly according to the current state of memory, and in general are bounded only by the number of free blocks of memory.

[2] This is due to the cost of creating an activation record for a procedure call.

Ada semantics do not in general permit limiting allocation rights in such a pool to a single task. Allocation and deallocation operations therefore require mutual exclusion. One way this may be insured is by treating them as supervisor functions. This is the approach followed by the authors in their implementation. The biggest problem with this is that while these operations are going on no Ada task may execute. The problem might be partially avoided by treating the manager of each storage pool somewhat as if it were an Ada task. That is, requests for allocation/deallocation are treated similarly to entry calls. They cannot be treated exactly like entry calls, since honoring priorities requires that a manager accept requests in order according to the priority of the caller, and execute at the highest priority of all the waiting callers. If this is done, deallocation requests need cause no immediate action, but can be queued for later processing at low priority, if sufficient storage is available to meet all requests. Weaknesses of this approach are the extra execution time involved in enqueueing and dequeueing requests, and greater complexity in the dispatching and storage management algorithms.

As mentioned above, this form of storage allocation is required for task creation and for evaluation of an allocator. It also may happen as a result of elaboration of a declaration of an object or evaluation of an expression whose result size is not statically determinable. Deallocation may occur as a result of explicit use of UNCHECKED_DEALLOCATION, or implicitly, as determined by the implementation. The times at which deallocation is safe for declared task objects are discussed in the next section; typically they are associated with scope exits. For objects created by allocators, the first time deallocation can be guaranteed to be safe in general is on exit from the scope of the corresponding access type declaration. For declared objects, it is on exit from the scope of the object declaration. Thus, though an implementation has considerable freedom in determining when, or whether at all, to perform implicit deallocation, bursts of this activity are likely to occur upon exit from an environment with such declarations.

4 Transitions in the Lifetime of a Task

The semantics of Ada define a number of phases in the life of a task. The transitions between these phases mostly require some action by the tasking supervisor.

Creation of a task is the transition by which a task object comes to exist, and after which reference may be made to the task. This happens as a result of elaboration of an object declaration or the evaluation of an allocator, for a task type or a composite type containing a task component. Execution of the task does not begin until later, however. We call the task whose thread of control created a task its *creator*.

Task creation minimally requires allocation of some storage for the task, and

initialization of most of the data structures used by the tasking supervisor that are specific to the task. It is likely that this will involve the supervisor, at least for allocation of storage to the task.

The *start of activation* is the transition by which a task begins execution. At this point it begins the elaboration of the declarations within its declarative part, setting up the local environment in which it will later execute its sequence of statements. From this point on until the elaboration of its declarative part has been completed, the task is said to be *activating*. In the case of task objects created through the elaboration of an object declaration, activation starts immediately before the creator begins to execute the sequence of statements of the declarative unit where the declaration appears. In the case of task objects created though the action of an allocator, activation starts immediately before the creator completes evaluation of the allocator.

At the start of activation of a collection of tasks the supervisor inserts the tasks to begin activation into the ready queue, and blocks the creator.

The *end of activation* is the transition by which a task signals to its creator that it need no longer wait for that task to finish activation. It takes place between the time a task finishes elaborating its declarative part and the time the task begins executing its sequence of statements. [3] After this, the task is no longer *activating*, but *active*.

The end of activation requires a supervisor action, since the supervisor may need to unblock the creator task, or at least record that it is waiting for one less dependent task to finish activation. If the creator becomes unblocked and has a higher priority than the task finishing activation, it may also be necessary to preempt the processor for the creator.

Completion is the transition by which a task reaches the end of its execution. This may happen as a result of reaching the end of its sequence of statements, or by other means, such as an exception or an abort statement.

Even though a completed task cannot execute any more, it is not yet safe to deallocate its working storage at this point, since other tasks that are allowed to access this storage may still be executing. Nevertheless, completion of a task requires action by the supervisor. The task must be removed from any queues on which it may happen to be, and must be marked as completed. A check must be made for pending calls on entries of the completed task, and the exception TASKING_ERROR must be raised in any such calling tasks. Dependent tasks may need to be terminated.

The supervisor action for task completion would ordinarily be initiated by request of the completing task, upon reaching the end of its own sequence of statements. It might also take place as a side-effect of a supervisor action initiated by another task, as in the case of an exception or an abort.

Termination is the transition by which a task becomes *terminated*. This

[3] A task cannot end its own activation until the end of activation of each local task created by the elaboration of that task's declarative part.

must be distinguished from completion because of the "terminate alternative" feature of the select statement (to be discussed at greater length below). This generally happens as a result of some other event, and does not require a separate supervisor request.

Termination is the first point at which the working storage of a task may safely be deallocated. However, at least some of the data structures associated with a task must be maintained, because reference may still be made to the task. In particular, it is possible for other tasks to still attempt entry calls to a terminated task, to abort it, and to interrogate it's status via the 'TERMINATED and 'CALLABLE attributes. For simplicity, an implementation may therefore choose to defer deallocation of the task's working storage until later, when all the data structures associated with it can be deallocated.

In general, the earliest point at which it is completely safe for an implementation to discard *all* storage associated with a task is when execution leaves the scope of the task's type declaration. This is so because reference to a task may be passed far from its point of creation, as via task access variables and functions returning task values.

In the authors' implementation, all the storage specifically associated with a task is deallocated somewhat earlier than this. For tasks created via allocators, we deallocate the storage of all tasks accessible via a given access type at the time execution leaves the scope of the access type. For tasks created by task object declarations, we deallocate the storage at the time execution leaves the scope of the object declaration. The only trickiness is that a function may return the value of a local task object (passing it outside the scope of the object declaration). Fortunately, such a task must be terminated, and the only use that Ada permits of such a task value (as opposed to a task access value) is to apply the 'TERMINATED and 'CALLABLE attributes, which will always be false. We therefore maintain a single dummy task descriptor, which is terminated, and use this for local task values returned by functions.

Of course, it is desirable to deallocate the storage associated with a task as early as possible. For an implementation in which all references to tasks are indirect, through a common table, the strategem described above for functions returning local tasks can be applied more generally. By replacing the table entry for a terminated task by a reference to a single dummy task descriptor, it should be possible to recover most of the storage associated with a task at the time of termination. The exception is the table entry of the task, itself. It is not in general safe to reassign that until exit from the scope of the task.

5 Time Delays

An executing Ada task may request to be delayed for a specified real time duration. This may be by a simple delay statement, or as part of a timed entry call or selective wait statement. Since an implementation can have only a fixed

number of hardware timers available, and since the number of tasks that may request delays is unbounded, it is necessary to maintain a data structure to keep track of the tasks waiting on delays, and the times their delays are due to expire. We call this the *delay queue*. Its usage very much parallels that of the ready queue, discussed above. Tasks are entered in the queue when they request a delay, and removed from the queue when the delay expires, or is cancelled. Since the supervisor must process wake-up requests in order of increasing wake-up time, it helps to use a data structure ordered by increasing wake-up time. The logical choices of data structure are the same as for the ready queue, so we do not discuss them further.

6 Entry Calls

Synchronization of tasks is by means of *rendezvous*. A rendezvous is requested by one task making an entry call on an entry of another task. For the rendezvous to take place the called task must accept this entry call. During the rendezvous, the calling task waits, while the accepting task executes. When the accepting task ends the rendezvous, both tasks are freed to continue their execution.

Achieving rendezvous ordinarily requires that one of the two tasks wait until the other is ready. In the case that more than one task is waiting on the same entry of a task, Ada requires the calls be accepted in first-in-first-out order. An implementation must therefore maintain data structures to keep track of which tasks are waiting on entry calls, which entries they are calling, and in what order the calls on each entry of a task arrived. We call such structures *entry queues*.

In order to make an entry call, a task must uniquely specify both the task and the entry it wishes to call. It does this by executing an entry call statement. These come in three forms: (1) the unconditional entry call; (2) the conditional entry call; (3) the timed entry call. (See Figure 2.)

To a task executing an unconditional entry call, the call is much like a procedure call. Like a procedure call, it may have parameters, which permit values to be passed in both directions between the calling and accepting tasks. The calling task is blocked until completion of the requested rendezvous. If the call is completed normally, it resumes execution with the statement following the call, just as it would after return from a procedure call. Recovery from any exception raised by the call is also treated as it would be for a procedure call. One minor difference detectable by the calling task is that an entry call may result in TASKING_ERROR being raised in the calling task, whereas an ordinary procedure call would not.

To implement an entry call, however, requires supervisor action. The supervisor must block the calling task. It must also insert the calling task into an appropriate entry queue, or, if the called task is waiting to accept, release the accepting task to begin executing the rendezvous. In both cases, control is

```
TASK_NAME.ENTRY_NAME(PARAMETER);          -- a simple entry call

select TASK_NAME.ENTRY_NAME(PARAMETER);
                                                  -- a conditional entry call
      . . .          -- a sequence of statements to be executed after a rendezvous
else                                                              -- else part
      . . . -- a sequence of statements to be executed if no rendezvous is made
end select;

select TASK_NAME.ENTRY_NAME(PARAMETER);    -- a timed entry call
      . . .          -- a sequence of statements to be executed after a rendezvous
or delay DELAY_DURATION;
      . . . -- a sequence of statements to be executed if no rendezvous is made
end select;
```

Figure 2: The forms of entry call.

ordinarily transferred from the calling task to another task.

A conditional entry call differs from an unconditional entry call in that the calling task need not wait unless the call can be accepted immediately. If the called task is ready to accept, execution proceeds as for an unconditional call. Otherwise, the calling task resumes execution without completion of a rendezvous. The syntax provides for execution to resume at different places, depending on whether any rendezvous took place.

Implementing the conditional entry call efficiently requires a simple test for whether the called task is ready to accept. This can be done in constant time if the supervisor maintains an *accept vector* for each task, telling on which entries, if any, the task is ready to accept a call. (See the discussion of selective wait, below.) If the test fails, the supervisor may return control immediately to the calling task. Otherwise, the actions are similar to those for the unconditional call.

A timed entry call allows the task that executes it to make an entry call with provision that it be awakened, and the call cancelled, if the call is not accepted before the expiration of a specified delay. As with the conditional entry call, provision is made for execution to resume in different places, depending on whether a rendezvous takes place.

In addition to the processing required for a normal entry call, the timed entry call requires scheduling of a wake-up event if the call cannot be accepted immediately. If the call is accepted before this delay expires, the calling task

```
accept ENTRY_NAME(PARAMETER: PARAMETER_TYPE) do
        . . .           -- a sequence of statements to execute during rendezvous
end ENTRY_NAME;
```

Figure 3: A simple accept statement.

must be removed from the delay queue. If the delay expires first, the task must be removed from the entry queue.

Ada also allows declaration of *families* of entries. These may be understood as arrays of entries. They present no special implementation problems, but may contribute to the cost of some operations by effectively increasing the number of distinct entries. (See the discussion of selective wait, below.)

7 Accepts

The accepting task sees a rendezvous much differently from the calling task. It has no direct knowledge of the origin of the call. It is forced to accept calls on each entry in their order of arrival. What it can control is when it will accept an entry call, and on which entries it will accept calls at a given time. It does this primarily by means of the several forms of accept statement. These are: (1) the simple accept statement (see Figure 3); (2) the selective wait with only accept alternatives; (3) the selective wait with else part; (4) the selective wait with delay alternative(s); (5) the selective wait with terminate alternative (see Figure 1).

A simple accept statement specifies a particular entry on which the task is prepared to accept a call, together with a sequence of statements to be performed during the rendezvous. The entry may have parameters, to be passed between the tasks during the rendezvous.

When an accept statement is executed, the implementation must check whether there are any calls queued for the specified entry. If there are, a rendezvous is begun with the task whose call has been waiting longest. Beginning the rendezvous means that the accepting task begins to execute the sequence of statements inside the body of the accept statement.

The cost of checking whether there are any calls queued for a given entry depends on the data structure chosen for the entry queue(s). There are at least two reasonable alternatives:
(1) A separate queue may be kept for each entry of each task. If the queue is nonempty, the next caller to be served is at the head of the queue. The authors' implementation does this, making use of circular doubly linked lists, so that checking, insertion, and deletion are all constant-time operations.

(2) All the pending entry calls to each task may be kept in a single queue associated with that task. In this case, checking time is proportional to the number of pending calls queued for that task. This alternative is reasonable if we do not expect very many calls ever to be queued for one task.

If the entry has parameters, provision must be made, before the accepting task begins executing the rendezvous, for it to access the parameters. This may be done by copying their values or their addresses into the working storage of the accepting task. This can be done quickly on a single processor if the parameters are stored in a contiguous block, since all that need be passed is the base address of this block.

When an accepting task has completed the sequence of statements inside the accept statement, the rendezvous must be ended. This involves releasing the calling task, so that it may resume execution. It would ordinarily also involve reinsertion of either the accepting or the calling task into the ready queue. If the entry has parameters, the values of parameters modified by the accepting tasks may need to be copied back to the calling task.

If there are no tasks waiting on entry calls to the specified entry at the time the accept statement is executed, the accepting task must wait until one arrives. This means the supervisor ordinarily must choose another task to execute.

In the case that a task is willing to accept calls on several entries, it may use the selective wait statement. (See Figure 1.) This statement has several forms. The simplest form has only *accept alternatives*. Each of these is an accept statement followed by an optional sequence of statements. The accept alternatives may be guarded by boolean conditions, which are evaluated at the beginning of the selective wait. Any accept alternatives without guards or whose guards evaluate to true are considered *open*. At least one alternative must be open, or else the exception PROGRAM_ ERROR is raised. If there are tasks queued on any entry for which there is an open accept alternative, one such alternative is arbitrarily chosen, and execution proceeds with that alternative. If there are no tasks queued on open alternatives, the task executing the selective wait is blocked until some task makes a call on one of the open entries. When such a call is made, the waiting task resumes execution with a corresponding open accept alternative.

The special implementation problem introduced by the selective wait is that a task may at one instant be ready to accept a call on a set of *several* entries. From the viewpoint of the supervisor, this is really two problems, since it comes up in the processing of entry calls, as well as selective waits:
(1) Since a task may be waiting on more than one open accept alternative, processing an entry call requires checking whether the called entry corresponds to one of the open alternatives.
(2) Since there may be several open accept alternatives, processing the selective wait requires checking the set of pending entry calls against the set of open accept alternatives.
The need to be able to perform both of these operations efficiently strongly

influences an implementation's choice of data structures.

There are two obvious ways to perform the first operation, checking whether a called entry has a currently open accept alternative:
(1.1) If the set of open accept alternatives is represented as a list, checking requires comparing the called entry against each of the entries in this list. We call this approach the use of an *open entry list*. It may be time consuming if there are many open entries.
(1.2) An alternative is to use a vector representation for the set of open entries. Such an *accept vector* would have one component for each entry of the task. Each component would minimally indicate whether the corresponding entry is open. One fault of such a vector is that it must be cleared at the beginning of each rendezvous. Most machines can zero a contiguous block of memory very efficiently, so this is unlikely to noticeably affect execution time, except in the case that a task has many entries. In our implemention the accept vector also contains the address of the accept alternative for each open entry. This simplifies processing of rendezvous, but there may be a benefit to storing these addresses separately, since they do not need to be cleared.

There are also several ways of looking for queued calls on one of the currently open accept alternatives:
(2.1) With separate queues for each entry, it is necessary to check the queue corresponding to each open entry. This requires sequencing through the open entries. The way the authors do this is to sequence through all the entries of the task, checking the accept vector entry of each, to see if it is open. Alternatively, if the open entries are represented by an open entry list, this check can be performed more quickly, without looking at the non-open entries. This may be a good reason to keep both an open entry list and an accept vector, though this redundancy may cost more in overhead than it saves through faster execution of the check for pending calls.
(2.2) With one queue per task, the open entry list is very inefficient, since checking takes time proportional to the product of the lengths of the queue and the open entry list. We therefore consider this combination impractical. On the other hand, with an entry vector, it is possible to sequence through the queue, checking the accept vector value for the entry corresponding to each call. The time this takes depends on the number of entries queued for the task. This might be a problem if the task is designed to serve some entries with high priority, while letting many calls queue up on others.

Note that the accept vector or open entry list must be created at the time the selective wait statement is executed, once it is known which alternatives are open. The time needed to do this only depends on the number of alternatives in the selective wait statement.

Members of entry families may be treated as full entries, in which case they will contribute to the size of accept vectors and the number of entry queues. This is the approach followed in the authors' implementation. Alternatively, members of entry families may be treated as a special case. If so, several hybrids

of the general solutions described above are possible. For example, each entry family might be represented by a single item in the accept vector. This would shorten the length of the accept vector, and therefore reduce the cost of clearing it at the beginning of rendezvous. However, it would then take more time to check for a possible rendezvous when the accept vector showed a member of an entry family to be open, since an open entry list would have to be used for members of the family. Another possibility is sharing a single entry queue for all the members of an entry family. Such alternatives may be justified if it is believed that entry families will typically be large, and the number of pending calls and open alternatives for a family will typically be small. There does not seem to be any reason to believe this, however. Moreover, such hybrids add logical complexity to the tasking supervisor.

A selective wait may also include an *else clause*. The difference this makes is that if there are no pending entry calls for any of the open accept alternatives, the task executing the selective wait does not wait, but proceeds by executing the else clause. This poses no special implementation problems, but since a task that is not willing to wait is likely to be time-critical, it is especially important that the check for pending entry calls (discussed above) be performed as quickly as possible.

A selective wait statement may instead include one or more *delay alternatives*. A delay alternative consists of a delay statement, followed by an optional sequence of statements. The effect is to limit the amount of time the task executing the selective wait is allowed to wait for an entry call. If the delay expires before a rendezvous has begun, the selective wait is cancelled, and the task resumes execution with the sequence of statements of the corresponding delay alternative.

The delay alternative poses no special implementation problems. However, in addition to the work involved in processing a selective wait, if no rendezvous is immediately possible, it involves scheduling a wake-up event. Even though there may be more than one delay alternative, only the one of these with the shortest delay need be processed.

Finally, a selective wait statement may include a *terminate alternative*. (See Figure 1.) The correct implementation of the terminate alternative imposes special demands on the tasking supervisor. It adds execution cost to the selective wait statement that is in the worst case proportional to the number of surrounding levels of nested tasks. We explain why this is so in the next section, and present a simplified version of the termination algorithm used in our implementation[BR83a].

8 Termination

The Ada language definition describes the semantics of the terminate alternative in terms of *masters* of tasks. Each task has one or more masters on which it

depends. Each master corresponds to a library package, a task, or an activation (execution) of a block statement or subprogram.

The *direct master* of a task is the library package, task, or activation of the block statement or subprogram which elaborated: (a) the object declaration which created the task, or (b) the access type declaration for the type returned by the allocator whose evaluation created the task. That is, a task is direct master of tasks only as a result of object or type declarations appearing in the declarative part of the corresponding task body. The analogous statement is true for library packages and for activations of subprograms and block statements. A task is said to be *directly dependent* on its direct master.

A task may be further (indirectly) dependent on a hierarchy of masters, determined according to the dynamic sequence of control that led to elaboration of the task object or access type declaration:
(1) If the task is dependent on a block statement, and the block statement is "executed by another master"[Ada83], then the task is also dependent on this master.
(2) If the master of a task is a subprogram called by another master the task is also dependent on this master.
(3) If the master of a task is a task that depends on another master the task is also dependent on this master. This is illustrated in Figure 4. There, procedure P executes block B, which calls procedure Q. While procedure Q is active, the masters above Q are B and P (in that order). Task T2 is directly dependent on that activation of Q, and indirectly dependent on the activations of B and P. Note, however, that the task accessed by access variable A1, in block B, is directly dependent on procedure P, where the access type A is declared.

Execution may not leave a master until all of its dependent tasks have terminated. This means that if the master is a task, it may not terminate; if it is a subprogram activation, control may not return from the call; if it is a block statement, control may not leave the block statement until all local tasks are terminated.

The definition of when a task may terminate makes use of the concept of *completion* of a master. This is defined as follows:
(1) A task or activation of a subprogram or block statement is completed when execution has reached the end of the corresponding sequence of statements.
(2) An activation of a block statement is completed when execution should leave the block, due to an exit, goto, or return statement.
(3) An activation of a subprogram is completed when control is transferred out of the subprogram, due to a return statement.
(4) A task or activation of a subprogram or block statement is completed when "an exception is raised by the execution of its sequence of statements if there is no corresponding handler, or, if there is one, when it has finished the execution of the corresponding handler"[Ada83].
Note that a library package is never completed.

Let a task be termed *asleep* if the task and all of its dependent tasks are

```
procedure P is                                  -- direct master of T1,A1.all
          task type T;
          type A is access T;
          T1: T;
          task body T is
          begin loop null;
                end loop;
          end T;
          procedure Q is                            -- direct master of T2
                T2: T;
          begin null; end Q;
begin
B:        declare                                 -- direct master of T3,T4
                T3,T4: T;
                A1: A:= new T;                        -- task access variable
          begin
                Q;
          end B;
end P;
```

Figure 4: Masters of tasks.

either terminated or waiting on an open terminate alternative. A task that is not asleep is *awake*. Note that if a task is asleep then all of its dependents are also asleep.

The Ada language definition divides the criteria for when an activated task should be terminated into several cases:
(1) A task T that has no dependent tasks is terminated when execution of T is completed.
(2) A task T that has dependent tasks is terminated when execution of T is completed and all tasks dependent on T are terminated.
(3) A task T that is waiting on an open terminate alternative is terminated when for some master M of T every dependent task of M is asleep.

It is possible to keep track of which tasks are asleep by associating an *awake count* with each master. The implementation must arrange to decrement and increment this count as tasks go to sleep and wake up, so that it always reflects the number of direct dependents of the master that are awake. Thus, the awake count of a master becomes zero when all tasks dependent on the master are asleep.

The definition of awake counts treats tasks as members of families. The *children* of a task T are all the tasks dependent on T that are not dependent on any other task that is dependent on T. A task is called the *parent* of the tasks that are its children. Thus, in the hierarchy of masters, there is no other task between a task and its children. Note that some tasks, such as those directly dependent on a library unit or main program, have no parent.

For any task T and master M define:

$$IS\_AWAKE(T) = \begin{cases} 0 & \text{if } T \text{ is terminated} \\ & \text{or is waiting on a terminate alternative} \\ & \text{and} \\ & CHILDREN\_AWAKE(T) = 0; \\ 1 & \text{otherwise.} \end{cases}$$

$CHILDREN\_AWAKE(T) = IS\_AWAKE(T_1) + \ldots + IS\_AWAKE(T_n)$,
where $T_1, \ldots, T_n$ are all the children of T.

$AWAKE(M) = IS\_AWAKE(T_1) + \ldots + IS\_AWAKE(T_n)$,
where $T_1, \ldots, T_n$ are all the dependent tasks of M.

Note that a terminated task has CHILDREN_AWAKE equal to zero, since all of its dependents must already be terminated. Any other task for which CHILDREN_AWAKE is zero must have all its descendants asleep. Thus IS_AWAKE(T) is zero if and only if T is asleep.

The awake count of a master can be used to check the requirements for termination. The rule to be followed is *whenever the awake count of a master is zero and the master is completed, terminate all dependent tasks of the master.* If a task has no dependents, it can be terminated immediately upon completion, since its awake count will be zero. If a task with dependents completes execution, and all dependents are terminated, its awake count will also be zero, so it can

terminate immediately. If a task is waiting on an open terminate alternative, and there is some master whose execution is completed and such that every task depending on the master is asleep, the awake count of the master will be zero and the task can be terminated. Conversely, whenever the awake count of a master is zero, all the dependent tasks of the master must be asleep. It is thus safe to terminate them all.

A key observation is that in the hierarchy of masters below any task T there is always exactly one master that is currently being executed by T. This corresponds to T itself, or to a subprogram call or block statement being executed by T. None of the masters between this *current submaster* of T and the next master above T can possibly complete before control leaves the current submaster. We can therefore ignore all masters between any task and its current submaster, when checking for terminable tasks.

To keep the awake counts up to date, whenever a task T begins to wait on an open terminate alternative, the supervisor must check whether the value of CHILDREN_AWAKE for T is zero. If so, T has just gone to sleep. The supervisor must then decrement the awake count of the direct master of T. It must also decrement CHILDREN_AWAKE of T's parent, if any. In the case that T's parent is waiting on a terminate alternative and its CHILDREN_AWAKE goes to zero, T's parent goes to sleep, and the process described in this paragraph must be repeated, with T's parent taking the role of T. This update process should continue up through the hierarchy of parents until a task is found that is awake or has no parent.

If in the course of this upward propagation, a master is found that not only has zero awake count, but is also completed, all the dependents of that master should be terminated. In fact, it is enough to do this for the highest such master, in the case that there is more than one.

Similarly, when a rendezvous involving a task that has been waiting on an open accept alternative begins, if the task was asleep the awake count of the direct master of that task must be incremented, as should the value of CHILDREN_AWAKE for its parent, and so on up through the hierarchy of parents.

Note that tasks dependent on a master that is not complete may still be terminated as a result of completion of a higher level master. For this to happen, the task containing the uncompleted master must be waiting on a terminate alternative. Since selective wait statements may only appear within a task body, and not within any more deeply nested program unit, the uncompleted master must be either the task body or a block statement within. In this case, however, if there is a higher level master (above the task body) which is completed, and all the dependents of this master are asleep, the entire collection may be terminated. For this reason, it is necessary to carry the updating of awake counts up the hierarchy of masters, even past uncompleted masters.

An implementation need not maintain awake counts for trivial masters, that have no local task object declarations or task access type declarations. This

means that the actual overhead of maintaining awake counts is proportional to the number of tasks that are masters over the task containing the terminate alternative. Note, however, that this nesting of tasks is dynamic, and not generally determinable from the static structure of a program.

In addition to requiring action whenever a select statement with an open terminate alternative is executed, proper task termination also requires action upon the completion of every nontrivial task master. At this time, it may be necessary to force the task completing the master to wait, pending the termination of local tasks. This may involve updating awake counts, which may in turn cause the termination of a collection of tasks. As mentioned above, termination is likely to be the occasion for deallocation of a task's storage. In the case that the master is itself a task, completion also involves checking for pending entry calls, and raising the exception TASKING_ERROR in any tasks with such pending calls. The execution time for all this may be high. It is likely to be higher for tasks with many entries, and for tasks that are members of hierarchies.

Note that library packages and main programs are special cases of task masters. The main program is treated much as if it were a task. The "termination" of the main program "task" does not wait for termination of tasks that depend on library packages, since it is not their master. Ada does not specify what happens to any unterminated tasks that depend on library packages when a main program terminates. Thus an implementation may allow such tasks to continue, or treat this situation as an error.

Since the costs of completion of a task or master, abort statements, and select statements with open terminate alternatives all are affected by the number of levels of task nesting, programmers interested in execution speed may be wise to avoid building complex hierarchies of tasks. In the event that building such hierarchies is necessary, care should at least be taken to avoid unnecessary task completions and terminations.

9 Other Tasking Features

The abort statement permits a task to cause the abnormal completion of a specified set of tasks, along with any other tasks dependent on them. This requires that the task executing the abort statement initiate action by the tasking supervisor. The supervisor must examine the state of each of the tasks to be aborted, and mark them as "abnormal". In addition to being marked as abnormal, some of these tasks may need to be completed, and possibly also terminated, immediately. These are the tasks waiting on an accept, select, or delay statement, waiting on an entry call and not yet in a rendezvous, or not yet started activation. This must be done before control is returned to the task executing the abort.

As mentioned above, the "cleanup" involved in task completion and termina-

tion may be complex and time consuming, since it includes storage deallocation, checking for pending entry calls, and propagating awake count information. The cost of an abort will therefore be higher than most tasking operations, but not much higher than an equivalent number of normal completions. It will be spread out somewhat more randomly, since aborted tasks in rendezvous or between synchronization points may not be completed until they end the rendezvous or reach their next synchronization point. During the time the supervisor is doing this cleanup work, it would be difficult for an implementation to permit other Ada tasks to execute. For this reason, executing an abort statement may wreak havoc with the execution timing of other tasks on the same processor.

The 'CALLABLE and 'TERMINATED attributes, by which a task may obtain information about the status of another task, do not necessarily require action by the supervisor, since they do not require modification of supervisor data structures. They do require reading such structures, however, and so an implementation may treat them as supervisor requests, in order to keep a clean interface between the supervisor and Ada tasks.

Calculating the 'COUNT attribute of an entry is most sensibly done by counting the number of pending calls on the entry in the associated queue. In this case, it will take time proportional to the number of calls in the queue, which is the value of 'COUNT if a separate queue is used for each entry. It is conceivable that an implementation might keep an up-to-date count for each entry, so that fetching this value would take constant time. This seems unwise since the 'COUNT attribute is not likely to be used often[4], and the overhead of keeping such counts for every entry would therefore not be justified.

The implementation of Ada tasking interacts heavily with the implementation of exceptions. An exception may be propagated from the accepting task to the calling task at the end of a rendezvous. This can be done when the supervisor releases the calling task, by examining and copying the relevant state information from the accepting to the calling task. There is no reason for it to add more than a small constant to the execution time of this supervisor function. The supervisor also must check for certain conditions that require an exception to be raised in a task. Most of these checks, such as the check performed at the time of an entry call to make sure that the called task is callable, add a small constant to the execution time of the associated supervisor function. One,the checking of entry queues for pending calls at the time of task completion, may have execution time dependent on the number of entries of the task, as discussed above.

[4]The value returned by 'COUNT is not reliable, since new entry calls may be made and others cancelled, between the time this value is fetched and the time it is used.

10 Conclusions

It has been shown that Ada tasking can be implemented using fairly simple and efficient techniques, for a single processor with conventional architecture. Simple and conditional entry calls can be implemented so as to take constant time, as can simple accept statements. The only operation involved in simple rendezvous that cannot be done in constant time occurs at the end of rendezvous. This is inserting the calling task into the ready queue. Selective wait statements can be implemented in time that is proportional to the number of accept alternatives, plus possibly the time required to zero an accept vector, which is proportional to the number of entries in the accepting task. Statements involving delays take more execution time, due to delay queue insertion. Selective waits with open terminate alternatives may also require more time, especially if they occur in a hierarchy of dependent tasks. Task creation, activation, completion, termination and abortion, as well as storage allocation and deallocation, have costs that depend on a number of factors, and may be high.

The authors believe that tasking is one of the simpler, cleaner, and more efficiently implementable features of Ada. The tasking supervisor of the FSU Ada cross-compiler, which makes use of techniques described in this paper, was coded and debugged without need for any design changes. The same cannot be said for the implementations of a number of other features, including the scoping rules for basic operations on access types and limited and private types, the runtime checks, and functions returning unconstrained composite values. The small size and simplicity of the supervisor code actually came as a suprise, after considerable initial prejudice against the supposed inefficiency of the Ada tasking model.

Tasking is one of the strengths of Ada. In using it, some thought should be given to the costs of the more complex operations, but there appears to be no justification for refraining from using it for fear of inefficiency.

References

[Ada79] "Preliminary Ada Reference Manual", *SIGPLAN Notices* 14.6 ACM (June 1979).

[Ada80] *Reference Manual for the Ada Programming Language*, proposed standard document, United States Department of Defense (July 1980).

[Ada82] *Reference manual for the Ada programming language*, draft revised MIL-STD-1815, draft proposed ANSI standard document, United States Department of Defense (July 1982).

[Ada83] *Reference Manual for the Ada Programming Language*, ANSI/military standard MIL-STD-1815A, United States Department of Defense (January 1983).

[BR83a] Baker, T.P., and G.A. Riccardi, "A runtime supervisor to support Ada tasking, Part 1: activation, execution and termination," FSU Ada Project report 83-6 (May 1983).

[BR83b] Baker, T.P., and G.A. Riccardi, "A runtime supervisor to support Ada tasking, Part 2: rendezvous and delays," FSU Ada Project report 83-7 (May 1983).

[BR84] Baker, T.P., and G.A. Riccardi, "A runtime supervisor to support Ada task activation, execution, and termination," *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota (October 1984)14-22.

[BBH80] Belz, F.C., E.K. Blum, and D. Heimbigner, "A multiprocessing implementation-oriented formal definition of Ada in SEMANOL," *Proceedings of the ACM-SIGPLAN Conference on the Ada Programming Language*, Boston, ACM (November 1980) 202-212.

[Cl82] Clemmensen, G.B., "A formal model of distributed Ada tasking," *Proceedings of the AdaTEC Conference on Ada*, Arlington, Virginia, ACM (October 1982) 224-237.

[Co84] Cornhill, D., "Four approaches to partitioning Ada prograrms for execution on distributed targets," *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota (October 1984)153-162.

[Fa82] Falis, E., "Design and implementation in Ada of a runtime task supervisor," *Proceedings of the AdaTEC Conference on the Ada Programming Language*, Arlington, Virginia, ACM (October 1982) 1-9.

[HN80] Haberman, A.N. and I.R. Nassi, "Efficient implementation of Ada tasks," technical report, Department of Computer Science, Carnegie-Melon University (January 1980).

[HPT81] Hartig, H., A. Pfitzmann, and L. Treff, "Task state transitions in Ada," *Ada Letters* 1.1 (July-August 1981) 31-42.

[Hi82] Hilfinger, P.N., "Implementation strategies for Ada tasking idioms," *Proceedings of the AdaTEC Conference on the Ada Programming Language*, Arlington, Virginia, ACM (October 1982) 26-30.

[JA82] Jones, A. and A. Ardo, "Comparative efficiency of different implementations of the Ada rendezvous," *Proceeedings of the AdaTEC Conference on the Ada Programming Language*, Arlington, Virginia, ACM (October 1982) 212-223.

[LB80] Lovengreen, H.H. and D. Bjorner, "On a formal model of the tasking concept in Ada," *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programmin Language*, Boston, ACM (November 1980) 166-175.

[St80] Stevenson, D.R., "Algorithms for translating Ada multitasking," *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, Boston, ACM (November 1980) 166-175.

[We84] Weatherly, R.M. "A message-based kernel to support Ada tasking", *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota (October 1984) 136-144.

on dist. systems, Vancouver.