An Implementation of Ada Tasking

Thomas D. Newton
October 1987
CMU-CS-87-169

Abstract

This paper describes the tasking implementation in the VAX/Mach version of the Spice Ada compiler. The implementation is built on the primitives supplied by the C Threads package, and uses run-time system routines to reduce the size of the inline code generated to handle various tasking constructs. The code that the compiler generates for various constructs and the operation of several key run-time system routines are detailed at length. Average times to run several test programs on a uniprocessor and a multiprocessor under both coroutine and Mach threads implementations of C Threads are presented.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-84-K-1520 and monitored by the Air Force Avionics Laboratory.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

List of Figures

List of Tables

1. Introduction

One of the more notable features which distinguishes the Ada[1] programming language from Pascal and similar languages is its support for concurrent programming in the form of *tasking*. Like Mesa and CHILL, Ada allows programs to have more than one thread of control and provides tools for synchronizing the activities of different threads. In Ada, threads of control are called *tasks*, and the rules concerning their use emphasize convenience and safety. While this support for concurrent programming can be quite handy, it also complicates the job of the compiler, as compiled tasking programs must in general perform a large amount of synchronization to comply with Ada rules. This paper attempts to describe how the VAX/Mach version of the Ada+ compiler implements Ada tasking. After an overview of Ada's tasking constructs and an overview of the Ada+ compiler, the main part of the paper will discuss various implementation details. Following this, the results of some performance tests will be presented to close the paper.

2. Overview of Ada Tasking Constructs

The basic parallel programming construct in Ada is the task. Tasks are executable objects which act as if each was running sequentially on its own logical processor. Every task is an instantiation of a possibly anonymous *task type*. The specification of a task type introduces its name (or the name of the unique task object) and declares the *entries* that tasks of that type provide to synchronize with other tasks; its body contains code that these tasks will execute. Task types may be declared in roughly the same places as subprograms and packages; an important implication of Ada's visibility rules is that global and "intermediate" (local to a block, procedure, function, or task) variables may be read and updated by several tasks. Figure 2-1 shows a one-of-a-kind task (`CONSUMER`), a task type (`PRODUCER`), and two task variables (`P1` and `P2`) declared inside a procedure.

Although Ada seems to assume shared memory, synchronization and communication is generally accomplished by *rendezvous* rather than by modifying variables. *Entry calls*, which look syntactically like procedure calls, allow one task to attempt to rendezvous with another. Each entry call names a particular task, and must correspond to one of the *entry declarations* for the task's type. Entry declarations look much like procedure declarations, but may appear only inside task specifications. However, there are important differences between entries and procedures, and between entry calls and procedure calls. When a task calls a procedure, it is always possible to execute the procedure body immediately. Furthermore, that body never changes, and several tasks may use it simultaneously. When a task calls an entry, it may be suspended for an indefinite amount of time because the availability of a "body" for the called entry in the form of an *accept statement* or *select statement* is under the control of the called task. A rendezvous takes place when a task is willing to accept a call on one of its entries and there is at least one caller. During a rendezvous, the callee executes the body of the accept statement (select alternative) while the caller remains suspended, after which both go their separate ways. Note that both tasks trying to call "unready" entries and tasks trying to accept "unwanted" entries are suspended until a partner is available. If several tasks are waiting to call an entry, subsequent accepts service them in FIFO order of their "arrival".

Several extensions to this basic model make it more flexible and usable. To avoid indefinite waiting,

---

[1]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

Figure 2-1: Basic Tasking Features

```
procedure FEATURES is
    task CONSUMER is
        entry CALL_ME(in out X: INTEGER);
    end CONSUMER;

    task type PRODUCER is
    end PRODUCER;

    P1, P2: PRODUCER;
    SHARED: INTEGER := 0;

    task body CONSUMER is
    begin
        for i in 1..2 loop
            accept CALL_ME(X: in out INTEGER) do
                X := X + 1;
            end CALL_ME;
        end loop;
    end CONSUMER;

    task body PRODUCER is
    begin
        CONSUMER.CALL_ME(SHARED);
    end PRODUCER;
begin
null;
end FEATURES;
```

entry calls may be *conditional* or *timed*, supplying statements that will be executed in place of a rendezvous if a rendezvous cannot begin immediately or within the specified amount of time. Select statements allow a task to accept calls on any of several entries and to make the offer to accept particular ones conditional on run-time conditions. This is useful for writing server tasks to serialize a mix of operations, such as a bounded buffer task which accepts reads when its buffer is not empty and writes when its buffer is not full. Select statements may also be conditional or timed in the manner of entry calls. Finally, select statements may contain *terminate alternatives*, which allow tasks to terminate gracefully if all their potential callers are terminated or have made similar offers.

Because programs do not always work correctly, Ada's tasking features make use of its exception mechanism. If a parent task launches a group of tasks and any of them raise exceptions before completing their activation, an exception must be raised in the parent. If a task tries to call an entry belonging to a task which has completed execution, it reaps an exception rather than a rendezvous. Exceptions that propagate out of an accept body propagate into both tasks involved in the rendezvous. Additionally, Ada's rules about task and master termination have important implications for the implementation of exceptions. Before propagating exceptions out of constructs on which tasks depend, an exception-handling mechanism must wait for those dependent tasks to finish their execution.

Some miscellaneous features round out Ada's tasking facilities. *Abort statements* allow one task to kill another and are only meant for use in the most extreme circumstances. In general, aborted tasks are not

required to die immediately but must do so by their next "synchronization point" (task activation, accept statement, entry call, etc.). Tasks may be assigned fixed *priorities* with the guarantee that low-priority tasks will never be executed if the processor (or processors) could be executing higher-priority tasks. When used on systems that implement them, *representation clauses* allow entries to be associated with hardware interrupts and allow control of the amount of storage associated with task activations. The pragma SHARED can be used to impose some order upon accesses to shared variables. Finally, three *attributes* let programs find out whether a task is callable or terminated and the number of tasks that are waiting to call a particular entry.

3. Overview of Ada+ Compiler

The Ada+ compiler, originally written to run on the PERQ workstation under the Accent operating system, is a compiler that implements most of the features of Ada. It processes programs in four major phases, split between three programs called the Front End, the Middle End, and the Back End. The Front End performs syntactic and semantic analysis and produces compilation databases both for its own use and for the use of the Middle and Back Ends. The Middle End copies the bodies of generic units to complete their instantiations. Using the files that the Front and Middle Ends produce, the Back End generates PERQ QCode files that may be linked and run on a PERQ under Accent.

For the purposes of adding the tasking mechanism described below, the compiler was moved to the VAX under Mach [9] with the aid of a program that translates PERQ QCodes to VAX assembly language (called *pcod*) and a VAX/Mach version of the PERQ Pascal compiler. *Pcod* had to be modified on several occasions to fix bugs that did not show up when translating the simple code that the PERQ Pascal compiler generated, and it does not translate all of the PERQ's architectural features faithfully even now. But the VAX was and is where the "C Threads" package (described below) resides, and given the declining use of PERQs as a general computing resource at CMU, the move would probably have been desirable even without the attraction of the facilities provided by "C Threads".

More information about the non-tasking-related aspects of the compiler may be found in [2], which gives a more detailed overview, [8], which describes how it performs overload resolution, and [6], which describes how it implements generics. The instruction sets of the PERQ and the VAX are documented respectively in [7] and [4].

4. Implementation Details

The Ada+ compiler implements tasking using several kinds of descriptor records and a number of run-time system routines called from compiled code. The run-time routines encapsulate the synchronization code needed to implement Ada tasking semantics, which has the benefits of keeping inline code small and, in our case, of keeping synchronization code in an easy-to-read high-level language (PERQ Pascal). The implementation was built on top of the "C Threads" package [3], which provides multiple threads of control within a single address space, *mutex variables* to serialize access to critical variables, and *condition variables* to help threads sleep while waiting for conditions to become true. In the following pages, we will examine the subjects of task representation, task activation, task and master termination, simple entry calls and accept statements, simple select statements, timed and conditional entry calls and select statements, and selects with terminate alternatives.

4.1. Task Representation

Ada defines tasks as "entities whose executions proceed in parallel" in the sense that each task executes on a "logical processor" of its own, and that tasks "proceed independently except at points where they synchronize." But its detailed rules about items like activation, termination, and rendezvous imply that some state information must be associated with each task in some manner that allows other tasks to access it.

The Ada+ compiler implements a task as some code, a "logical processor" to run it, and some state information stored in a place where other tasks can access it. The code is the compiled version of the task body, obtained by compiling the Ada source to QCodes, then translating the QCodes into VAX object code. The "logical processor" is composed from a C thread and two auxillary stacks, the MStack and the AStack, needed to set up a proper run-time environment. The C thread provides context switching, shared memory, and VAX stack space. The assembly code that *pcod* generates uses the MStack to simulate the PERQ MStack, and the AStack, a "software stack" created by the compiler, holds large function return values (arrays and records) that cannot be safely stored on the MStack. Finally, the state information consists of a task descriptor, whose address is stored in the appropriate task variable or task-valued component, and an array of entry descriptors.

Actually, the picture is slightly more complicated than this. As noted earlier, task bodies may access the local variables of surrounding blocks, subprograms, and tasks. The addresses of such intermediate variables may change between invocations, and the code that *pcod* generates to access them locates them by calling a routine that searches up the VAX stack until it finds a frame with a nesting level equal to that of the desired variable[2]. Merely placing an intermediate variable in shared memory is not sufficient to support its use by tasks other than the one which elaborated it, as the stack frame containing the variable does not form a part of the stacks of the other tasks, thus causing the standard *pcod* search to fail. The run-time system thus modifies a frame pointer and an activation pointer in the VAX stack frame for each new task so that the stack for the new task is effectively linked to the stack for its master task at the point of the stack frame for its master. The modifications allow the standard *pcod* search to succeed for both private and shared intermediate variables, although they necessitate restoring the changed pointers before thread termination to avoid program crashes.

To allow all the objects of a task type to share one compiled task type body, it is necessary to store some information for each task object in such a way that each task can access its own information by using a piece of shared code. Operating systems typically use registers to hold crucial information like the top of the call stack and program counter for each process, since registers generally must be saved and restored during context swaps and accesses to them tend to be fast. On the VAX, four registers are dedicated to system functions and several more are used by complex instructions, leaving six general-purpose registers both free and safe. *Pcod* uses one of these registers to contain a pointer to the top of each task's MStack, and the code generated by the Ada+ compiler uses another to give each task access to its descriptor. Figure 4-1 shows how the VAX registers are used by running Ada tasks. Note that by convention, C functions may trash low-numbered registers.

VAX register 10 goes to good use. Because task descriptors contain a number of fields used to

---

[2]This level is not part of the standard frame, but a *pcod* addition. Each VAX procedure generated by *pcod* starts by pushing several words onto the VAX call stack, next to the standard frame.

Figure 4-1: Use of VAX Registers by Compiled Ada Code

Register(s)	VAX Architecture/Ada+ Use	Safe wrt C function?
0 & 1	function results, service status	no
2 - 5	any / (not used if possible)	no
6 - 9	any / expression evaluation temp.	yes
10	any / address of task descriptor	yes
11	any / points to simulated MStack	yes
12	argument pointer (AP)	yes
13	frame pointer (FP)	yes
14	stack pointer (SP)	yes
15	program counter (PC)	yes

implement various aspects of tasking semantics, the address of a task's descriptor forms the "handle" by which that task is known both to run-time system routines and to other tasks. The fields of a task descriptor are as follows:

```
type
    TaskPtr   = ^TaskRec;
    MasterPtr = ^MasterRec;
    EntryPtr  = ^EntryRec;
    ActivPtr  = ^ActivationRec;
    SelectEnt = record
                  DelayAlt:     Boolean;
                  RetIndex:     Integer;
                  DelOrEnt:     Long;
                  end;
    SelectArr = Array[0..49] of SelectEnt;
    myVRD     = Array[0.. 1] of Long;

    TaskRec = record
                t_Current_Stack_Top: Long;       { environment }
                t_Stack_Top:         Long;       { environment }
                t_Stack_Base:        Long;       { environment }
                t_MStack_Base:       Long;       { environment }
                t_AP_For_LDA_I:      Long;       { environment }
                t_FP_For_LDA_I:      Long;       { environment }
                t_Funct:             myVRD;      { environment }
                t_Mutex:             Mutex;      { basic state }
                t_StatSem:           Mutex;      { basic state }
                t_State:             Integer;    { basic state }
                t_Activating:        Boolean;    { activation }
                t_Abnormal:          Boolean;    { exceptions }
                t_RaiseFlag:         Boolean;    { exceptions }
                t_CurrentTimer:      Timer_T;    { timer }
                t_TimeRefCnt:        Integer;    { timer }
                t_ActCount:          Integer;    { termination }
                t_ourParentT:        ActivPtr;   { activation }
                t_ourMasterT:        TaskPtr;    { termination }
                t_ourMasterB:        MasterPtr;  { termination }
                t_curMasterB:        MasterPtr;  { termination }
                t_masterLink:        TaskPtr;    { termination }
                t_SleepLock:         Mutex;      { sleep }
                t_SleepCond:         Condition;  { sleep }
```

```
    t_SleepCntr:        Integer;     { sleep }
    t_EntryNum:         Integer;     { rendezvous }
    t_EntryDesc:        EntryPtr;    { rendezvous }
    t_EntryPrev:        TaskPtr;     { rendezvous }
    t_EntryNext:        TaskPtr;     { rendezvous }
    t_Thread:           Long;        { 'cthread_t', env.}
    t_CurCaller:        TaskPtr;     { rendezvous }
    t_SavCaller:        TaskPtr;     { rendezvous }
    t_CurPriority:      Integer;     { rendezvous }
    t_SavPriority:      Integer;     { rendezvous }
    t_CallingTask:      TaskPtr;     { rendezvous }
    t_CalledEntry:      EntryPtr;    { rendezvous }
    t_EntryParams:      Long;        { 'void *', rndv. }
    t_Select:           SelectArr    { rendevous }
end;
```

These fields can be divided into eight rough categories, based on the functions they help implement. *Environment* fields contain information related to various stacks and to the launching of new tasks. The three *Basic State* fields consist of two locks which control access to most of the other fields in the task descriptor, and a field which indicates the current status of the task (running, attempting an entry call, etc.). *Activation* and *Termination* fields contain information related to the activation and termination of tasks and to the termination of master constructs. *Rendezvous* fields are used to implement entry calls, accept statements, and select statements. The *Sleep* fields allow a task to put itself to sleep awaiting an event, and the *Timer* fields are used for delay statements, timed entry calls, and timed selects. Finally, the *Exceptions* fields are meant to be used for the implementation of abort statements and the tasking-related aspects of exceptions.

4.2. Task Activation

Tasks are elaborated somewhat differently from other Ada objects. Whereas the elaboration of an object declaration with an initialization part results in the assignment of the value to the storage allocated for the object, the elaboration of a task declaration generally does not result in the immediate execution of the task. Instead, tasks created by elaborating a declarative part, package specification, or allocator begin execution at the "same" time, after other elaboration work (like initializing variables) has been performed. Before the access value can be returned or next statement begin execution, each task must complete its *activation.* A task's activation consists of the elaboration of the declarative part, if any, of its body. If any of the tasks activated by a declarative part or allocator raise an exception during their activation, the exception TASKING_ERROR must be raised after they have all completed their activations. For the purposes of this discussion, we will refer to a collection of such tasks as an *activation group* and to the construct responsible for launching them as an *activating construct.*

For each activation group, the code generated by the Ada+ compiler uses an *activation record* with the following fields:

```
type
    ActivationRec = record
                        a_count:        Integer;
                        a_raiseTE:      Boolean;
                        a_actlist:      TaskPtr;
                        a_mutex:        Mutex;
                        a_cond:         Condition;
                        a_prev:         ActivPtr
```

```
end;
```

These fields have the following purposes:

- A_Count is the number of tasks in the activation group whose declarations have been elaborated but which have not yet completed activation. After the activating construct finishes non-task-related elaboration and this count drops to zero, execution of the next program construct (for example, a function body) may begin if A_RaiseTE is not set (see below).
- A_RaiseTE indicates whether or not the activating construct should raise the exception TASKING_ERROR after A_Count reaches zero. Initially, it is false, but it may be set to true by any of the activating tasks.
- A_ActList is the head of a linked list of tasks that have not yet begun execution. Tasks on these lists are linked through their T_MasterLink fields, which are reused for another purpose once the tasks begin execution.
- A_Mutex and A_Cond serialize access to A_Count and A_RaiseTE by the tasks of the activation group, and permit the activating construct to sleep until all the tasks have "checked in" by decrementing A_Count.
- A_Prev is used to link all of the activation group records for a block, subprogram, task, or package into a single list whose head is stored in some location associated with the program construct. This field exists to make it easier to implement the Ada rule that an exception occuring in a declarative part or allocator causes unactivated tasks which would otherwise have been launched to pass into the terminated state without ever being activated.

Figure 4-2: Implementation of Activation and Termination Semantics

Part 1: Source for a Sample Ada Program

```
procedure PARENT is
    task type T is
        entry BAR(ARG1, ARG2: INTEGER);
    end T;

    X, Y: T;
    A   : integer := 5;
    Z   : T;

    task body T is
    begin
    -- statements of task body
    end T;
begin
-- statements of procedure body
end PARENT;
```

Several run-time routines support task activation. They are:

```
procedure Init_Activation(a, prev: ActivPtr);
function  New_Task(masterT: TaskPtr; masterB: MasterPtr;
                   actGroup: ActivPtr; entryCnt: Integer;
                   Proc: myVRD; ap_link: Long; fp_link:
                   Long): TaskPtr;
function  Launch_Tasks(me, MasterT: TaskPtr; MasterB: MasterPtr;
                   a: ActivPtr): ActivPtr;
procedure Complete_Activation(me: TaskPtr);
```

Figure 4-2, continued

Part 2: The Program's Translation, Expressed in Terms of C

```
int  entry_proc_bar(called_task, arg1, arg2, is_timed, delay_amt)
TaskPtr called_task;
int     arg1, arg2;
boolean is_timed;
float   delay_amt;
{ . . . }

void task_body_t(me)
TaskPtr me;
{
    Complete_Activation(me);
    /* statements of task body */
    SimpleTComplete(me);
}

void parent()
{
    /* prelude */
    ActivationRec  MyActGroup;
    Init_Activation(&MyActGroup, NULL);

    ActivationRec *ActGroupList;
    ActGroupList = &MyActGroup;

    MasterRec      MyMasterRec;
    Init_Master(me, &MyMasterRec);

    /* declarative part */
    TaskPtr        x = New_Task(me, &MyMasterRec, MyActGroup,
                                1 /* entry */,  &task_body_t,
                                0, 0 /* use default AP/FP */);
    TaskPtr        y = New_Task(me, &MyMasterRec, MyActGroup,
                                1, &task_body_t, 0, 0);
    int            a = 5;
    TaskPtr        z = New_Task(me, &MyMasterRec, MyActGroup,
                                1, &task_body_t, 0, 0);

    ActGroupList = Launch_Tasks(me,me,&MyMasterRec,ActGroupList);

    /* statements of procedure body */

    SimpleMComplete(me, &MyMasterRec);
}
```

`Init_Activation` initializes the fields of an `ActivationRec`. It is typically called at or near the start of the code generated for an activating construct. `New_Task` creates a task descriptor for a new task, complete with pointers to its activation group, master task, and master construct records as well as the information needed to launch the task at some later time. It is called by the code generated for task variable or component declarations. `Launch_Tasks` takes a whole activation group, launches each of its tasks, and makes the appropriate master responsible for monitoring their termination. It also waits until all

of the tasks in the group have finished activation before returning, and is normally called by the last part of the code generated for an activating construct. `Complete_Activation` is called by each new task as it completes its activation to record the event in the activation record and to wake the parent task if appropriate. Figure 4-2 shows the use of these routines.

4.3. Task and Master Termination

Both tasks and the constructs on which they depend, called *masters*, are subject to rules regulating their termination. Every task depends directly on one master and may depend indirectly on several others. A master may be a task, a currently executing block or subprogram, or a library package. Tasks created by evaluating an allocator depend on the master that elaborated the access type definition, while those created more "directly" depend on the master whose execution created them.

A master construct may not terminate until all of the tasks which depend on it have terminated. This guarantees that the storage used by intermediate variables will not disappear, or be reused, until all of the tasks which can access them are through running. Showing the validity of this guarantee for machines with a typical call stack organization is simple. Given Ada's visibility rules, the master of any task must have a nesting depth at least as great as that of the task type body[3]. Since the body can only access variables at its own or lower depths, this implies that these variables must remain allocated throughout the execution of the master construct.

A task may terminate if its execution is completed and any tasks dependent on it are terminated. A task may also terminate if it has reached a select statement with an open terminate alternative, its master has completed execution, and every task dependent on the master is terminated or has also reached a select statement with an open terminate alternative. When a task terminates, the exception `TASKING_ERROR` must be raised in any task waiting to call one of its entries.

For each direct master, the code generated by the Ada+ compiler uses a *master construct record* with the following fields:

```
type
    MasterRec = record
                m_Completed:    Boolean;
                m_DepCount:     Integer;
                m_DepTasks:     TaskPtr;
                m_Previous:     MasterPtr;
                m_SleepCond:    Condition
                end;
```

These fields have the following purposes:

- `M_Completed` indicates whether or not the master construct has completed execution. Currently, its value is set but not used, as the task termination algorithms do not care whether the master is completed.
- `M_DepCount` indicates the number of tasks directly dependent on the master construct which are still "active". Termination of the master construct may not take place until the master has completed its execution and this field has a value of zero.
- `M_DepTasks` points to the head of a linked list of tasks which are directly dependent upon

[3]We do not count packages towards the nesting depth, since their variables are typically allocated as globals or as locals of immediately-enclosing subprograms, blocks, and tasks.

the master construct. The tasks on the list are linked through their T_MasterLink fields.

- M_Previous is used to link all of the master construct records for masters belonging to one task into a list whose head is stored in the T_curMasterB field of the task descriptor. This field, not currently used, could provide a way to implement abort statements in the absence of exceptions by giving tasks a way to "unwind" the tasking-related portion of their call stacks and to find out which tasks are directly dependent upon a task that is to be aborted.
- M_SleepCond allows a completed master construct to sleep while awaiting the termination (or offers to terminate) of its dependents.

Normally the value of M_DepCount starts high, when an activation group for a declarative part is launched, and becomes lower as tasks complete executing. However, it may go up, even after the master construct has completed execution, for either of two reasons. The evaluation of an allocator may create new tasks directly dependent on the master. Additionally, a task which has decremented M_DepCount in preparation for selecting an open terminate alternative may acquire a caller which makes selecting it impossible and requires that the change to M_DepCount be "undone".

Several run-time routines support task and master block completion. They are:

```
procedure Init_Master(me: TaskPtr; m: MasterPtr);
procedure SimpleMComplete(me: TaskPtr; m: MasterPtr);
procedure SimpleTComplete(me: TaskPtr);
procedure MasterTComplete(me: TaskPtr; m: MasterPtr);
```

Init_Master initializes a master construct record. SimpleMComplete handles the completion of a master which is not also a task by setting the M_Completed field, sleeping until all dependent tasks have terminated or offered to terminate, and waking up dependents who offered to terminate with a message saying "go ahead". SimpleTComplete handles the completion of a task which is not also a master by waking up all of the task's current callers with the exception TASKING_ERROR and informing the task's master(s) that the task has completed its execution. Finally, MasterTComplete combines the functions of SimpleMComplete and SimpleTComplete for a task which is also a master construct. Figure 4-2 shows the use of three of these four routines.

4.4. Simple Entry Calls and Accept Statements

Entry calls, accept statements, and select statements are implemented using several run-time system routines whose purpose is to encapsulate synchronization details, together with a moderately large amount of state information which these routines manipulate. Some of this information is per single entry or entry family and per task type, some is per single entry or member of an entry family and per task, and some is per task. Additionally, the compiler generates an interface function for each single entry or entry family which is shared by all tasks of the corresponding type. This section describes how the compiler implements simple entry calls and accept statements -- those involving a single entry, no timeouts, no conditional test, and no terminate alternative. More complex constructs are implemented on top of this foundation and will be discussed later.

One of the most basic data structures used to implement entry calls and accept statements is the entry descriptor. For each single entry and for each member of an entry family, one descriptor is allocated to hold information describing its current state. An entry descriptor contains three fields

```
type
    EntryRec = record
                   e_Gate:          Boolean;
```

```
    e_Count:        Integer;
    e_WList:        TaskPtr
end;
```

which are used as follows:

- `E_Gate` is set by a task executing an accept or select statement to indicate that a call on the corresponding entry can be immediately accepted.
- `E_Count` counts the number of tasks which are waiting to call the entry, and thus makes implementation of the `'COUNT` attribute trivial.
- `E_WList` points to the head of a list of sleeping tasks which are waiting to call the entry. The list is circular to allow quick insertion and removal, and it uses two fields (`T_EntryPrev` and `T_EntryNext`) in the descriptors for the calling tasks to hold the links so that dynamic allocation can be avoided.

Since any given entry call is a call on a particular task, every task of a task type has its own set of entry descriptors; that is to say, if task type T has an entry E, and two variables X and Y are of type T, there will be separate entry descriptors for X.E() and Y.E().

Some information about entries and entry families is shared between all tasks of a type. Part of the process of calling an entry when its `E_Gate` is set is to clear the `E_Gate` field of every entry descriptor belonging to the called task, to indicate that the current accept or select statement has been "taken". Following Habermann and Nassi [5], the compiler places the entry descriptors for each task into an array to facilitate this process. The assignment of entries to array slots is made on a per-task-type basis, as every task of the same type has the same interface. Normally, the assignment is made at compile time. However, index ranges in entry family declarations are not required to be static, so in some cases slot assignments and the sizes of descriptor arrays must be computed at run time. This work, when needed, is performed during the elaboration of a task type's specification. Figure 4-3 shows an assignment of entries to slots in a descriptor array that might result from elaborating a typical task type specification.

Figure 4-3: Sample Assignment Of Entry Descriptors To Slots

```
task type EXAMPLE is
    entry FOO(<parms>);
    entry BAR(1..N)(<parms>);
    entry BAZ(8..9)(<parms>);
end EXAMPLE;

ENTRY   START   INDEX RANGE
-----   -----   -----------
 FOO      0     <not applicable>
 BAR      1     1 to N
 BAZ     N+1    8 to 9

Total number of descriptors:
        1 + (N-1+1) + (9-8+1)
     or
        N + 3
```

```
  Entry Descriptor Array
+-------------------------+
| 0     FOO               |
+-------------------------+
| 1     BAR(1)            |
+-------------------------+
| 2     BAR(2)            |
+-------------------------+
| ...   BAR(3)..BAR(N-1)  |
+-------------------------+
| N     BAR(N)            |
+-------------------------+
| N+1   BAZ(8)            |
+-------------------------+
| N+2   BAZ(9)            |
+-------------------------+
```

Three run-time system routines handle all of the synchronization required to implement simple entry calls and accept statements. They are:

```
function Do_Entry_Call(me, CalledTask: TaskPtr;
```

```
                        entryDesc      : EntryPtr;
                        HaveDelay      : Boolean;
                        DelayAmount    : Real;
                        ParamAddress   : Long): Boolean;
procedure Do_Accept(me: TaskPtr; e: EntryPtr);
procedure End_Rendezvous(me: TaskPtr);
```

A simple entry call is implemented as a call to an *entry function* whose return value is ignored. The compiler generates an entry function for each single entry or entry family which is shared by all the members of a family and by all the tasks of a task type. The entry function takes a pointer to the descriptor for the called task, followed by the index from the entry call (if an entry family is involved), the declared arguments of the entry, a flag indicating whether the call is timed (in this case, `FALSE`), and a delay amount (in this case, ignored). If the function is for an entry family, it checks to see if the index is within the proper range. In any event, it computes the address of the appropriate descriptor, calls `Do_Entry_Call` to do the real work, and returns the result of `Do_Entry_Call`. The main reason that entry functions exist is to reduce variable numbers of arguments scattered all over memory into blocks of constants and pointers which can be described by their addresses and sizes (all `Do_Entry_Call` needs is an address), and which can be block-copied to the locals of an accept statement.

Figure 4-4: Translation of Simple Entry Calls and Accept Statements

Part 1: Source for a Sample Ada Program

```
procedure FOO is
    task CALLEE is
        entry ADD(A: in out INTEGER; B: INTEGER);
    end CALLEE;

    task CALLER is
    end CALLER;

    task body CALLEE is
    begin
        accept ADD(A: in out INTEGER; B:INTEGER) do
            A := A + B;
        end ADD;
    end CALLEE;

    task body CALLER is
        X: INTEGER := 5;
    begin
        CALLEE.ADD(X, X);
    end CALLER;
begin
null;
end FOO;
```

An accept statement is implemented as a check of the entry index (if the entry belongs to a family), followed by a call to `Do_Accept` with the addresses of the descriptors for the current task and the entry it wants to accept, a block-copy of the arguments from the caller's stack into some locals reserved to hold them (if the entry has arguments), code for the statements of the body (if the accept statement has a body), and finally a call to `End_Rendezvous`. Figure 4-4 shows the translation of a simple entry call and

Figure 4-4, continued

Part 2: Task CALLER's and CALLEE's Translations, Expressed in Terms of C

```
int entry_proc_add(called_task, a, b, is_timed, delay_amt)
TaskPtr called_task;
int     *a, b;
boolean is_timed;
float   delay_amt;
{
    return Do_Entry_Call(me,called_task,&called_task->t_EntryDesc[0],
                                                is_timed, delay_amt, &a);
}

void task_body_callee(me)
TaskPtr me;
{
    Complete_Activation(me);
    {
        int *a,b;
        Do_Accept(me, &me->t_EntryDesc[0]);
        bcopy(me->t_EntryParams, &a, sizeof(a) + sizeof(b));
        *a = *a + b;
        End_Rendezvous(me);
    }
    SimpleTComplete(me);
}

void task_body_caller(me)
TaskPtr me;
{
    int x = 5;
    Complete_Activation(me)
    {
        int vrtemp = x;     /* in out scalars by value-result */
        entry_proc_add(callee, &vrtemp, x, FALSE, 0.0);
        x = vrtemp;
    }
    SimpleTComplete(me);
}

void foo()
{...}
```

an accept statement.

Since the inline code generated for simple entry calls and accepts is fairly trivial as a result of its use of run-time system routines, just stating how and when run-time system routines are called is not sufficient to adequately describe the implementation. The implementation is perhaps best understood by examining simplified versions of the run-time system routines `Do_Entry_Call` and `Do_Accept`, which show the basic logic that underlies the implementation of all kinds of entry calls, accept statements, and select statements.

`Do_Entry_Call` handles all synchronization connected with simple, timed, and conditional entry calls.

Ignoring complications caused by timed and conditional entry calls, abort statements and exceptions, and terminate alternatives in select statements, its heart is as follows:

```
{ var me, CalledTask: TaskPtr;     entryDesc, e: EntryPtr;}
Mutex_Lock(me^.t_StatSem);
Mutex_Lock(CalledTask^.t_Mutex);
if entryDesc^.e_Gate                { entryDesc is called entry }
then    begin
        { the task that owns the entry is waiting for a caller }
        set e^.e_Gate to FALSE for all entry descriptors e which
            belong to the task CalledTask;
        set me^.t_State to IN_RENDEZVOUS;
        set "calling task" field in CalledTask to point to me;
        Mutex_Unlock(me^.t_StatSem);
        { note that we pass CalledTask the lock on its t_Mutex }
        Wake(CalledTask)
        end
else    begin
        set me^.t_State to ENTRY_CALL;
        { link me onto the waiting list entryDesc^.e_Wlist and }
        {    increment entryDesc^.e_Count by one.              }
        Mutex_Unlock(CalledTask^.t_Mutex);
        Mutex_Unlock(me^.t_StatSem)
        end;
Sleep(me);      { sleep until exception or end of rendezvous }
```

`Do_Accept` handles all the synchronization for accept statements leading up to the execution of the accept body. With similar simplifications, its heart is as follows:

```
{ var me, t: TaskPtr;     e: EntryPtr;}
Mutex_Lock(me^.t_Mutex);
1:
if e^.e_WList <> nil
then    begin
        set t to the first task on e^.e_WList, and remove this task
            from the waiting list.
        lock t^.t_StatSem while checking its state to see if it can
            still serve as a valid caller -- if not, go to label 1;
        Mutex_Unlock(me^.t_Mutex)
        end
else    begin
        Mutex_Lock(me^.t_StatSem)
        set me^.t_State to ACCEPT;
        set e^.e_Gate to TRUE;
        Mutex_Unlock(me^.t_StatSem);
        Mutex_Unlock(me^.t_Mutex);
        Sleep(me);
        { whoever wakes us should pass us the lock on our t^_Mutex }
        set t to point to the task which woke us;
        set t^.t_State to IN_RENDEZVOUS;
        Mutex_Unlock(me^.t_Mutex)
        end;
```

Both of these routines work by examining and modifying the information stored in entry descriptors, using two task descriptor mutex fields to control access to this information. The reason that task descriptors contain two mutex fields (not counting `T_SleepLock`) is that `Do_Entry_Call` modifies state information for both the calling task and the called task, and using just one mutex field for each task could

cause the run-time system to deadlock when two tasks attempted to call each other simultaneously. Simultaneous calls should not cause deadlock if at least one is conditional or timed. So the run-time system routines related to rendezvous adopt the convention of locking the `T_Mutex` variable belonging to the called or accepting task in order to examine and modify entry descriptors, and locking `T_StatSem` in the calling task to modify its state. However, in general the proper way for a run-time routine to modify a task's state is to lock both its `T_Mutex` and its `T_StatSem` variables (in that order), change its state, and release both locks in reverse order. This guarantees that no simultaneous modifications are being made to the task's state. Routines that lock only one mutex variable must be written carefully to avoid problems that could arise because they do not always run in a mutually exclusive fashion relative to each other.

The critical regions in `Do_Entry_Call` and `Do_Accept` defined by the use of `T_Mutex` force a rendezvous to happen in one of two ways. The accepting task may "arrive" before any callers, in which case it sets `E_Gate` for the specified entry and goes to sleep. The next caller, on finding the `E_Gate` field set, clears all the callee's `E_Gate` variables, wakes it up and gives it the lock on its own `T_Mutex`, and then goes to sleep awaiting the end of the rendezvous. Or one or more callers may "arrive" first, in which case they queue themselves on `E_WList` and go to sleep. The accepting task will then find a non-empty waiting list, from which it selects the first task as a partner. In either case, the calling task continues to sleep while the accept statement body is executed and is woken up only at the end of the rendezvous. Note that the convention of passing the lock on an accepting task's `T_Mutex` when the caller is the last to "arrive" ensures that other tasks may not interfere with the start of a rendezvous once the decision has been made to start it, whether by `Do_Entry_Call` or by `Do_Accept`.

This scheme works, but corresponds to the "naive" implementation described by Habermann and Nassi [5] in requiring three scheduling points when an accept precedes an entry call[4]. There are several reasons why our tasking implementation does not use a scheme like the one they describe, where the task which arrives last executes the accept body. One is that given the PERQ QCode architecture, the compiler would be forced to translate the bodies of accept statements and select alternatives into QCode procedures rather than into inline code, which would introduce some extra run-time overhead and, more importantly, use that many more procedure slots of the roughly 250 available for each QCode file. Another is that it would involve manipulating the contents of registers to do pseudo-context-switches, a task that would not be overly hard, but that seems contrary to the spirit of the "C Threads" package. Finally, some fields in task descriptors have been designed under the assumption that a task can be involved in at most one entry call, an assumption which does not hold under the Habermann and Nassi optimization. An example of this is the set of fields which allow a task to put itself to sleep; a calling task which assumes the identity of the called task and executes its accept body could not safely reuse the callee's sleep fields in the event the accept body contained an entry call.

These reasons aside, it would probably be possible to do a Habermann and Nassi style tasking implementation for the Ada+ compiler without too many changes to the compiler itself by changing the declarations of the descriptor records and rewriting parts of the run-time system library. It would be necessary to make the compiler generate procedures for accept bodies, but the tasking interfaces and the parts of the compiler which generate code to use them could remain the same, and the existing run-time system provides a starting point for writing a more optimizing implementation.

---

[4]Ignoring competition for the locks of the caller and the called task and any rescheduling that might be done to improve fairness.

4.5. Simple Select Statements

Simple select statements differ from accept statements in that they may offer to accept more than one entry, and make the offer to accept a particular entry conditional on a run-time condition known as a guard. To handle them, the compiler uses the *select alternative table* built into each task descriptor and a more generalized version of `Do_Accept` called `Do_Select`:

```
function  Do_Select(me:TaskPtr; n:Integer; OpenTerm:Boolean): Integer;
```

The select alternative table, as its name suggests, contains information about each alternative in a select statement. It contains enough room to describe up to fifty select alternatives, fifty being arbitrarily picked as a larger number of alternatives than any reasonable select statement is likely to contain[5]. Each entry contains a flag indicating whether it describes a delay alternative or an accept alternative, an integer "return index" which `Do_Select` should return if that alternative is selected, and a field which is big enough to hold either a delay value or a pointer to an entry descriptor, depending on the type of the alternative as indicated by the flag.

Figure 4-5: Implementation of Select Statements

Part 1: Source for a Sample Ada Task

```
task MUTEX is
    entry LOCK;
    entry UNLOCK;
end MUTEX;

task body MUTEX is
    BUSY : BOOLEAN := FALSE;
begin
    loop
        select
            when not BUSY =>
                accept LOCK do
                    BUSY := TRUE;
                    end LOCK;
        or
            when BUSY =>
                accept UNLOCK do
                    BUSY := FALSE;
                end UNLOCK;
        or
            terminate;
        end select;
    end loop;
end MUTEX;
```

A select statement is translated into a series of assignments and tests to load the select alternative table for the current task's descriptor with information describing all open alternatives, followed by a call to `Do_Select` with the number of open alternatives in `n`, a "case statement" jump indexed using the value

[5] If this limit turns out to be a problem, it would be easy to raise it or add code to dynamically allocate space for large select tables. However, the PERQ-related limit on the number of procedures that can be in a QCode file effectively limits task types to have less than 250 single entries or families, and thus also limits the size of potential select statements.

Figure 4-5, continued

Part 2: Task Body MUTEX's Translation, Expressed in Terms of C

```
void task_body_mutex(me)
TaskPtr me;
{
    int busy = FALSE;
    Complete_Activation(me);
    while ( ; TRUE ; ) {
        int count = 0;
        if (! busy) {
            SelectEnt *sel_ent;
            sel_ent = me->t_Select + (Size_Select_Ent * count++);
            sel_ent->RetIndex = 1;
            sel_ent->DelayAlt = FALSE;
            sel_ent->DelOrEnt = (long) &me->t_EntryDesc[0];
        }
        if (busy)   {
            SelectEnt *sel_ent;
            sel_ent = me->t_Select + (Size_Select_Ent * count++);
            sel_ent->RetIndex = 2;
            sel_ent->DelayAlt = FALSE;
            sel_ent->DelOrEnt = (long) &me->t_EntryDesc[1];
        }
        switch (Do_Select(me, count, TRUE)) {
            case 1:
                busy = TRUE;
                End_Rendezvous(me);
                break;
            case 2:
                busy = FALSE;
                End_Rendezvous(me);
                break;
        }
    }
    SimpleTComplete(me);
}
```

returned by `Do_Select`, and code for each of the select alternatives. The code generated for each accept alternative is identical to the code that is generated for accept statements after the call to `Do_Accept`, except that it ends by jumping to the end of the select after calling `End_Rendezvous`. Figure 4-5 shows the translation of a select statement with a terminate alternative; imagine that the third parameter to `Do_Select` is `FALSE` rather than `TRUE` to see how the select would be translated if it did not have a terminate alternative.

Inside `Do_Select`, the difference between one open entry and multiple open entries adds some, but not much, code complexity as compared to `Do_Accept`. Rather than checking one entry descriptor, `Do_Select` checks multiple entry descriptors to see if tasks are waiting to call any one of them. Rather than setting one `E_Gate` if there are no waiting callers, `Do_Select` sets the `E_Gate` fields for each accept alternative which is described in the select alternative table[6]. Finally, if `Do_Select` puts itself to

[6]This why a calling task that finds the `E_Gate` variable for the entry it is attempting to call set must clear all `E_Gate` variables belonging to the called task, and not just that particular one.

sleep awaiting an entry call, it must compare the pointer to the descriptor for the called entry, provided to it through a task descriptor field by `Do_Entry_Call`, against the pointers stored in the select alternative table to determine which index value should be returned and thus which case branch should be executed. Note that while Ada allows programs to contain select statements with two or more alternatives selecting the same entry, there is no guarantee that calls will be distributed "fairly" between such alternatives, and in particular `Do_Select` will always select the first one.

4.6. Timed and Conditional Entry Calls and Select Statements

Timed entry calls, select statements with delay alternatives, and simple delay statements are implemented on top of a flexible timer package previously built for another application. The timer package uses a thread to manage a queue of events and provides operations to add events to the queue and to try to cancel them. Associated with each event is a pointer to a C function supplied by the user which will be executed when the event's deadline is reached and a pointer which may contain arbitrary data supplied by the user of the package.

Timed entry calls and select statements are translated into inline code which is not significantly more complex than that generated for simple entry calls and select statements. Both `Do_Entry_Call` and the functions the compiler generates to access it take a flag indicating whether an entry call is timed and a floating-point value[7] indicating how long to wait if the flag is set, and return a Boolean indicating whether the corresponding delay alternative should be executed. So a timed entry call is translated as a simple if statement whose condition is a call to the entry function and whose body contains the statements of the delay alternative. Timed select statements are even simpler, because entries in the select alternative table implicitly passed to `Do_Select` may describe delay as well as accept alternatives. Finally, delay statements are translated into calls to `Delay_Statement`. Since conditional entry calls and select statements have the same semantics as timed entry calls and select statements with a delay of zero seconds, they are translated as such.

Within the run-time system, the implementation of timed entry calls consists of three major pieces of code:

- Code to put a calling task onto the timer list if a rendezvous with the called task is not immediately possible and the delay time is greater than zero. This is located in `Do_Entry_Call` and a C interface routine it calls.
- Code to remove a calling task from the appropriate entry waiting list and wake it up if the specified amount of time expires without a rendezvous happening. This is located in the C action routine that the timer thread executes for all entry call timeout events.
- Code to cancel the timeout of a timed entry call if a rendezvous happens before the time expires. This is located in `Do_Accept` and `Do_Select`.

One problem confronting this code is the possibility that a "race condition" might arise between code in `Do_Accept` (or `Do_Select`) attempting to cancel a timeout, and code in the C action routine attempting to cause one. To make absolutely sure that it does not cause a deadlock, the code in the C action routine locks down the mutex variables for only one task at a time, although it must modify state information belonging to both the calling task and the called task. Its rough outline looks like this:

---

[7]Because the compiler does not currently implement fixed-point types, delay values must be expressed as floating-point numbers.

```
lock caller's t_Mutex and t_StatSem;
if (caller's state is TIMED_ENTRY_CALL and the Interference
    variable has not been set)
then    begin
        set caller's state to TIMEDOUT;
        unlock caller's t_Mutex and t_StatSem;
        lock callee's t_Mutex;
        if caller is on entry waiting list remove it from list;
        unlock callee's t_Mutex;
        wake caller;
        end
else    unlock caller's t_Mutex and t_StatSem;
set Interference to FALSE and decrement caller's TimeRefCnt;
```

The relevant code from `Do_Accept`, in paraphrased form, is

```
lock callee's t_Mutex;
1:
remove task "caller" from entry waiting list;
lock caller's t_StatSem;
if (caller is abnormal, or caller's state is not one of
    ENTRY_CALL or TIMED_ENTRY_CALL)
then    begin
        unlock caller's t_StatSem;
        goto 1 to look for another task;
        end;
if caller's state is TIMED_ENTRY_CALL
then    CancelTimeout(caller);
        { try to remove caller from timer queue;
          if delete operation successful
          then    decrement caller's t_TimeRefCnt
          else    /* the timer thread must have removed it */
                  set Interference := TRUE;
        }
unlock caller's t_StatSem;
unlock callee's t_Mutex;
```

Note that the check of the calling task's state in `Do_Accept` occurs when both the `t_StatSem` of the calling task and the `t_Mutex` of the callee are locked, which gives rise to four cases:

- `Do_Accept` gets to the state check before the timer action routine has been called, and subsequently removes the entry call timeout event from the timer's queue successfully. In this case, the timeout has been cancelled successfully and execution of the rendezvous may proceed uninhibited.

- `Do_Accept` gets to the state check before the timer action routine has set the caller's state to TIMEDOUT, but too late to remove the timeout event from the timer queue. This would cause major problems if the C action routine was allowed to continue, as it would be trying to change the state of the calling task and to wake up the calling task as the rendezvous was happening, or even afterwards when the caller was engaged in another entry call. To handle this condition, we observe that since both pieces of code lock the caller's `T_StatSem`, the timer action routine must be stuck at or before its lock request. We use this observation to set a variable called Interference which tells the timer action routine that it should not change the task's state for the reason that a rendezvous occurred. This prevents the action routine from making destructive state changes, but leaves open the remote possibility that if the timer thread is starved until after the task descriptor is deallocated it could dereference a dangling pointer. To prevent this from happening, the run-time system keeps a timer reference count in each task descriptor that is incremented when a task is placed on the

timer list and is decremented when the task's timeout event is removed from the list or its action routine completes execution.

- `Do_Accept` gets to the state check after the timer action routine has set the caller's state to TIMEDOUT, but before it can lock the callee's `T_Mutex` and attempt to remove the task from the entry queue. Upon looking at the state field of the caller, `Do_Accept` will realize that a timeout has occurred and look for another task. When it releases its `T_Mutex` variable at some later point, the action routine will be free to time out the caller, which will still be sleeping and in the TIMEDOUT state.
- `Do_Accept` never gets to the state check because the action routine removes the caller from the entry queue before `Do_Accept` gets a chance to even see it. The action routine thus times out the calling task without interference.

Even with the interlocks provided by the Interference variable and the timer reference counts, one problem remains. Although the timer will not interfere with a rendezvous in progress and will not be left with dangling pointers, it is still possible for it not to time out an entry call or select statement "on time" if it is starved for cycles or must wait to obtain locks. Potentially, a rendezvous could occur after the delay period specified in a timed entry call or timed select had expired, which is not quite as bad as a crash, but is still not desirable. To solve this problem, it would be sufficient to add an "expiration time" field to each task descriptor and a small amount of code to set and check it to `Do_Entry_Call`, `Do_Accept`, and `Do_Select`.

The implementation of timed select statements involves similar pieces of code, problems, and solutions. Therefore, in the interests of brevity, we will not discuss it further.

4.7. Selects with Terminate Alternatives

Select statements with terminate alternatives, like timed select statements, are translated into simple inline code. All the inline code must do is pass either TRUE or the value of the guard as the third argument to `Do_Select`. But inside the run-time system, terminate alternatives cause a significant amount of extra complexity. In brief, tasks that execute select statements with an open terminate alternative are placed into one of two states depending on "local" conditions. Tasks with no active dependents and no callers are marked `TERMINABLE` and their masters are notified as if they had died; all other tasks are marked `SELECT_TERM`. Subsequent events that occur while a task in either state is waiting for rendezvous or termination may force that task to be put into the other state, with attendant changes to the records of its master construct, and perhaps those of indirect master constructs. This scheme is based on the one used by the NYU Ada/Ed compiler.

The events that may force a task to change state are well-defined. Two events may force a task's state to change from `SELECT_TERM` to `TERMINABLE`: it may go from having one or more active dependents to having none as dependents terminate, and it may go from having one or more callers to having none as the delays on timed entry calls expire. One event can force a task's state to change from `TERMINABLE` to `SELECT_TERM`: the acquisition of new callers (whether they are calling open entries named in the select statement or not). To handle these events, the run-time system uses two internal procedures named `IndicateTerm` and `IndicateLife`, both of which take pointers to a task's master task and master construct descriptors.

`IndicateTerm` decrements the `M_DepCount` and `T_ActCount` fields which belong to the master construct and task, and calls `condition_signal` on the master construct's `M_SleepCond` field if

necessary. It then quickly checks the current state of the master task, and if it finds that the master task has no active dependents and no callers and is in the SELECT_TERM state, calls itself recursively on behalf of the master task. IndicateTerm is called by Do_Entry_Call in response to some entry timeouts, by Do_Select in response to some select statements, and by the routines SimpleTComplete and MasterTComplete in response to the normal completion of a task.

IndicateLife increments the M_DepCount and T_ActCount fields which belong to the master construct and task. It checks to see if the master task is in the TERMINABLE state, and if so, changes the master task's state to SELECT_TERM and calls itself recursively to increment the counters for the master task's master task and master construct. IndicateLife is called by Do_Entry_Call in response to entry calls on TERMINABLE tasks.

4.8. Other Topics

Dynamically allocated tasks. Tasks created by evaluating allocators are treated much like declared tasks. But because the tasks which launch them are not always the same as their master tasks, the compiler generates extra code for each access type whose use can result in task object creation. This code stores four pointers: a pointer to the current task, a pointer to the current master construct record, and the VAX frame and activation pointers needed to link the stacks of new tasks to the stack of the current task at the frame corresponding to the access type declaration. These pointers can then be passed to New_Task by the code generated for allocators.

Library tasks. Some tasks depend on library packages rather than blocks, subprograms, or tasks. To handle them, the run-time system allocates a task descriptor and a master construct record for a fictional "library task", and calls Init_Master and SimpleMComplete at appropriate times. Although the main program may complete execution before the library tasks terminate, the run-time system returns control to the shell only when both the main program and the library tasks are finished.

Entry renaming and entries as generic actuals. Single entries and entry family members may be renamed as subprograms and used as generic actual parameters in place of subprograms. In both cases, the compiler must generate code to freeze the identity of the task being called and (in the case of a family) the entry index at the point of the renaming declaration or generic instantiation.

Goto, exit, and return statements. Goto, exit, and return statements may transfer control out of one or more master constructs (blocks, and subprograms in the case of return statements). Masters are not allowed to terminate while they have active dependents. Thus, the compiler must sometimes generate calls to SimpleMComplete as part of the code it generates for these statements.

4.9. Remaining Work

Currently the VAX/Mach version of the compiler does not implement exceptions, because the compiler uses more of the PERQ's exception architecture than the *pcod* program can translate. The code that the compiler generates and the code in the run-time system contain all of the hooks needed to raise them, and *pcod* translates them well enough that when a program raises one it prints a message and dumps the contents of memory to a file, but a suitable handling mechanism is still needed. If *pcod* suddenly started translating the PERQ architecture perfectly, it would still be necessary to modify the compiler to implement the tasking-related aspects of exception handling, such as waiting for a block's dependents to

terminate before propagating an uncaught exception outside the block.

Other features that should be present in a full Ada implementation but which are missing are abort statements and the SHARED pragma. The compiler also restricts programs to a single priority and disallows representation clauses, as allowed by Ada. Nothing special is done about reclaiming memory allocated by tasks after their death because the compiler did not try to reclaim memory used by allocators before the tasking mechanism was added. Finally, the Ada+ versions of the standard I/O packages have yet to be adapted for the VAX and multitasking.

5. Performance Measurements

To obtain some indication of the performance of our tasking implementation, we ran some tests whose results are presented below. Several factors contributed to the results we obtained, including the

- Machines used to run the tests
- Operating systems running on those machines
- System libraries used to link the test programs
- Compiler options used when compiling the test programs
- Method used to obtain test times

The two machines used to run the tests were a VAX 11/780 (ARPAnet host SPICE.CS.CMU.EDU) and a four-processor VAX 8200 (R2D2.MACH.CS.CMU.EDU). Although the operating system on both machines was Mach, the versions of Mach on the two machines were different. The Spice VAX ran version "4.3 #5.1(F9): Tue May 5 00:13:53 EDT 1987; /usr3/uk/GENERIC_VAX (dist.fac.cs.cmu.edu)", an old version without kernel support for threads. The R2D2 VAX ran Mach version "4.3 #1.0: Thu Sep 17 15:28:57 EDT 1987; /usr2/rvb/mk/MACHTT (temp.ius.cs.cmu.edu)", which supports threads and appears to be better tuned. Since the libraries on both machines were identical, despite the different versions of Mach, all binaries were produced and linked on the Spice VAX and simply copied to the R2D2 VAX for the purpose of getting timings there.

Two different versions of the C Threads libraries were used to run the tests. One implemented threads as coroutines; only one Ada task could be serviced by the available processor(s) at any given time. The other implemented threads as preemptively-scheduled Mach threads running within a single address space. Because context switches under the coroutine version of C Threads occur only when the threads package gains control from the user program, we divided the coroutine threads tests into those in which the normal code generator output was used and those in which a compile-time switch was used to force the code generator to insert a call to `cthread_yield` into the code for each loop. In the tables presented in this section, coroutine threads tests will be denoted by ".yield" or ".noyield", depending on whether or not the compiler was instructed to insert `cthread_yield` calls, and Mach threads tests will be denoted by ".thread".

Times were computed by placing the constructs to be tested inside a procedure named TEST and by bracketing a call to TEST with calls to timer start and stop routines, so as to avoid counting "startup" costs (loading run-time libraries) against the constructs being measured. Three time measurements were taken for each test run: elapsed "user" time, elapsed "system" time, and elapsed "real" time. As defined by the Mach manual, "user" time is the amount of time that a program spends executing in user mode, and

"system" time is the amount of time spent in the system executing on behalf of the program. Both measurements are available through the `getrusage` call. "Real" time is simply clock time as measured by two calls to `gettimeofday`; since the test programs were run on multiuser systems, each real time measurement reflects the then-current system load as well as the time consumed by the test program. All times given below were obtained by averaging the results of five consecutive runs, to minimize the effects of run-to-run variations.

Please note that the choice of tests is somewhat arbitrary. We make no claims that these tests represent a complete set, but we believe the results presented below to be reasonably accurate. We would also like to acknowledge that code for two of the tests whose results are presented below, the Ada task chaining tests and the select statement tests, was obtained from a 1984 posting of Ada tasking benchmarks on USENET's net.sources newsgroup.

5.1. LOOP - cost of dividing work among tasks

To determine the practicality of dividing a small piece of work among several tasks, a series of test programs were written to perform ten thousand integer additions, subtractions, and assignments divided equally among one, two, five, and ten tasks. To complete the series, a program was written which performed the computations without tasks, as part of the main thread of control. The results of running these tests are summarized in Tables 5-1 and 5-2.

Table 5-1: Times to run the LOOP tests on SPICE.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Iteration
Loop0.yield	632ms	0ms	0.68/0.63	633ms	63us
Loop0.noyield	130ms	0ms	0.51/0.68	128ms	13us
Loop1.yield	756ms	162ms	1.69/1.53	932ms	92us
Loop1.noyield	202ms	156ms	0.20/0.47	362ms	36us
Loop2.yield	1248ms	416ms	1.53/1.16	1722ms	166us
Loop2.noyield	252ms	390ms	0.64/0.76	642ms	64us
Loop5.yield	1722ms	1138ms	1.16/1.03	2867ms	286us
Loop5.noyield	424ms	1160ms	0.84/0.93	1593ms	158us
Loop10.yield	2080ms	2752ms	1.03/1.00	5061ms	483us
Loop10.noyield	752ms	2732ms	0.95/0.99	3533ms	348us
Loop50.yield	5240ms	35872ms	1.00/2.24	64183ms	4102us
Loop50.noyield	3896ms	35710ms	0.99/2.03	60930ms	3961us

The results from both machines strongly suggest that tasks are not lightweight constructs that can be casually invoked. Even moving the computation from the main program into a single task was expensive, and the thinner the computation was spread, the slower it got. This was especially true when going from using ten tasks to using fifty tasks; the high elapsed system times observed suggest that somewhere between ten and fifty tasks, a limit is reached after which any new tasks incur considerable additional expense. Possibly this is related to the large amount of memory which the C Threads package and the Ada+ run-time system allocate for each task; by default, each task gets a megabyte of VAX call stack space and 150 kilobytes of AStack and MStack space.

Table 5-2: Times to run the LOOP tests on R2D2.MACH.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Iteration
Loop0.yield	594ms	6ms	1.42/2.30	597ms	60us
Loop0.noyield	128ms	0ms	3.88/3.59	132ms	13us
Loop0.thread	126ms	0ms	3.96/3.97	126ms	13us
Loop1.yield	826ms	134ms	2.89/2.51	958ms	96us
Loop1.noyield	196ms	136ms	3.39/2.37	333ms	33us
Loop1.thread	266ms	148ms	2.30/3.02	400ms	41us
Loop2.yield	142ms	268ms	2.67/2.85	1406ms	141us
Loop2.noyield	170ms	276ms	2.91/2.94	445ms	45us
Loop2.thread	198ms	320ms	3.02/2.45	411ms	52us
Loop5.yield	1502ms	656ms	2.85/3.10	2158ms	216us
Loop5.noyield	268ms	642ms	2.96/2.97	912ms	91us
Loop5.thread	356ms	1070ms	2.45/2.76	951ms	143us
Loop10.yield	1628ms	1296ms	3.40/2.20	2926ms	292us
Loop10.noyield	398ms	1330ms	2.98/2.11	1724ms	173us
Loop10.thread	976ms	4732ms	2.84/2.03	2590ms	571us
Loop50.yield	3180ms	8972ms	2.20/1.42	17647ms	1215us
Loop50.noyield	1626ms	8994ms	2.11/2.75	16471ms	1062us
Loop50.thread	6944ms	29642ms	2.03/2.05	20008ms	3659us

5.2. MULT - cost of dividing work among tasks

After obtaining the above results, we ran a very limited test to see if giving each of two tasks a large amount of work to perform would reduce the overhead of using tasks under the coroutine threads library and produce a speedup under the Mach threads library. The results of the limited test were very positive, so we devised a range of tests that fell between these two extremes in an attempt to discover roughly when it becomes inexpensive or advantageous to divide independent pieces of work among several tasks. These tests all had the characteristics that the body of the TEST procedure was a null statement (except in the special case of zero tasks), that the real work consisted of a certain number of iterations of an inner loop body produced by executing a pair of nested loops, and that this work consisted of ten integer multiplications, ten integer divisions, and ten integer assignments. The iterations were equally divided among the tasks.

Tables 5-3 and 5-4 summarize the results of running these tests on the Spice and R2D2 VAXen in textual form, while Figures 5-1 through 5-4 present the R2D2 results graphically. The results seem to show a definite advantage to using tasks when each task is given at least one-half second of work to perform. They also show that giving tasks less than one-quarter second of work can be costly.

5.3. COMP - cost of competition for rendezvous

To determine the effect that competition for an entry of a task would have on rendezvous times, four tests were run in which one, two, five, and ten client tasks called an entry of a server task in a tight loop. The server looped on a select statement with one accept alternative and one terminate alternative, and the number of entry calls was fixed at one thousand entry calls per test. Table 5-5 presents the times these tests took to run on the Spice VAX, while Table 5-6 and Figure 5-5 present the times they took to run on the R2D2 VAX.

Figure 5-1: MULT - 100,000 iterations

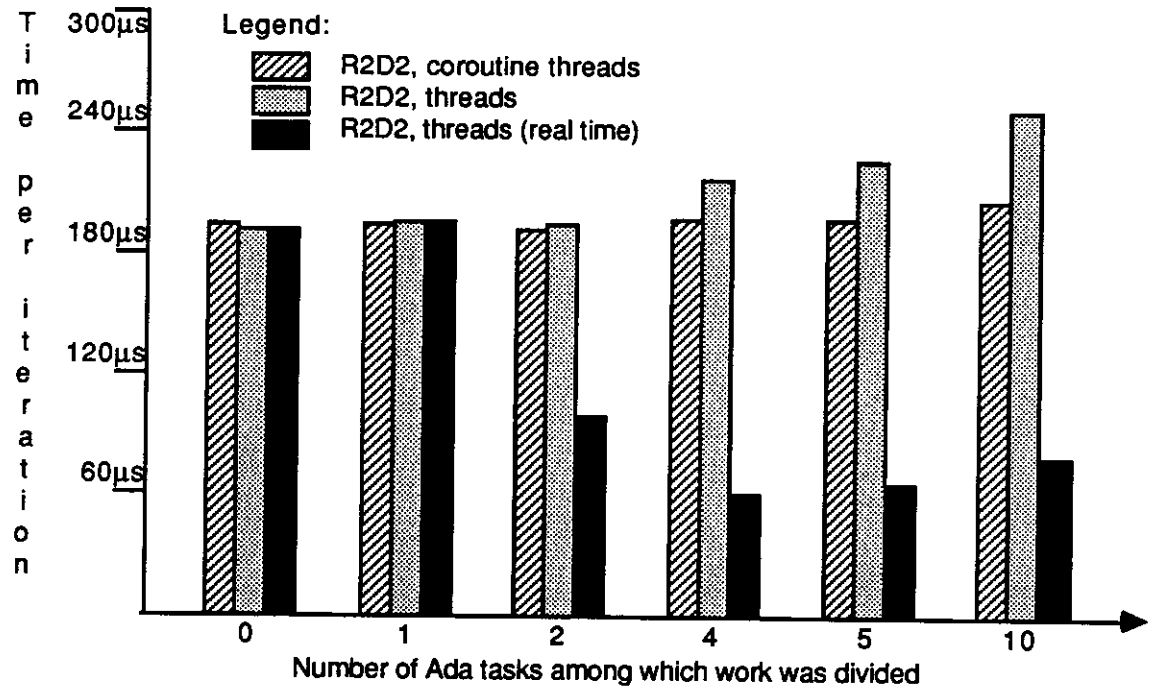

Figure 5-2: MULT - 25,000 iterations

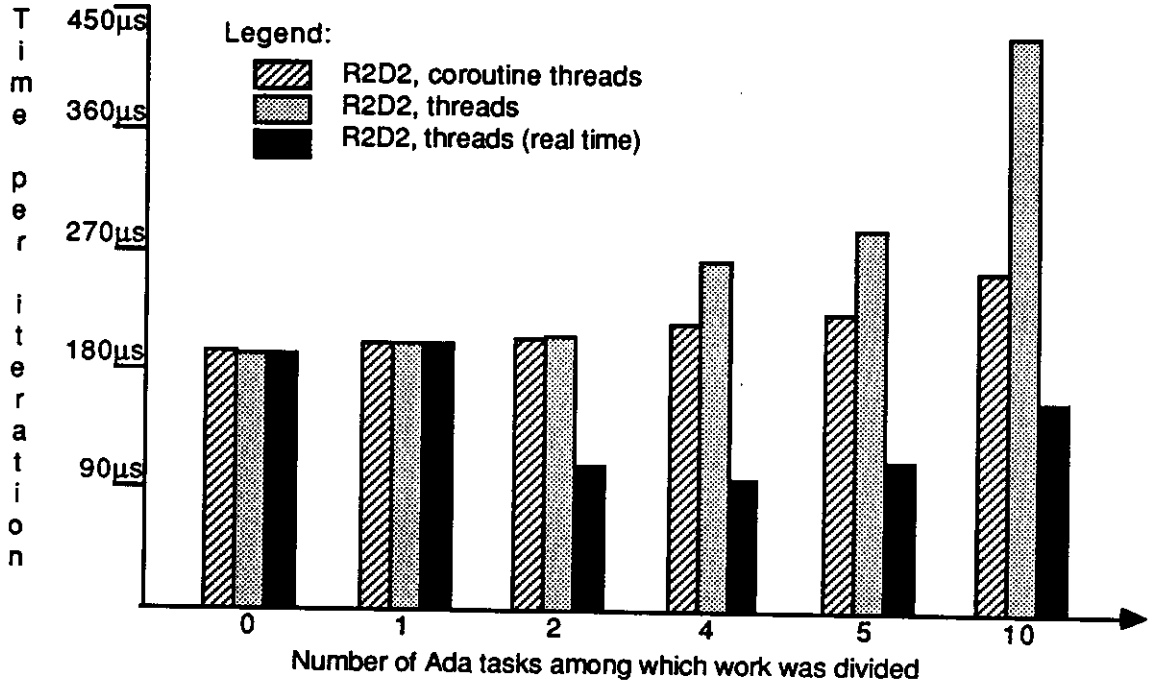

Figure 5-3: MULT - 5,000 iterations

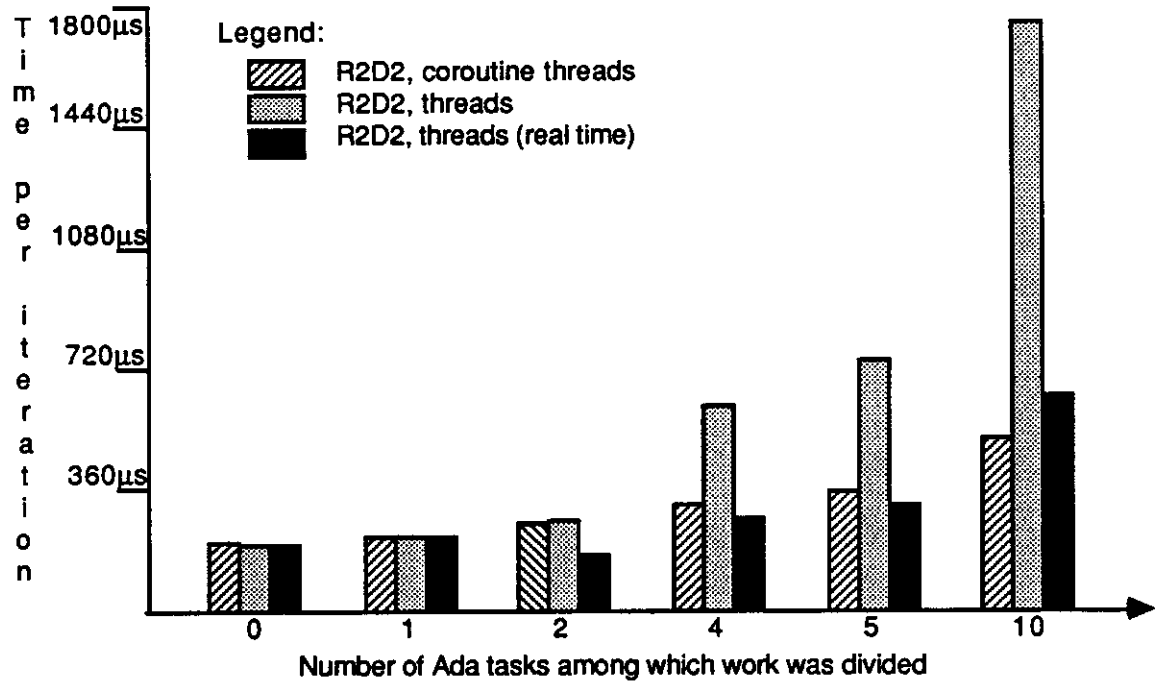

Figure 5-4: MULT - 2,500 iterations

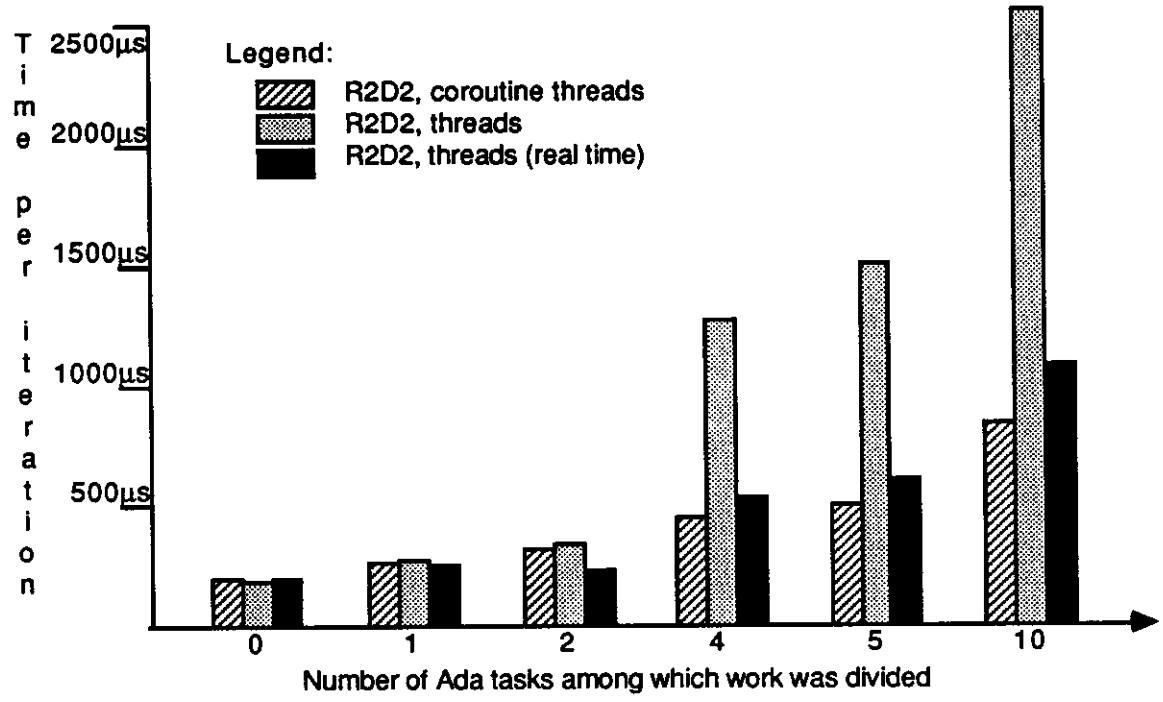

Table 5-3: Times per iteration to run the MULT tests on SPICE.CS.CMU.EDU

Total Iterations	Times for 0 tasks	Times for 1 task	Times for 2 tasks	Times for 4 tasks	Times for 5 tasks	Times for 10 tasks
100,000	250us	252us	259us	269us	274us	292us
25,000	253us	257us	269us	304us	324us	419us
5,000	264us	299us	360us	475us	558us	1005us
2,500	261us	341us	458us	695us	864us	1774us

Table 5-4: Times per iteration to run the MULT tests on R2D2.MACH.CS.CMU.EDU

Total Iterations	Times for 0 tasks	Times for 1 task	Times for 2 tasks	Times for 4 tasks	Times for 5 tasks	Times for 10 tasks
100,000	194us	194us	192us	198us	198us	206us
(threads)	192us	195us	194us	217us	226us	253us
(threads-real)	192us	195us	99us	62us	67us	79us
25,000	195us	200us	203us	218us	223us	255us
(threads)	192us	200us	205us	263us	286us	434us
(threads-real)	192us	200us	109us	99us	112us	159us
5,000	198us	231us	258us	320us	352us	514us
(threads)	195us	232us	268us	613us	748us	1752us
(threads-real)	196us	231us	165us	275us	317us	649us
2,500	198us	264us	320us	448us	512us	850us
(threads)	194us	267us	343us	1280us	1519us	2579us
(threads-real)	195us	264us	233us	536us	617us	1089us

The results seem to suggest that, at least under the coroutine version of C Threads, competition between a small number of tasks makes the average time per rendezvous somewhat longer, but not extremely so. In fact, the average times per rendezvous for Comp2.yield and Comp2.thread were better than those for their Comp1 counterparts. Under Mach threads, competing tasks suffered more severely; the time per rendezvous when five or ten tasks were competing was more than three times the time per rendezvous for the single-client case.

5.4. SELECT - cost of using select statements

To determine the cost of using select statements as opposed to simple accepts, several tests were run in which a calling task called an entry of a server in a tight loop. The body of the server task consisted of a loop whose body was either an accept statement or a select statement with one or more open accept alternatives. The tests were run for the case of an accept statement and for the cases of select statements with one, five, ten, and twenty alternatives, with the called entry placed either at the beginning or ending of the list of select alternatives. The test times are presented in Tables 5-7 and 5-8 and in Figure 5-6. They indicate that large select statements are expensive; the twenty-alternative select statements were from 50 to 91 percent more costly than simple accepts depending on the position of the accept alternative for the called entry and upon the environment under which they ran.

Figure 5-5: COMP

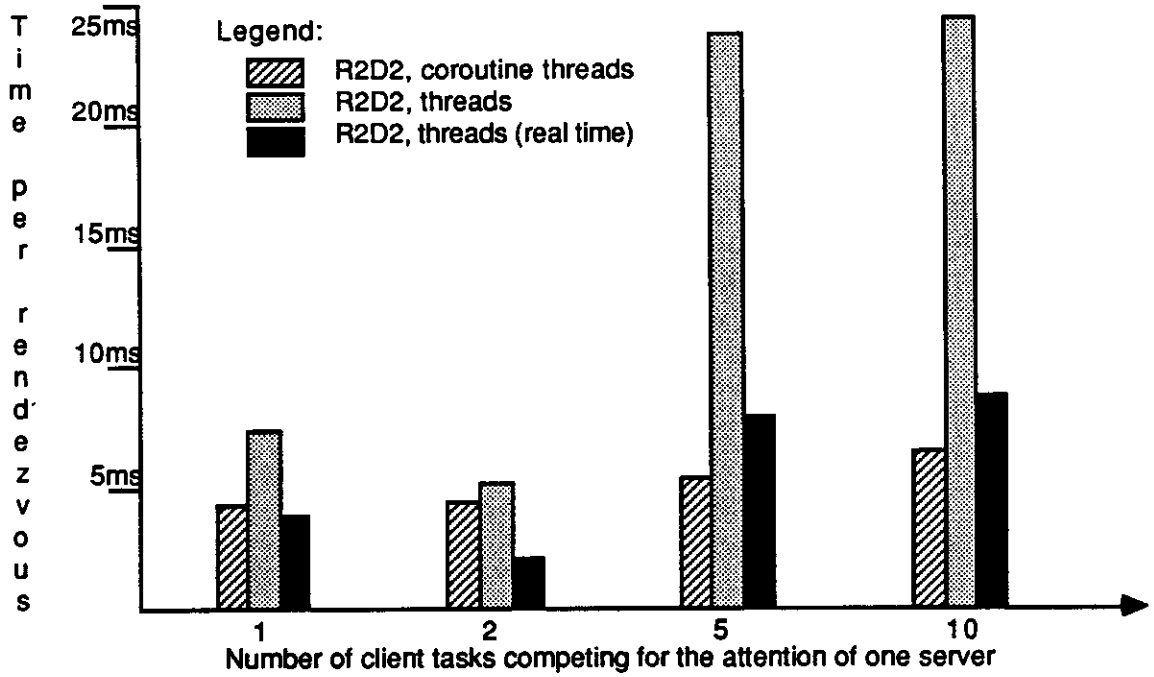

Figure 5-6: SELECT

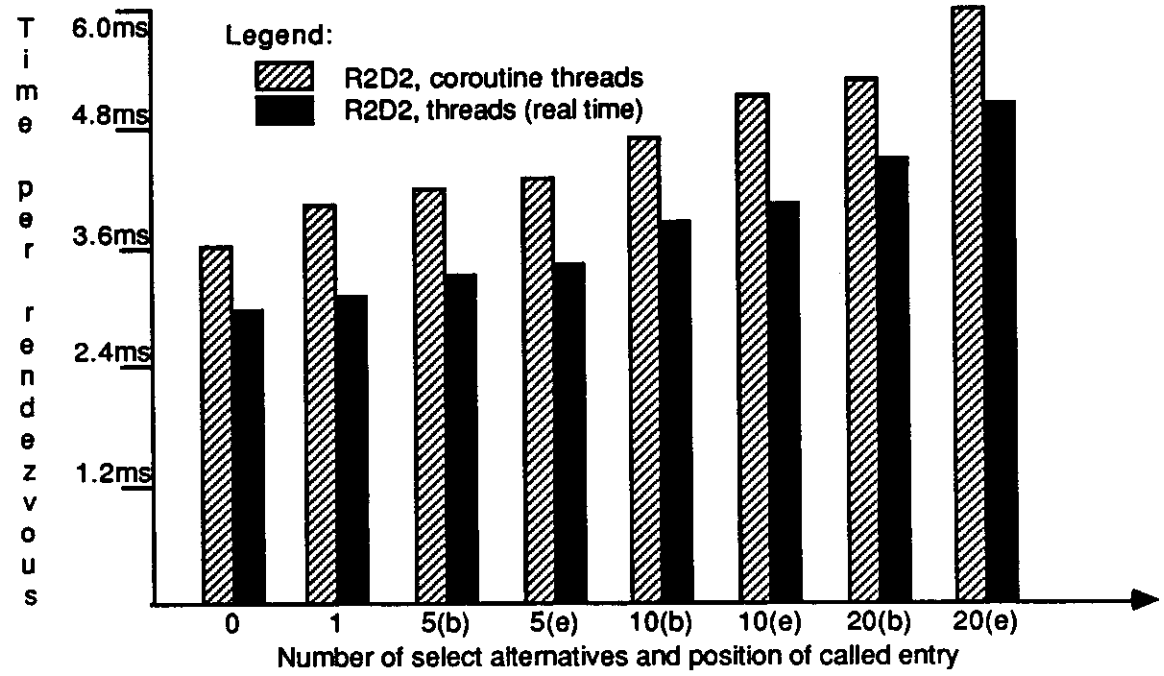

Table 5-5: Times to run the COMP tests on SPICE.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Rendezvous
Comp1.yield	4040ms	598ms	1.04/2.66	10768ms	4.6ms
Comp1.noyield	3682ms	524ms	1.00/1.14	5951ms	4.2ms
Comp2.yield	2402ms	704ms	2.66/2.01	6370ms	3.1ms
Comp2.noyield	3996ms	632ms	1.09/1.00	5435ms	4.6ms
Comp5.yield	2770ms	1552ms	2.01/1.76	6228ms	4.3ms
Comp5.noyield	4536ms	1604ms	1.00/2.45	8270ms	6.1ms
Comp10.yield	3024ms	3478ms	1.76/1.35	8080ms	6.5ms
Comp10.noyield	5482ms	3576ms	2.45/1.04	14200ms	9.1ms

Table 5-6: Times to run the COMP tests on R2D2.MACH.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Rendezvous
Comp1.yield	4334ms	258ms	2.84/2.98	4596ms	4.6ms
Comp1.noyield	4010ms	280ms	3.99/3.13	4334ms	4.3ms
Comp1.thread	4728ms	2690ms	2.95/1.86	3865ms	7.4ms
Comp2.yield	2430ms	390ms	2.98/2.90	2813ms	2.8ms
Comp2.noyield	4034ms	388ms	3.09/2.76	4423ms	4.4ms
Comp2.thread	3372ms	1860ms	1.90/2.38	2093ms	5.2ms
Comp5.yield	2754ms	814ms	2.90/2.47	3570ms	3.6ms
Comp5.noyield	4614ms	804ms	2.76/2.64	5417ms	5.4ms
Comp5.thread	6736ms	16976ms	2.38/1.05	7973ms	23.7ms
Comp10.yield	2792ms	1466ms	2.64/2.95	4259ms	4.3ms
Comp10.noyield	5080ms	1476ms	2.76/2.76	6533ms	6.6ms
Comp10.thread	7082ms	17420ms	1.70/1.61	8820ms	24.5ms

Table 5-7: Times to run the SELECT tests on SPICE.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Rendezvous	Cost vs. Select0
Sel0.yield	3086ms	426ms	1.01/1.00	3565ms	3.5ms	N/A
Sel0.noyield	2898ms	408ms	0.00/0.87	3415ms	3.3ms	N/A
Sel1.yield	3246ms	414ms	1.00/1.22	3792ms	3.7ms	+ 6%
Sel1.noyield	3026ms	422ms	0.87/0.98	3529ms	3.4ms	+ 3%
Sel5b.yield	3750ms	436ms	1.22/1.17	4258ms	4.2ms	+20%
Sel5b.noyield	3492ms	422ms	0.98/1.00	4257ms	3.9ms	+18%
Sel5e.yield	3924ms	434ms	1.17/1.24	4426ms	4.4ms	+26%
Sel5e.noyield	3656ms	420ms	1.00/0.66	4141ms	4.1ms	+24%
Sel10b.yield	4222ms	462ms	1.24/1.20	5585ms	4.7ms	+34%
Sel10b.noyield	4000ms	448ms	0.66/1.17	4663ms	4.4ms	+33%
Sel10e.yield	4626ms	480ms	1.20/1.02	5293ms	5.1ms	+46%
Sel10e.noyield	4394ms	452ms	1.17/1.08	4902ms	4.8ms	+45%
Sel20b.yield	5212ms	434ms	1.01/1.00	5689ms	5.6ms	+60%
Sel20b.noyield	4936ms	462ms	1.08/1.15	5464ms	5.4ms	+64%
Sel20e.yield	6054ms	448ms	1.00/1.09	6676ms	6.5ms	+86%
Sel20e.noyield	5800ms	460ms	1.15/1.01	6430ms	6.3ms	+91%

Table 5-8: Times to run the SELECT tests on R2D2.MACH.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Rendezvous	Cost vs. Select0
Sel0.yield	3524ms	284ms	2.33/2.91	3810ms	3.8ms	N/A
Sel0.noyield	3312ms	284ms	3.91/3.18	3603ms	3.6ms	N/A
Sel0.thread	3638ms	1918ms	3.00/2.30	2949ms	5.6ms	N/A
Sel1.yield	3882ms	286ms	2.91/3.05	4169ms	4.2ms	+11%
Sel1.noyield	3676ms	306ms	3.18/3.25	3988ms	4.0ms	+11%
Sel1.thread	4186ms	1738ms	2.30/2.53	3119ms	5.9ms	+ 5%
Sel5b.yield	4548ms	276ms	3.03/3.00	4819ms	4.8ms	+26%
Sel5b.noyield	3956ms	292ms	3.25/3.00	4267ms	4.2ms	+17%
Sel5b.thread	4204ms	2118ms	2.53/2.33	3328ms	6.3ms	+13%
Sel5e.yield	4376ms	268ms	3.00/3.00	4644ms	4.6ms	+21%
Sel5e.noyield	4068ms	276ms	3.00/3.00	4351ms	4.3ms	+19%
Sel5e.thread	4250ms	2282ms	2.55/2.11	3423ms	6.5ms	+16%
Sel10b.yield	4786ms	296ms	3.00/3.00	5082ms	5.1ms	+34%
Sel10b.noyield	4386ms	278ms	3.00/3.00	4663ms	4.7ms	+31%
Sel10b.thread	4754ms	2562ms	2.11/2.18	3827ms	7.3ms	+30%
Sel10e.yield	4850ms	286ms	3.00/3.00	5139ms	5.1ms	+34%
Sel10e.noyield	4784ms	292ms	3.00/3.00	5080ms	5.1ms	+42%
Sel10e.thread	5236ms	2628ms	2.18/2.58	4014ms	7.9ms	+41%
Sel20b.yield	5448ms	294ms	3.00/3.00	5735ms	5.7ms	+50%
Sel20b.noyield	5094ms	278ms	3.00/3.00	5375ms	5.3ms	+47%
Sel20b.thread	5386ms	3220ms	2.58/2.08	4469ms	8.6ms	+54%
Sel20e.yield	5990ms	280ms	3.00/3.00	6267ms	6.3ms	+66%
Sel20e.noyield	5710ms	270ms	3.00/3.23	6032ms	6.0ms	+67%
Sel20e.thread	6312ms	3470ms	2.38/2.30	5051ms	9.8ms	+75%

5.5. CHAIN - cost of forcing context switches among several tasks

As another method of determining the amount of time that a rendezvous takes, we ran tests in which two or more tasks, through the use of entry calls and accept statements, formed a control loop in which each task, upon receiving control of the processor, called its successor and immediately went back to sleep by waiting for an entry call. We ran tests for chains of two, three, four, five, ten, and twenty tasks, with control passing around the chain one thousand times in each test. The results, presented in Table 5-9 and Figures 5-7 and 5-8, show the cost per rendezvous to be about three milliseconds under the coroutine threads library. However, they also show the cost per rendezvous to be extremely high under the Mach threads library. While the three and four task tests were run under a different version of Mach than the other tests, which may account for their slightly higher times, the overall trend is still not good.

In an effort to determine the source of these high costs, we wrote a set of C programs to pass control around a chain of threads and timed their execution. Because we were not trying to duplicate Ada tasking semantics, but merely the behavior of the Ada chaining tests, the C programs execute much less code to perform synchronization than their Ada equivalents. The times the C programs took to run can be found in Table 5-10 and are also presented in Figures 5-7 and 5-8. As with the Ada tests, the coroutine times remained stable as the length of the chain increased, but the threads times became high very quickly. Several variant tests, including ones which looped 10,000 times around the chain, exhibited similar behavior. The only conclusion that can be drawn at this time is that the major cause of the high

costs for the Ada tests running under the Mach threads version of the C Threads package appears to be something unrelated to the Ada+ compiler.

6. Conclusion

This paper has attempted to describe how the Ada+ compiler implements tasking. The implementation is built on a set of lower-level facilities provided by the C Threads package, and makes heavy use of run-time system routines. Currently the Ada rendezvous constructs are implemented in a straightforward manner, but a Habermann and Nassi style implementation reusing many of the tasking "hooks" in the code generator appears possible. Using excerpts from the Ada+ run-time system and C pseudo-code, we have presented the run-time tasking interface and shown how it is used by compiled Ada code. We have also presented the results of running some performance benchmarks. We hope this report will prove of use to those who want to study the implementation of Ada tasking or who are trying to build their own implementations.

Figure 5-7: CHAIN - Ada and C chaining times under Mach threads

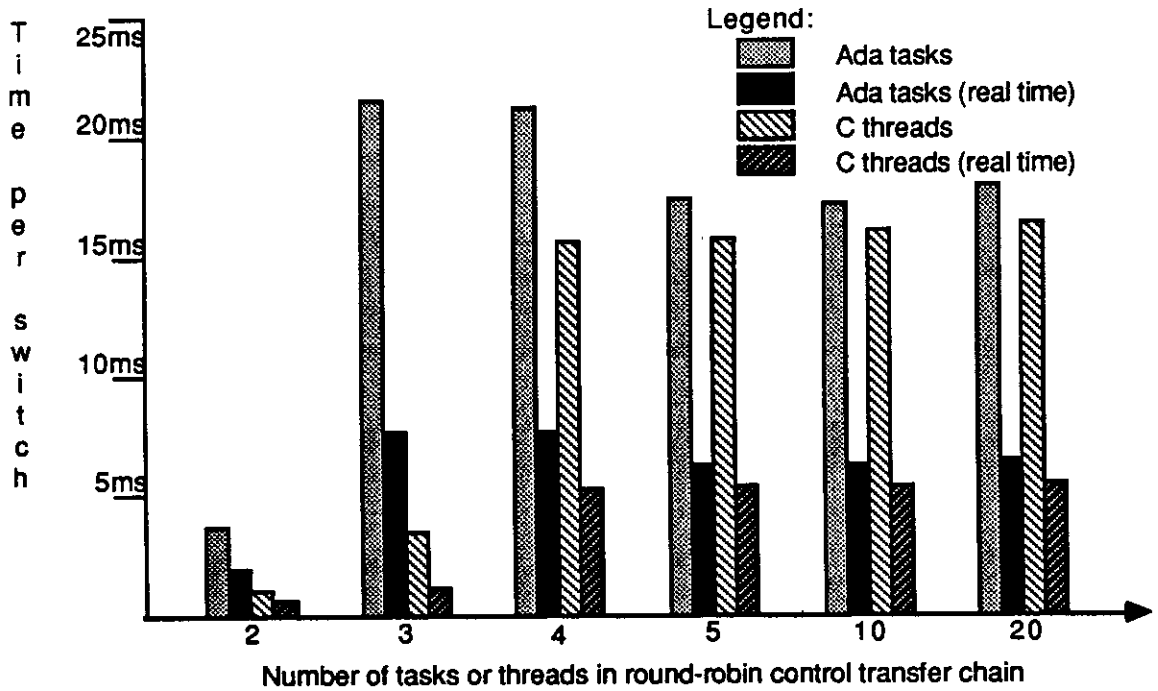

Figure 5-8: CHAIN - Ada and C chaining times under coroutine threads

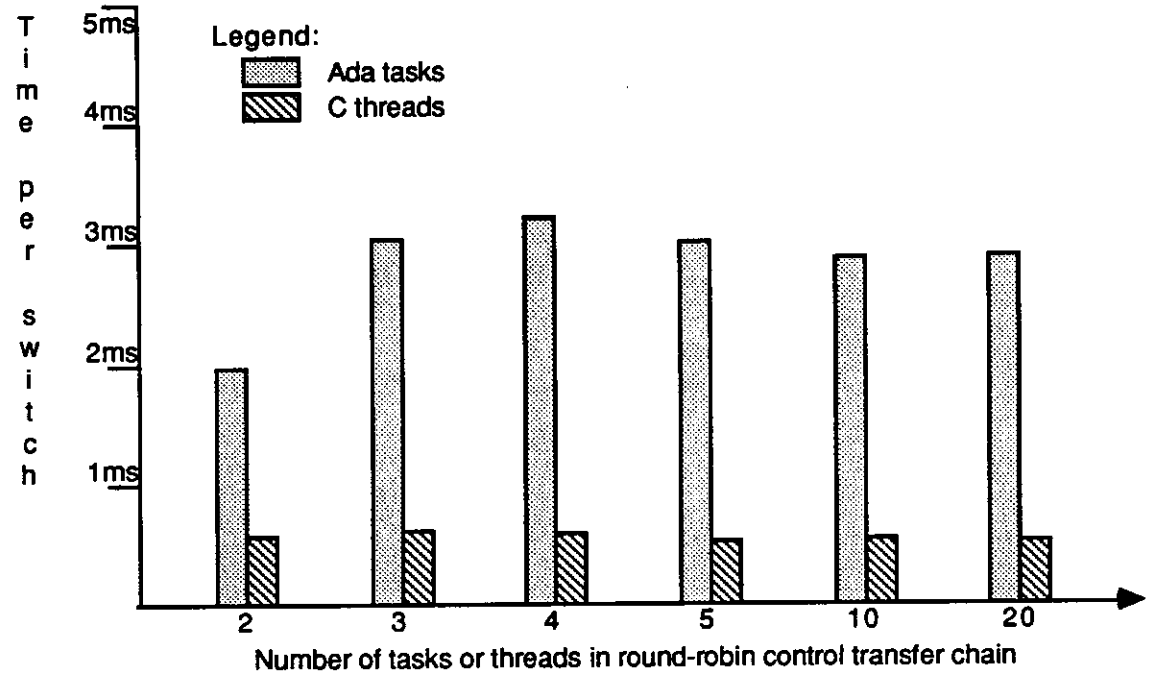

Table 5-9: Times to run the CHAIN tests on R2D2.MACH.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Rendezvous	Real time per Rendezvous
Ch2.noyield	3694ms	276ms	3.97/3.99	4024ms	1.99ms	2.01ms
Ch2.thread	5132ms	2380ms	4.00/4.00	3906ms	3.76ms	1.95ms
Ch3.noyield	8750ms	410ms	3.99/4.00	9171ms	3.05ms	3.06ms
Ch3.thread	23814ms	41026ms	4.00/4.00	23229ms	21.61ms	7.74ms
Ch4.noyield	12346ms	544ms	4.00/4.00	12899ms	3.22ms	3.22ms
Ch4.thread	31206ms	53940ms	4.00/4.00	30954ms	21.29ms	7.74ms
Ch5.noyield	14368ms	756ms	3.99/4.00	15261ms	3.02ms	3.05ms
Ch5.thread	32898ms	54922ms	4.00/4.00	31951ms	17.56ms	6.39ms
Ch10.noyield	27548ms	1414ms	4.00/4.00	29407ms	2.90ms	2.94ms
Ch10.thread	61344ms	111250ms	4.00/4.00	63089ms	17.26ms	6.31ms
Ch20.noyield	55720ms	2886ms	4.00/4.00	59220ms	2.93ms	2.96ms
Ch20.thread	133450ms	228430ms	4.00/4.00	131347ms	18.09ms	6.57ms

Table 5-10: Times to run the C chaining tests on R2D2.MACH.CS.CMU.EDU

Test	Elapsed User Time	Elapsed System Time	Start/End Load Factors	Elapsed Real Time	U+S per Switch	Real time per Switch
CCh2.noyield	1102ms	34ms	4.00/4.00	1137ms	0.57ms	0.57ms
CCh2.thread	1348ms	918ms	4.00/4.00	1140ms	1.13ms	0.57ms
CCh3.noyield	1780ms	46ms	4.00/4.00	1825ms	0.61ms	0.61ms
CCh3.thread	3718ms	6868ms	4.00/4.00	3719ms	3.53ms	1.24ms
CCh4.noyield	2278ms	70ms	4.00/4.00	2350ms	0.59ms	0.59ms
CCh4.thread	11468ms	51348ms	4.00/4.00	21319ms	15.70ms	5.33ms
CCh5.noyield	2520ms	94ms	4.00/4.00	2612ms	0.52ms	0.52ms
CCh5.thread	14232ms	65184ms	4.00/4.00	26873ms	15.88ms	5.37ms
CCh10.noyield	5462ms	168ms	4.00/4.00	5626ms	0.56ms	0.56ms
CCh10.thread	28808ms	133020ms	4.00/4.00	54819ms	16.18ms	5.48ms
CCh20.noyield	10214ms	358ms	4.00/4.00	10572ms	0.53ms	0.53ms
CCh20.thread	63486ms	266460ms	4.00/4.00	111544ms	16.50ms	5.58ms

References

[1] *American National Standard Reference Manual for the Ada Programming Language* ANSI/MIL-STD-1815A-1983 edition, 1983.

[2] Barbacci, M. R., Maddox, W. H., Newton, T. D., and Stockton, R. G.
The Ada+ Front End and Code Generator.
In *Proceedings of the Ada International Conference: Ada in Use.* ACM, Paris, France, May 1985.

[3] Cooper, E. C.
C Threads.
Technical Report, Computer Science Department, Carnegie-Mellon University, 1987.

[4] Digital Equipment Corporation.
VAX11 Architecture Handbook
1979.

[5] Habermann, A. N. and Nassi, I. R.
Efficient Implementation of Ada Tasks.
Technical Report CMU-CS-80-103, Computer Science Department, Carnegie-Mellon University, January, 1980.

[6] Newton, T. D.
An Implementation of Ada Generics.
Technical Report CMU-CS-86-125, Computer Science Department, Carnegie-Mellon University, May, 1986.

[7] PERQ Systems Corporation.
Pascal/C Machine Reference
1984.

[8] Stockton, R. G.
Overload Resolution in Ada+.
Technical Report CMU-CS-85-186, Computer Science Department, Carnegie-Mellon University, December, 1985.

[9] Tevanian, A. and Rashid, R. F.
MACH: A Basis for Future Unix Development.
Technical Report CMU-CS-87-139, Computer Science Department, Carnegie-Mellon University, June, 1987.