Corset and Lace

Adapting Ada* Runtime Support to Real-time Systems

T. P. Baker†
Department of Computer Science
Florida State University
Tallahassee, FL 32306
Kevin Jeffay
Department of Computer Science
University of Washington
Seattle, WA 98195

October 20, 1987

Abstract

Corset and Lace are runtime environment interfaces. Corset is an interface specification for a compact runtime support environment for tasking, for Ada. Lace is a specification for a low-level adaptable common executive that implements a model of real-time, lightweight tasks. These interfaces are designed to promote a wide range of implementations and extensions, tailored to the needs of diverse applications and hardware configurations, especially real-time embedded systems.

1 Introduction

The minimal language required by the Ada Standard [1] presents problems for use in building real-time systems. In particular, early Ada implementations are characterized by large size, high execution overhead, a lack of timing predictability, and a lack of control over the detailed resource management decisions on which correct timing and system reliability depend. Corset and Lace are two products of a program to produce implementations of Ada that are more suitable for programming real-time embedded systems.

Corset is an interface specification for a compact runtime support environment for tasking, for Ada. Lace is an interface specification for a low-level adaptable common executive, designed to support a Corset implementation of Ada tasking as well as light-weight real-time processes compiled as separate programs. The way in which Corset and Lace are intended to operate together to help Ada fit into constrained applications is illustrated in Figures 1 and 2.

Compiled Ada tasks and programs request Corset and Lace services via normal Ada procedure calls. This is intended to promote a wide range of implementations, tailored to the needs of diverse applications and hardware configurations.

Corset hides details of the runtime support environment (RSE) from the compiler, a necessary first step toward experimentation with specialized runtime support environments (RSEs). Lace, in turn, hides the details of processor allocation from the Ada RSE. This permits tailoring the dispatching policy to fit the application. In particular, the Lace interface is adaptable to implementation via a table-driven cyclic executive.

In addition to information hiding, Lace also plays another important role by supporting multiprogramming of simple Ada procedures without involvement of the Ada RSE, eliminating whatever inefficiency or unpredictability this may impose. Such multiprogrammed procedures can be executed alongside other tasks that make use of the full Ada RSE, so that hybrid systems can be constructed. Execution timing remains under control of the Lace dispatcher. By providing this means of cutting out what is perceived by many as being the most problematic part of Ada, Lace is intended to reduce the risk of using Ada for real-time applications.

Corset/I and Lace/I, described in [2] and [3] are designs for a prototype implementation designed to be compatible with a multiprocessor shared-memory configuration of the MIL-STD-1750A instruction set architecture. The 1750A is typical of most 16-bit microprocessors in not providing any special support for operating systems, beyond a simple test-and-set-bit instruction.

The Corset interface has evolved from the runtime system interface of the FSU/AFATL Ada compiler, a cross-compiler to the Z8002 microprocessor validated in 1985. The present versions of Corset and Lace are the outgrowth of a project whose objective is to improve the quality of Ada implementations for resource-constrained real-time applications.

Specific objectives of Corset and Lace include:

1. Defining a clear interface between the Ada runtime support environment (RSE) and compiler-generated code, such that:
   - efficient implementation is possible;
   - language features fall cleanly on one side or the other of the interface;
   - normal Ada procedure calls can be used to access RSE services from compiler-generated code;
   - no data structures are shared between RSE and compiler-generated code, so that all communication is by explicit parameters;
   - alternate RSE versions and extensions [4], suitable for real-time systems, can be implemented without special compiler support.
2. Decomposing the RSE and its data into subcomponents with well-defined interfaces between the subcomponents, in such a way that:
   - components can be omitted from the RSE for programs that do not use the features implemented by those components;
   - the RSE can be tailored to the needs of different applications and hardware configurations without modifying the interfaces between RSE components.
   - the interface is implementable for various processor configurations, as well as modifications to the tasking paradigm.
3. Experimenting with alternative data structures and algorithms for the implementation of Ada tasking, with a view to:
   - the limitations of existing microprocessor architectures;
   - decentralized algorithms, which can be executed by multiple processors concurrently, with little or no waiting;
   - multiprogramming of Ada procedures which have no internal tasks without the overhead of supporting tasking;
   - keeping overhead down, and in particular to avoid unnecessary saving and restoring of task state both during context switches and in interrupt handlers;
   - predictable performance, through the use of simple, deterministic algorithms where possible (e.g. design the dispatcher with a minimum number of locks and so that interrupts never have to be disabled);
   - localized cost, so that performance of simpler functions (e.g. rendezvous) is not be degraded because of side effects of implementing more complex functions, even if this means making other operations (e.g. task abortion) more costly.

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

†This work supported in part by Boeing Aerospace Company and the Washington Technology Center.

CH2475-2/87/0000/0158$01 00 © 1987 IEEE

- *limiting capacities (e.g . number of tasks, range of priorities)* to obtain greater efficiency or predictable timing.

The rest of this report describes the Corset and Lace interfaces, and the major internal component interfaces of the Corset/I implementation.

2 Corset Interface

The interface between Corset and compiler-generated code is divided into several views, comprising;

1. Corset as seen by compiled code;
2. the compiled code, as seen by Corset;
3. Corset, as seen by an interrupt handler;
4. Corset, as seen by the implementation of package CALENDAR.

We will consider each of these, in turn.

2.1 The Compiler's View

The interface presented by Corset to compiled code is described by the following virtual package.

```
with SYSTEM;
package RTE_PROVIDES is
  type TASK_ID is private;
  NULL_TASK: constant TASK_ID;
  type TASK_LIST is array (POSITIVE range <>) of TASK_ID;
  type COLLECTION_ID is private;
  function NEW_COLLECTION(COLLECTION_SIZE: NATURAL:=0;
                          MAX_BLOCK_SIZE: NATURAL:=0)
                          return COLLECTION_ID;
  function NEW_BLOCK(SIZE: NATURAL;
                     C: COLLECTION_ID) return SYSTEM.ADDRESS;
  procedure RELEASE_BLOCK(LOC: SYSTEM.ADDRESS);

  procedure RELEASE_COLLECTION(C: COLLECTION_ID);
  procedure DELAY_SELF(D: DURATION);
  type ENTRY_INDEX is private;
  NULL_ENTRY: constant ENTRY_INDEX;
  type ENTRY_BINDING is
       record INDEX: ENTRY_INDEX;
              PROC:  SYSTEM.ADDRESS;
       end record;
  type ENTRY_BINDING_LIST is array (POSITIVE range <>)
                             of ENTRY_BINDING;
  NULL_RENDEZVOUS:      constant INTEGER:= 0;
  RENDEZVOUS_COMPLETED: constant INTEGER:= 1;
  SIMPLE_MODE:          constant INTEGER:= 0;
  DELAY_MODE:           constant INTEGER:= 1;
  ELSE_MODE:            constant INTEGER:= 2;
  TERMINATE_MODE:       constant INTEGER:= 3;
  subtype MODES is INTEGER range SIMPLE_MODE..TERMINATE_MODE;
  procedure ACCEPT_CALL(E: ENTRY_INDEX;
                        PROC: out SYSTEM.ADDRESS);
  procedure CALL_SIMPLE(A: TASK_ID;
                        E: ENTRY_INDEX;
                        LOC: SYSTEM.ADDRESS;
                        SIZE: NATURAL);
  procedure CALL_CONDITIONAL(A: TASK_ID;
                             E: ENTRY_INDEX;
                             LOC: SYSTEM.ADDRESS;
                             SIZE: NATURAL;
                             RESULT: out INTEGER);
  procedure CALL_TIMED(A: TASK_ID;
                       E: ENTRY_INDEX;
                       LOC: SYSTEM.ADDRESS;
                       SIZE: NATURAL;
                       D: DURATION;
                       RESULT: out INTEGER);
  function  COUNT(T: TASK_ID; E: ENTRY_INDEX) return NATURAL;
  function  CALLABLE(T: TASK_ID) return BOOLEAN;
  procedure SELECTIVE_WAIT(OPEN_ENTRIES: ENTRY_BINDING_LIST;
                           D: DURATION;
                           SELECT_MODE: MODES;
                           RESULT: out INTEGER);
  procedure ABORT_TASKS(TASKS: TASK_LIST);
  function MIN_PRIORITY return INTEGER;
  function MAX_PRIORITY return INTEGER;
  subtype PRIORITY is INTEGER range
                              MIN_PRIORITY..MAX_PRIORITY;
  type MASTER_ID is private;
  NULL_MASTER: constant MASTER_ID;
  procedure ACTIVATE_TASKS(TASKS: TASK_LIST);
  procedure COMPLETE_ACTIVATION;
  procedure COMPLETE_TASK;
  procedure CREATE_TASK(SIZE: NATURAL;
                        PRIO: PRIORITY;
                        INIT_STATE: MACHINE.STATE;
                        M: MASTER_ID;
                        NUM_ENTRIES: NATURAL;
                        CREATED_TASK: out TASK_ID);
  function  CURRENT_MASTER return MASTER_ID;
  procedure ENTER_MASTER;
  procedure EXIT_MASTER;
  function  IS_LOCAL_TASK(T: TASK_ID;
                          M: MASTER_ID) return BOOLEAN;
  function  TERMINATED(T: TASK_ID) return BOOLEAN;
private ...
end RTE_PROVIDES;
```

In Corset/I the declarations above are implemented by several packages rather than one. These packages also export other declarations, not part of the Corset interface, to the Corset implementation components.

The rationale for this interface is explained at some length in a separate report [5]. The semantics are explained in later sections of this report.

In addition to the operations above, which are needed to support the standard Ada semantics, a Corset implementation may provide extended features. One such example is offline entry calls, a feature which appears needed for communication between time-constrained tasks and other tasks. The difference between on offline call and a normal entry call is that the calling task does not wait. This capability can be described ideally by the procedure OFFLINE_CALLS, below:

```
procedure CALL_OFFLINE(A: TASK_ID;
                       E: ENTRY_INDEX;
                       LOC: SYSTEM.ADDRESS;
                       SIZE: NATURAL);
```

Procedure CALL_OFFLINE makes a non-waiting call on entry E of task A. The parameters LOC and SIZE are the address and size of a buffer containing the parameters to the entry call. Ownership of this buffer is relinquished by the caller. That is, it should not attempt to access this record again, unless arrangements are made for it to be explicitly returned (later) by the caller.

2.2 Corset's View

In order to implement Corset, some things must be known about the compiler:

1. the exception-raising protocol (i.e. how the *raise* statement is implemented by the compiler);
2. the procedure-calling protocol;
3. how the address of the workspace allocated to the task is passed to the task initialization code.

The package COMPILER_PROVIDES (below) provides some of this information.

```
with SYSTEM;
package COMPILER_PROVIDES is
  type EXCEPTION_ID is private;
  ABORTION:          constant EXCEPTION_ID;
  NULL_EXCEPTION:    constant EXCEPTION_ID;
  CONSTRAINT_ERROR:  constant EXCEPTION_ID;
  NUMERIC_ERROR:     constant EXCEPTION_ID;
  PROGRAM_ERROR:     constant EXCEPTION_ID;
  STORAGE_ERROR:     constant EXCEPTION_ID;
  TASKING_ERROR:     constant EXCEPTION_ID;
  procedure RAISE_EXCEPTION(E: EXCEPTION_ID);
  function  CURRENT_EXCEPTION return EXCEPTION_ID;
  procedure CALL(PROC, PARAM: SYSTEM.ADDRESS);
  procedure FAKE_CALL(PROC,RET: SYSTEM.ADDRESS);
  procedure RETURN_AND_CALL(PROC: SYSTEM.ADDRESS);
private ...
end COMPILER_PROVIDES;
```

Corset is not concerned with the implementation of exception propagation and handling within tasks. This is assumed to be a responsibility of the compiler. Most of what Corset needs to know about exception handling is contained in the package above.

However, any Corset implementation does need to make some assumptions about how exception handling is implemented. For Corset/I, the chief dependency on the compiler's treatment of exceptions can be found in procedure FORCE_CALL of Lace (described further below). We assume that a forced call can be imposed on a task at any point where it is preemptible. This is more or less equivalent to assuming that an exception can be raised asynchronously at any point during the execution of compiled code. Thus the compiler should not assume that any section of code (e.g. changing of exception handling context) is safe from having an exception raised during its execution, as it might be in response to a hardware interrupt.

In order to provide more freedom to the Corset implementation we would also like the compiler to support a predefined pseudoexception, "ABORTION". This should be treated like an ordinary exception except in two respects: (1) it cannot be handled by an "others" clause; (2) it is not be visible to an ordinary application program, so that it is only handleable by the RSE implementation. Though the Corset/I implementation does not make use of this feature, it appears that having this pseudoexception would simplify the implementation of task abortion.

A Corset implementation needs to know how procedures are called in order to implement entry calls. The compiler is assumed to generate the code for each accept statement as a procedure. The actual parameters of an entry call are assumed to be laid out in a contiguous block and the code of the entry procedure body must access this block indirectly, through a single parameter of type ADDRESS. (Of course if a parameter is passed by reference the block will contain the address of the actual parameter, rather than its value.)

The package above includes three procedures that provide procedure-calling capabilities used by the Corset (and Lace) implementations. Procedure CALL does a procedure call to address PROC with parameter block at address PARAM. Procedure FAKE_CALL calls a parameterless procedure at address PROC but arranges for the return to be to address RET, rather than the location from which the call actually originates. This should result in both normal return and exception propagation being to this substitute return address. Procedure RETURN_AND_CALL calls a parameterless procedure at address PROC after first returning from the procedure where it is called, so that the new call effectively replaces the current procedure activation.

In addition to the information above, it is helpful if the RSE implementation can predict that certain registers need not be saved in the event of a voluntary task switch.

2.3 An Interrupt Handler's View

Corset assumes that exception handlers do not call any of the procedures in package RTE_PROVIDES (above), directly or indirectly. Facilities for interrupt handlers to communicate with tasks are provided by Lace. In particular, the following Lace operations are intended for use in interrupt handlers.

```
function  PRIORITY(T:  TASK_ID) return INTEGER;
function  SELF return TASK_ID;
function  PREEMPTION_OK return BOOLEAN;
procedure PREEMPT;
procedure RELEASE(T:  TASK_ID);
```

An example of how these operations might be used in an interrupt handler is given below. The task T that is released by the handler is presumably one that consumes data brought in by this handler. Note that the handler need not be necessarily compiled as an Ada task; it could also be written in assembly language.

```
task HANDLER is
begin loop accept INTERRUPT do
          -- Put some data in a buffer for task T.
          RELEASE(T);
          if PREEMPTION_OK then PREEMPT;
               -- Will not return
          end if;
          end INTERRUPT;
     end loop;
end HANDLER;
```

Implicit in the example are the saving of whatever registers are used in the accept statement, the disabling of the interrupt, the restoring of registers on normal completion, and the resumption of the interrupted task.

2.4 Calendar's View

The Corset implementation requires an internal clock to implement delays. The predefined package CALENDAR is not suitable for this because:

1. the type TIME is required to have too great a range to be representable in 32 bits at the accuracy required for a real-time application;
2. the type DURATION is in seconds, which may require costly conversions if the hardware clock does not count in binary fractions of a second (e.g. the MIL-STD-1750A timers are calibrated in decimal fractions of a second).

The Corset/I implementation is therefore based on a cyclic clock that counts in TICKs. For convenience in implementing package CALENDAR (which is not part of Corset) consistently with Corset's internal clock, Corset exports the following types and operation.

```
type DAYS is range 0..(2**31)-1;
type DURATION is delta 2.0**(-13)
     range -2.0**18..(2.0**18)-1.0;
subtype DAY_DURATION is DURATION range 0.0 .. 86_400.0;
type TIME is record DAY: DAYS;
                    SECOND: DAY_DURATION;
               end record;
TICK: constant:= 0.0001;
function CLOCK return TIME;
```

The constraints of type DURATION are chosen to permit implementation of delays using a 32-bit integer clock, counting time in units of TICK, without loss of information when the clock rolls over to zero. In particular, we require that DURATION'LAST be no more than half of the clock period.

The requirements that Corset imposes on the compiler are that the representation and constraints of STANDARD.DURATION and the value of SYSTEM.TICK must agree with those used by Corset.

As with the package RTE_PROVIDES the actual declarations above are implemented in Corset/I's internal package CLOCK, but would be made available to an application via renamings in a higher-level package specification.

3 The Lace Interface

Lace is intended not only to support Corset, but to also serve directly as an executive for multiprogrammed applications without tasks. It is based on a simpler (and cheaper) model of tasks than Ada. To Lace, a task is simply an entry point into a executable code, a workspace, and a priority. Thus Ada tasks may be intermixed with Ada procedures invoked as independent programs, on the same processor(s).

Lace only provides for allocation of processors. In particular, it does not provide directly for intertask communication or memory management. Such services are provided separately, possibly using the Lace operations in their implementation.

The interface to Lace is a set of procedures that are invoked by user programs that wish to either execute as Lace tasks or manipulate other Lace tasks. Although not necessarily implemented as procedures, Lace operations have the property that, barring exceptional conditions such as abortion, they will eventually return to the user program at the point following the location of the call. The interfaces between Lace and user code can be categorized into three views analogous to those for Corset:

1. Lace as seen by the compiled user code;
2. the compiled code, as seen by Lace;
3. Lace, as seen by an interrupt handler.

Since Lace and Corset are designed to be used together, Views 2-3 are the same as for Corset, and have been discussed above. We will consider the view of Lace by user code in more detail. The interface presented by Lace to a compiled user program is described by the packages TASK_IDS and LACE.

3.1 Task Identification and Attributes

To the compiler-generated code, a Lace task is a value of type TASK_ID, representing a thread of control.

```
package TASK_IDS is
  type TASK_ID is range 0..127;
  NULL_TASK: constant TASK_ID:= TASK_ID'first;
  type TASK_LIST is array (POSITIVE range <>) of TASK_ID;
end TASK_IDS;
```

The type TASK_ID is used by Lace and Corset, both internally and as part of their external interfaces. Because this package is also part of the Lace and Corset internal interfaces, the full type declaration is given. However, for external interface purposes, all that should be assumed about this type is that it has a certain size; that is, the range is not part of the interface, nor is the value of NULL_TASK.

The state of a Lace task is characterized by and four Boolean attributes:

- allocated - the task is a valid Lace task;
- held - the task is not eligible for execution on a processor;
- executing - the task is executing on a processor;
- preemptible - task may be preempted on its processor by a higher priority Lace task.

A task also has a priority, within a range determined by the Lace implementation.

```
with SYSTEM;
with TASK_IDS; use TASK_IDS;
package LACE is
  MAX_PRIORITY, MIN_PRIORITY: INTEGER;
```

3.2 Preemptible Procedures

The following operations are required to be implemented so that they do not require waiting for any locks, and so may be called from interrupt handlers, as well as from tasks. Also, since they do not hold locks they may be safely executed in preemptible mode.

The first group of operations are functions that return attributes of a task. Care must taken in using these functions since the attribute of the task queried may be change concurrently with the query. These attributes can still be useful, however, especially when called by a task to determine an attribute of itself.

```
  function  SELF return TASK_ID;
  function  PREEMPTION_OK return BOOLEAN;
  function  HELD(T: TASK_ID) return BOOLEAN;
  function  EXECUTING(T: TASK_ID) return BOOLEAN;
  function  PRIORITY(T: TASK_ID) return INTEGER;
  function  VALID(T: TASK_ID) return BOOLEAN;
```

Function SELF returns the ID of the task which calls it, except if called from a hardware interrupt handler, in which case it returns the ID of the task which was executing when the interrupt occurred. Function PREEMPTION_OK returns true if and only if the processor on which it is called is currently preemptible. Function HELD returns true if and only if task T is held. Function VALID returns true if and only if T is a currently assigned task ID. Function EXECUTING(T) returns true if and only if the task T is currently executing on some processor. Function PRIORITY returns the current Lace priority of task T.

The second group of preemptible operations modify the state of a task.

```
  procedure DISABLE_PREEMPTION;
  procedure HOLD(T: TASK_ID);
  procedure RELEASE(T: TASK_ID);
```

Procedure DISABLE_PREEMPTION makes the processor from which it is called not be preemptible from the current task by the dispatcher. The effect is undone (only) by procedure DISPATCH (below) Procedure HOLD disables dispatching of the task T. If T is running it will continue until it next calls DISPATCH or is preempted by an interrupt. Procedure RELEASE enables dispatching of task T, undoing the effect of any previous HOLDS. If T is not held there is no change. How soon T gets to run depends on its priority and the activities of higher priority tasks.

3.3 Nonpreemptible Procedures

Because the implementation of the following procedures is anticipated to possibly involve waiting for and holding locks we require that before they are called the processor be made nonpreemptible. Failure to follow this rule will lead to system failure. They should never be called from an interrupt handler.

```
  procedure SET_PRIORITY(T: TASK_ID; P: INTEGER);
  procedure DEFER_TO(T: TASK_ID);
  procedure UNDEFER(T: TASK_ID);
  procedure DISPATCH;
  procedure FORCE_CALL(T: TASK_ID; P: SYSTEM.ADDRESS);
  function  NEW_TASK(S: MACHINE.STATE;
                     P: INTEGER) return TASK_ID;
  procedure COLLECT_ID(T: TASK_ID);
```

Procedure SET_PRIORITY sets the dispatching priority of T to P, if P is a priority value supported by the Lace implementation. Otherwise, the priority is set to the nearest supported value. A task with a numerically larger priority number always has preference for dispatching. The caller of this procedure is responsible for insuring that it is never called concurrently for a single task T by more than one task.

Procedure DISPATCH is called to choose the next task to execute on the processor on which it is called, and transfers control to this task (which may be the calling task, if that is not held). If there is no task eligible for execution, the dispatcher idles. Though it may switch

control to another task, from the point of view of the caller this procedure behaves like any other procedure. That is, it eventually returns to the instruction following the call (barring exceptions). Calling the dispatcher is the normal means of return from service routines of an RSE built on Lace.

Note that DISPATCH is the only way to undo the effect of DISABLE_PREEMPTION. This is because interrupts that arrive during a nonpreemptible period may cause the release of tasks which should be executed as soon as the processor becomes preemptible.

Procedure FORCE_CALL forces task T to call a procedure at address PROC, with parameter value PARAM. This procedure can be used to raise exceptions in tasks and to abort them, by forcing them to execute appropriate procedures. The effect is does not take effect until T is next dispatched. If FORCE_CALL is called again for the same task before a pending forced call has taken effect, the later call will override. The practice of overriding pending forced calls should be avoided, because how soon a call will take effect cannot in general be predicted. (Also to be avoided are nested calls, which may cause space problems as noted below.)

Care must be taken that every task has enough excess work space to tolerate any forced calls. Of course this is already true of other RSE calls, but it is especially critical here because of the interaction with exception recovery. If STORAGE_ERROR is raised and if exception recovery is through such a forced call, if the exception recovery procedure uses storage, cycling could result. The obvious solution is to write all such routines to use only statically allocated global data and registers, or at least to use a predictable bounded amount of stack space.

Function NEW_TASK allocates a new TASK_ID and returns it. It returns NULL_TASK iff it cannot allocate a new task ID. The new task starts out in the held state, and its register state is initialized to S. The priority of a new task is MIN_PRIORITY. In order for the new task to begin execution it must later be released.

Procedure COLLECT_ID deallocates the task ID T. It assumes that T is not executing or contending for a processor (i.e. it is the caller's responsibility to insure this). In particular, it assumes T is not the caller.

Other than the functions VALID, all operations with arguments of type TASK_ID require that the ID be valid. That is, it must have been returned by NEW_TASK and COLLECT_ID must not have been called for it since then.

3.4 Interrupt Handler Support

The procedure below is provided to support writing of hardware interrupt handlers.

```
  procedure PREEMPT;
end LACE;
```

This procedure preempts the current task, and calls the dispatcher. This should never be called unless PREEMPTION_OK returns TRUE, otherwise chaos will ensue. It should only be called from an interrupt handler, since it presumes that interrupts are already disabled, and does not return to the point of call. It should only be called at the conclusion of the handler (at the point of return) and assumes the handler has saved the state of the task T to be preempted in a location defined by machine-specific convention. The processor will be preempted from the current task and the dispatcher will be invoked to determine which task executes next.

4 Other Corset/I Internal Interfaces

Corset/I is divided internally into the following major packages:

1. SYSTEM is the standard predefined package, with implementation-specific extensions as permitted by the Ada standard [1]. It is imported by Corset.
2. TASK_IDS (described above) exports type TASK_ID, constant NULL_TASK, and some related types to all other Corset components.
3. LACE (described above) exports low-level executive functions to other Corset components, some of which are also exported to handlers.
4. TASK_DATA exports common per-task types and data structures that are shared between the Corset components.
5. ABORTION exports operations used in Corset to implement task abortion, plus some encapsulated versions of basic Lace dispatching operations with special interlocks to prevent unsafe interaction with abortion.
6. COLLECTIONS exports operations which are called by compiler-generated code to reserve storage collections and allocate and deallocate blocks within those collections. The collections reserved by a task and not released explicitly are implicitly released when the task is terminated, via a procedure exported to package STAGES.
7. CLOCK exports types and operations used internally by the Corset implementation for timing, some of which are also exported to the application level.
8. DELAYS exports procedure DELAY_TASK, which is called by compiler-generated code, and other variables and operations used internally by the Corset implementation to implement delays. One of these is a procedure that must be called periodically to check for expired delays, preferably at time DELAYS.TIME_OF_NEXT_CHECK. It is used by package RENDEZVOUS to implement timed entry calls and selective waits with delay alternatives.
9. CALLS exports types and operations used to maintain queues of entry calls.
10. OFFLINE_CALLS exports operations and types to support offline entry calls.
11. RENDEZVOUS exports types and operations required to implement rendezvous.
12. STORAGE_MANAGEMENT exports operations used to allocate and deallocate workspaces for tasks.
13. STAGES exports types and operations required to implement task creation, activation, completion, and normal termination.

Note that this modular breakdown assists configuration of a minimal RSE for a particular application program. First, those procedures within each package that are not called by the program need not be linked. Second, entire packages whose functionality is not required for the program may be omitted, provided references from other Corset components are stubbed off. Finally, alternate (simplified) versions of the remaining components may be substituted, to take advantage of the unused features. For example, if the procedure ABORT_TASKS is not called from a program we can replace the bodies of packages ABORTION, RENDEZVOUS, and STAGES by grossly simplified versions.

4.1 Rights

The Corset/I and Lace/I implementations assume the existence of some low-level interprocessor synchronization operations, which can be implemented using a test-and-set instruction. These are abstracted by the package RIGHTS, below. A "right" is a lock variable which a task may attempt to claim, or for which it may wait.

```
package RIGHTS is
  type RIGHT is private;
  AVAILABLE: constant RIGHT;
  procedure AWAIT(R: in out RIGHT);
  procedure POST(R: in out RIGHT);
  procedure CLAIM(R: in out RIGHT; BUSY: out BOOLEAN);
  function "not"(R: in RIGHT) return RIGHT;
private
  type RIGHT is new BOOLEAN;
  AVAILABLE: constant RIGHT:= FALSE;
end RIGHTS;
```

4.2 Collections

Package COLLECTIONS provides support for the implementation of Ada allocators. Procedure NEW_COLLECTION reserves SIZE addressable units of storage for a new collection, and returns an ID for it. The storage is reserved in the name of the calling task, at the current level of master nesting. It will be deallocated automatically no later than when the task is terminated. Procedure NEW_BLOCK allocates a contiguous block of SIZE storage units within the space reserved for collection C, and returns the low address of this block. Procedure RELEASE_BLOCK releases the block at address LOC, and procedure RELEASE_COLLECTION releases the collection C.

```
with SYSTEM;
with TASK_IDS;
package COLLECTIONS is
  type COLLECTION_ID is private;
  NULL_COLLECTION: constant COLLECTION_ID;
  function NEW_COLLECTION(COLLECTION_SIZE: NATURAL:=0;
                          MAX_BLOCK_SIZE: NATURAL:=0)
                          return COLLECTION_ID;
  function NEW_BLOCK(SIZE: NATURAL;
                     C: COLLECTION_ID)
                     return SYSTEM.ADDRESS;
  procedure RELEASE_BLOCK(LOC: SYSTEM.ADDRESS);
  procedure RELEASE_COLLECTION(C: COLLECTION_ID);
  procedure RELEASE_COLLECTIONS(T: TASK_ID);
private
  type COLLECTION_ID is range 0..(2**15)-1;
  NULL_COLLECTION: constant COLLECTION_ID:= 0;
end COLLECTIONS;
```

4.3 Clock

Package CLOCK provides a primitive form of real-time clock, that counts time cyclically, in ticks. This can be used for implementing more complex forms of time-keeping, such as the Ada standard calendar package, or can be used directly, as it is by the Corset implementation.

```
package CLOCK is
  type DAYS is range 0..(2**31)-1;
  type DURATION is delta 2.0**(-13)
                range -2.0**18..(2.0**18)-1.0;
  subtype DAY_DURATION is DURATION range 0.0 .. 86_400.0;
  type TIME is record DAY: DAYS;
                      SECOND: DAY_DURATION;
               end record;
  TICK: constant:= 2.0**(-12);
  type TICKS is private;
  function CLOCK return TICKS;
  function CLOCK return TIME;
  function CLOCK_IS_PAST(T: TICKS) return BOOLEAN;
  function "+"(L: TICKS; R: INTEGER) return TICKS;
private
  type TICKS is range -(2**31)..(2**31-1);
end CLOCK;
```

The constraints on the type DURATION declared here and that in STANDARD must agree. We require that DURATION'last*2 $\leq$ TICK*TICKS'last. This will avoid loss of information when the value of CLOCK rolls over to zero. We must be able to represent durations of one full day, which is 86,400 seconds. Since $2^{16} < 86,400 < 2^{17}$, we want DURATION'last $\geq 2^{17}$, which implies TICK*TICKS'last $\geq 2^{18}$. We want to be able to represent TICKS in 32 bits, so TICKS'last $< 2^{31}$. If follows that TICK $> 2^{-13}$. We also want to be able to request delays down to a granularity of TICK, so we would like DURATION'small*2 $\leq$TICK. If $2^{-13} <$ TICK $\leq 2^{-12}$, we want DURATION'small $\leq 2^{-13}$. Fortunately all these requirements are consistent with a 32-bit signed integer implementation of DURATION.

Function CLOCK returning TICKS should return the number of ticks that have elapsed since the clock was last (re)set to zero. It can be implemented as a global variable. This value should be incremented every TICK seconds. One way in which this might be done is via a periodic interrupt. It should be reset ONLY when it rolls over; that is, on the tick after TICKS'LAST. CLOCK should be monotonic. That is, it should never be set backward. The effect of setting it backward relative to another time-reference can be obtained by slowing it down (skipping ticks), if necessary.

Function CLOCK returning TIME should implement a real-time clock consistent with the function CLOCK returning TICKS.

The type TIME is defined to support the full range of times required by the Ada standard. Unfortunately, this cannot be represented in a single word. We therefore represent is as record, with the value (0, 0.0) representing the time the clock was started. Since fetching two words is not an atomic hardware operation, obtaining the value of CLOCK that returns TIME needs to be protected by a protocol to insure that is not read while being updated. On the other hand, enforcing such a protocol is costly in time. For this reason we have made the cyclic clock, which can be copied atomically, directly available.

Function CLOCK_IS_PAST returns TRUE if and only if the clock is past time T, in the cyclic sense of time implemented by this clock. It relies that T is not more than one half cycle (TICKS'last/2) away from the current clock value.

Function "+" returns the sum of L and R, in the cyclic sense of time implemented by this clock. It relies that R is not more than one half the clock cycle, and raises an exception if R is too large.

We have intentionally omitted a mechanism for adjusting the clock, since any adjustment of the clock is bound affect the duration of any pending delays. This is a fundamental problem with Ada's provision for delays. By not distinguishing relative from absolute delays, Ada does not provide the information needed to determine whether pending delays should be adjusted when the clock is adjusted. We presume that the meaning of delays is relative, and so the clock used to implement delays should never be adjusted. Any adjustment must be provided by package CALENDAR.

Note that the value of TICK will need to be adjusted for different applications, and must be consistent with that in package SYSTEM.

4.4 Delays

Package DELAYS provides support for the implementation of the compiler interface procedures DELAY_SELF, CALL_TIMED, and SELECTIVE_WAIT, which all require some form of delay.

```
with CLOCK;
with TASK_IDS; use TASK_IDS;
package DELAYS is
  procedure DELAY_SELF(D: DURATION);
  procedure INTERNAL_DELAY_SELF(D: DURATION);
  TIME_OF_NEXT_CHECK: CLOCK.TICKS;
  procedure CHECK;
  procedure OPEN_DELAYS(T: TASK_ID);
  procedure CLOSE_DELAYS(T: TASK_ID);
end DELAYS;
```

Procedure DELAY_SELF is callable directly from compiled code. It delays the current task for the specified duration. When the task resumes it is preemptible.

Procedure INTERNAL_DELAY_SELF is used within the Corset/I implementation to delay a task. The only difference between it and DELAY_SELF is that it assumes the processor is already non-preemptible. TIME_OF_NEXT_CHECK is the nearest time procedure CHECK needs to be called. It is advanced by procedure CHECK, which must be called periodically to check for expired delays. CHECK can be called by CLOCK, or by any periodic interrupt handler. So long as CHECK is called periodically, TIME_OF_NEXT_CHECK can be ignored, at the risk of unnecessary calls to CHECK or less accuracy in detecting expired delays.

Procedure OPEN_DELAYS is called when a task is created, to initialize data structures used in implementing delays. Procedure CLOSE_DELAYS is called when a task completes, to cancel any pending delays. It is intended specifically for task abortion.

4.5 Calls

```
with SYSTEM;
with TASK_IDS; use TASK_IDS;
package CALLS is
  type KIND_OF_CALL is
       (NORMAL, CONDITIONAL, TIMED, OFFLINE, NONE);
  type QUEUE is limited private;
  function  COUNT(Q: QUEUE) return NATURAL;
  procedure DELETE(Q: in out QUEUE; T: TASK_ID);
  procedure ENQUEUE(Q: in out QUEUE;
                    KIND: KIND_OF_CALL;
                    CALLER: TASK_ID;
                    PARAM: SYSTEM.ADDRESS);
  procedure MAKE_EMPTY(Q: out QUEUE);
  -- should not be called unless Q is uninitialized.
  -- (i.e., Q should not have anything queued on it)
  procedure DEQUEUE(Q: in out QUEUE;
                    KIND: out KIND_OF_CALL;
                    CALLER: out TASK_ID;
                    PARAM: out SYSTEM.ADDRESS);
private
  type CALL_REQUEST;
  type ACCESS_CALL_REQUEST is access CALL_REQUEST;
  type QUEUE is
       record HEAD, TAIL: ACCESS_CALL_REQUEST;
       end record;
end CALLS;
```

4.6 Rendezvous

The package RENDEZVOUS implements all the Ada rendezvous operations. Its services are needed only if these operations are used.

```
with SYSTEM;
with TASK_IDS; use TASK_IDS;
package RENDEZVOUS is
  type ENTRY_INDEX is range -(2**15)..(2**15)-1;
  NULL_ENTRY: ENTRY_INDEX:= 0;
  type ENTRY_BINDING is
       record INDEX: ENTRY_INDEX;
              PROC:  SYSTEM.ADDRESS;
       end record;
  type ENTRY_BINDING_LIST is
       array (POSITIVE range <>) of ENTRY_BINDING;
  NULL_CHOICE: INTEGER:= 0;
  NULL_RENDEZVOUS: constant INTEGER:= 0;
  RENDEZVOUS_COMPLETED: constant INTEGER:= 1;
```

The compiler should assign nonegative indices, starting with 1, to all the entries of a task, counting each member of a family as an individual entry. The acceptable range of type ENTRY_INDEX, will depend on the Corset implementation. Passing a Corset procedure an entry index that is out of range may cause STORAGE_ERROR to be raised if table sizes do not permit so many entries. The value NULL_ENTRY is reserved to denote no entry.

An entry binding list is a list of pairs, each of which gives the index of an entry and the address of the entry procedure which is to executed during a rendezvous on that entry. Each such pair corresponds to an open accept alternative of a selective wait statement. There may be more than one pair with the same entry index. Pairs with entry index NULL_ENTRY are ignored, but all other pairs must have legitimate entry procedure addresses. The lower index bound of an entry binding list is by convention always 1. The value NULL_CHOICE is assumed to be outside the range of entry-list indices.

For efficiency, separate procedures are provided for each of the variants of entry calls: CALL_SIMPLE, CALL_CONDITIONAL, and CALL_TIMED.

```
  procedure CALL_SIMPLE(A: TASK_ID;
                        E: ENTRY_INDEX;
                        LOC: SYSTEM.ADDRESS;
                        SIZE: NATURAL);
  procedure CALL_CONDITIONAL(A: TASK_ID;
                             E: ENTRY_INDEX;
                             LOC: SYSTEM.ADDRESS;
                             SIZE: NATURAL;
                             RESULT: out INTEGER);
  procedure CALL_TIMED(A: TASK_ID;
                       E: ENTRY_INDEX;
                       LOC: SYSTEM.ADDRESS;
                       SIZE: NATURAL;
                       D: DURATION;
                       RESULT: out INTEGER);
```

For each of these procedures LOC is the address of the parameter block, and SIZE is its size in addressable storage units. For the conditional and timed calls, RESULT is set to NULL_RENDEZVOUS if there was no rendezvous and otherwise it is set to RENDEZVOUS_COMPLETED.

The procedure SELECTIVE_WAIT implements all forms of the accept, including the selective wait statement. For efficiency, a separate procedure, ACCEPT_CALL, is also provided for the simple accept statement. Values of subtype MODES (below) are passed as parameters to the selective wait procedure to specify which form of selective wait is desired.

```
  SIMPLE_MODE:    constant INTEGER:= 0;
  DELAY_MODE:     constant INTEGER:= 1;
  ELSE_MODE:      constant INTEGER:= 2;
  TERMINATE_MODE: constant INTEGER:= 3;
  subtype MODES is INTEGER
                   range SIMPLE_MODE..TERMINATE_MODE;
```

SIMPLE_MODE indicates there are only select alternatives. DELAY_MODE indicates there are only accept and delay alternatives. ELSE_MODE indicates there are only accept alternatives and an else part. TERMINATE_MODE indicates there are only accept alternatives and a terminate alternative.

```
  procedure ACCEPT_CALL(E: ENTRY_INDEX;
                        PROC: SYSTEM.ADDRESS);
  procedure SELECTIVE_WAIT(OPEN_ENTRIES:
                           ENTRY_BINDING_LIST;
                           D: DURATION;
                           SELECT_MODE: MODES;
                           RESULT: out INTEGER);
```

ACCEPT_CALL accepts calls on a single entry E, with the entry procedure starting at address PROC.

SELECTIVE_WAIT takes as parameters an entry binding list, OPEN_ENTRIES, and a select mode, as described above. In addition, if the mode is DELAY_MODE, it takes a delay duration. It returns the index in the parameter OPEN_ENTRIES of the entry which was chosen, if any; otherwise it returns NULL_CHOICE.

```
  function  COUNT(T: TASK_ID; E: ENTRY_INDEX)
                                    return NATURAL;
  function  CALLABLE(T: TASK_ID) return BOOLEAN;
```

Functions COUNT and CALLABLE implement the standard Ada attributes of the corresponding names If T = NULL_TASK, COUNT should return zero and CALLABLE should return FALSE.

```
  function  CALLING_TASK return TASK_ID;
```

Function CALLING_TASK is support for an extension to standard Ada, returning the ID of the calling task in a rendezvous if called by the acceptor during the rendezvous. Otherwise, the result is undefined.

In addition to the exported operations this package includes the following procedures, which is are part of the Corset/I internal interface.

```
  procedure OPEN_ENTRIES(T: TASK_ID; NUM_ENTRIES: INTEGER);
  procedure CLOSE_ENTRIES(T: TASK_ID);
end RENDEZVOUS;
```

The procedure OPEN_ENTRIES at the time of task creation, to initialize the data structures used by the implementation of rendezvous The procedure CLOSE_ENTRIES is called when task T completes to insure correct behaviour with respect to rendezvous. This includes, for example, raising TASKING_ERROR in any tasks with pending calls to entries of T.

4.7 Storage Management

Package STORAGE_MANAGEMENT is a Corset internal package on which more complex forms of storage allocation can be based. It provides basic global storage allocation and deallocation services for large heterogeneous-sized blocks. It is used by the body of package STAGES to allocate and deallocate task workspaces. It might also be used by the body of package COLLECTIONS to allocate and deallocate collections. A good candidate for an implementation would be a boundary-tag scheme.

```
with SYSTEM;
package STORAGE_MANAGEMENT is
  UNIT: INTEGER:= INTEGER'size;
  procedure RELEASE_BLOCK(A: in out SYSTEM.ADDRESS);

  function  REQUEST_BLOCK(SIZE: NATURAL) return
SYSTEM.ADDRESS;
end STORAGE_MANAGEMENT;
```

RELEASE_BLOCK releases the block of storage beginning with address A. This must be an entire block previously allocated by REQUEST. REQUEST_BLOCK allocates a block of contiguous storage of SIZE addressable units, and returns the address of the block. That is, if B is the address returned, the allocated block occupies addresses B,...,B+SIZE-1. Any overhead storage that may be required by the storage manager is not included in the block. It returns NULL_ADDRESS if the request cannot be honored.

4.8 Stages

Package STAGES implements the normal transitions between stages in a task's lifetime: creation, activation, completion, and termination.

```
with SYSTEM;
with MACHINE;
with TASK_IDS; use TASK_IDS;
package STAGES is
  type MASTER_ID is private;
  NULL_MASTER: constant MASTER_ID;
  procedure ACTIVATE_TASKS(TASKS: TASK_LIST);
  procedure COMPLETE_ACTIVATION;
```

A value of type TASK_LIST is used to describe a set of tasks to be activated or aborted. The lower index bound of this array type is by convention always 1. Procedure ACTIVATE_TASKS is called by the creator of a list of tasks when it is time for them to begin activation; it does not return until all the tasks have completed activation. Procedure COMPLETE_ACTIVATION should be called by an activating task when it completes activation, so that its creator can eventually return from ACTIVATE_TASKS.

```
  procedure COMPLETE_TASK;
```

Procedure COMPLETE_TASK is called from a task when it completes execution.

```
  procedure CREATE_TASK(SIZE: NATURAL;
                        PRIO: INTEGER;
                        INIT_STATE: MACHINE.STATE;
                        M: MASTER_ID;
                        NUM_ENTRIES: NATURAL;
                        CREATED_TASK: out TASK_ID);
```

Procedure CREATE_TASK is called to create a new task, whose ID is returned via parameter CREATED_TASK. The parameter M specifies the master of the new task. The parameter SIZE is the required size of the workspace to be allocated to the task, which is a contiguous block of memory. PRIO is the priority of the task, and NUM_ENTRIES is the number of entries. INIT_STATE is a record specifying the initial values of all (nonpriviledged) registers for the new task, including the initial value of the program counter and any base registers. A workspace may be allocated for the task by the creator, in which case the address of this workspace may be passed to the new task as part of INIT_STATE. Alternatively, the new task may call the RTE to allocate its own workspace.

```
  function  CURRENT_MASTER return MASTER_ID;
  procedure ENTER_MASTER;
  procedure EXIT_MASTER;
```

Function CURRENT_MASTER returns the ID of the master from which it is called. ENTER_MASTER and EXIT_MASTER are to be called on entry to and exit from a nontrivial master of tasks other than a task body.

```
  function  TERMINATED(T: TASK_ID) return BOOLEAN;
  function  IS_LOCAL_TASK(T: TASK_ID;
                          M: MASTER_ID) return BOOLEAN;
```

Function TERMINATED implements the standard Ada attribute of the same name. It should return TRUE if T is a terminated task or T = NULL_TASK. Function IS_LOCAL_TASK should return TRUE if and only if T's parent is the current task and T is directly dependent on master M. This function is intended to support special-case treatment for functions that return local tasks.

The following procedures are not part of the interface to compiled code, but are part of the internal interface between Corset/I components.

```
  procedure MAKE_PASSIVE(T:  TASK_ID);
  procedure WAKE_UP(T: TASK_ID);
  procedure COMPLETE(T: TASK_ID);
  procedure TERMINATE_DEPENDENTS(M: MASTER_ID:=
                                    NULL_MASTER);
  procedure CLOSE_ACTIVATIONS(T: TASK_ID);
private
  type MASTER_ID is new INTEGER;
  NULL_MASTER: constant MASTER_ID:= 0;
end STAGES;
```

Procedure MAKE_PASSIVE is called when starting to wait on a terminate alternative, and procedure WAKE_UP is called when leaving a terminate alternative. Procedure COMPLETE is called to complete the execution of a task, whether through normal completion, an exception, or abortion. Procedure TERMINATE_DEPENDENTS is called by a task to terminate and collect all its dependents at relative master nesting levels greater than or equal to the parameter M. For M = NULL_MASTER, it terminates and collects all dependents of the calling task.

Procedure CLOSE_ACTIVATIONS is called when completing a task, to clean up any references to the task with respect to activation.

4.9 Abortion

Package ABORTION provides support for the task abortion. In addition to the procedures that directly implement abortion, it provides a shell around certain services exported by Lace. These are Lace operations that can be implemented fairly efficiently in the absence of abortion, but are significantly more complex if abortion is supported. The intention is that the checks which these subprograms perform can be eliminated, replacing these subprograms by the corresponding straight Lace operations which they encapsulate, if a program does not include any abort statements.

```
with SYSTEM;
with TASK_IDS; use TASK_IDS;
package ABORTION is
  procedure ABORT_TASKS(TASKS: TASK_LIST);
  procedure ABORT_TASK(T: TASK_ID);
  procedure ABORT_SELF;
  procedure FORESTALL_ABORTION(T: TASK_ID;
                               RESULT: out BOOLEAN);
```

```
  procedure SAFE_RELEASE(T: TASK_ID);
  procedure SAFE_HOLD(T: TASK_ID);
  procedure SAFE_FORCE_CALL(T: TASK_ID; P: SYSTEM.ADDRESS);
  procedure OPEN_ABORTION(T: TASK_ID);
end ABORTION;
```

Procedure ABORT_TASKS is the only one that is exported from Corset. It causes abortion of all the tasks in the list, implementing the Ada abort statement.

Procedure ABORT_TASK causes the abortion of task T, if T is not already completed. It assumes that the task T is not already abnormal. It interlocks with SAFE_HOLD and SAFE_RELEASE so that they do not hold or release abnormal tasks, and with SAFE_FORCE_CALL so that it does not force a call in an aborting task.

A task that is aborted is forced to call procedure ABORT_SELF to abort itself. The effect is similar to ordinary completion.

Procedure FORESTALL_ABORTION does not correspond to any Lace operation. It is used to change the effect of a previously executed SAFE_HOLD on task T, so that T will not not be released if it aborted. This is used when a calling task becomes committed to a rendezvous, so that if aborted it will not complete before the rendezvous is complete.

Procedure SAFE_RELEASE is like LACE.RELEASE, except that it has no effect unless the task T has suspended itself using SAFE_HOLD, and has not subsequently been released via SAFE_RELEASE.

Procedure SAFE_HOLD is like LACE.HOLD, except that it calls DISPATCH if the task T is abnormal, and is interlocked with SAFE_RELEASE as described above.

Procedure SAFE_FORCE_CALL is like LACE.FORCE_CALL, except that it has no effect if the task T has had the right RIGHT_TO_COMPLETE removed. This prevents exceptions from overriding and nullifying pending calls to ABORT_SELF, and prevents a task from being aborted or having an exception raised in it on return from calls it may make to DISPATCH during its normal completion code.

Procedure OPEN_ABORTION is called when a task is created, to initialize the data structures used in implementing abortion.

Note that if abortion is supported, it must be the "safe" versions of the Lace operations RELEASE and HOLD that are exported to the application level. For this reason, the non-waiting property of the Lace RELEASE must be preserved by SAFE_RELEASE.

5 Conclusions

An important thing we have learned from the design of the Corset/I and Lace/I implementations is that there are usable subsets of the full Ada tasking model that can be implemented very simply and efficiently, and with predictable execution timing. In particular, this is true of the subset that does not require dynamic creation and termination of tasks. Even greater efficiency can be obtained if certain other features are not supported.

Certainly the worst culprit in adding complexity is abortion. Eliminating abortion would permit removing almost all the locks (i.e. indeterminate waiting) in our draft implementation. Abortion therefore seems to be a strong candidate for banning in applications where runtime effienciency and deterministic timing are important.

We have not been able to fully achieve our goal of designing a RSE composed of modules from which one could produce a configuration to support only the actual Ada features used in a particular application program. Though the design presented here is composed of procedures that can be omitted if not called, some other ways in which an application might use less than the full generality of Ada tasking cannot be detected and taken advantage of so easily. An example is the issue of whether nested accept statements need to be supported. Eliminating this possibility would allow simplification of the code for rendezvous.

A big problem is that some of the more complex features like abortion insinuate themselves into the implementation of more basic features like rendezvous. It may be that the only way to get maximum runtime efficiency for applications that do not require the more complex features is to provide alternate versions of the more basic packages.

Examples of Ada features whose elimination would reduce implementation overhead and improve timing predictability include:

- dynamic task creation and termination;
- abortion;
- nested accepts;
- task priorities;
- the terminate alternative;
- delays used as time-outs on entry calls and selective waits;
- conditional entry calls.

Our own feeling is that a very usable subset of Ada tasking could be implemented that would reduce by 25 to 75 percent the code size and reduce by 20 to 50 percent the execution time of simple tasking operations, as compared with the full model. These figures are tentatively supported by experiments we have done.

During the design of the Corset/I implementation, we encountered some points where it seemed we could implement things more simply and efficiently by violating the interface rules we had laid down. A salient example is the requirement that Corset calls (and calls forced by Corset on a task) be compilable as ordinary procedure calls. This means that Corset calls will probably require stack space, which may not be available, and serious problems such as looping through the STORAGE_ERROR exception can ensue. At the machine level, this problem can be avoided, by using only static storage allocation and registers within Corset.

Another example is in the separation of the implementation of exception handling from Corset. This forced us to allow for exceptions being propagated through Corset calls, which seems hazardous and possibly inefficient. We would rather have been able to coordinate the handling of exceptions so that exceptions never propagate through Corset, but are simply reraised, when necessary, on return from a Corset call. This would require modification to the Lace dispatcher in raising the exception, somewhat like it now treats forced calls, and cooperation of the exception handling implementation in returning normally to Corset from entry procedure calls.

Of course, some loss of efficiency seems to be unavoidable whenever we try to enforce clean interfaces. Such costs are usually repaid in greater system reliability, simpler maintenance, and reusability of components. We intend now to go back and reconsider the Lace and Corset interfaces, to determine whether runtime efficiency can be improved without loss of functional separation or serious growth of interface complexity.

Corset and Lace are still under development. This version already represents several generations of revisions to our first draft design, and we continue to look for ways to improve it. In particular, we will reconsider whether modification to the Lace interface can reduce the complexity of the Corset implementation. Lace stabilized in the early stages of the Corset design, before the full complexity of task termination and abortion were confronted. Now we see some apparent weaknesses. For example, duplication of "safe" versions of Lace procedures in package ABORTION might be eliminated if slightly more general Lace operations were provided. On the other hand, it appears that the forced call may not require parameters, so an awkward-to-implement detail could be removed from Lace. It appears these two changes might be combined by supporting a primitive form of signal and signal-handler within Lace, in place of the forced call.

Another change we are considering is promoting entries to be independent of tasks. This would better support some Ada extensions with which we would like to experiment, including remote entry calls and multiple-server entries.

Work on an executive for the MIL-STD-1750A derived from Lace is under way at the Boeing Aerospace Company, in Kent, Washington. Work on the Corset/I implementation is continuing at the Florida State University. A simulated multiprocessor implementation, using standard Ada, is being developed to validate the interfaces and algorithms. Compiled Ada tasks are simulated by sections of non-tasking Ada code with Corset service calls inserted explicitly. These are then compiled and executed using an existing Ada compiler, bypassing the tasking implementation of the compiler. When this is done the next steps would be: to produce an efficient version, revising algorithms and recoding low-level operations in assembly language as needed; to

test it on a multiprocessor hardware configuration; to integrate it with a compiler. By experimenting with such an implementation we would hope to discover areas where improvements could be fed back.

Draft code for the Corset/I and Lace/I components are included in [2] and [3].

References

[1] *Military Standard Ada Programming Language*, ANSI/MILSTD1815A, U.S. Department of Defense, Ada Joint Program Office (January 1983).

[2] T.P. Baker, "A Corset for Ada" TR 86-09-07, University of Washington Computer Science Department (1987).

[3] T.P. Baker, and K. Jeffay, "A Lace for Ada's Corset" TR 86-09-06, University of Washington Computer Science Department (1986).

[4] Ada Run Time Environment Working Group (ACM SIGAda), *A Catalog of Interface Features and Options for the Ada Run Time Environment*, Release 1.0, ACM SIGAda (October 1986).

[5] T.P. Baker, "An Improved Ada Runtime System Interface", TR 86007-05, University of Washington Computer Science Department (July 1986).

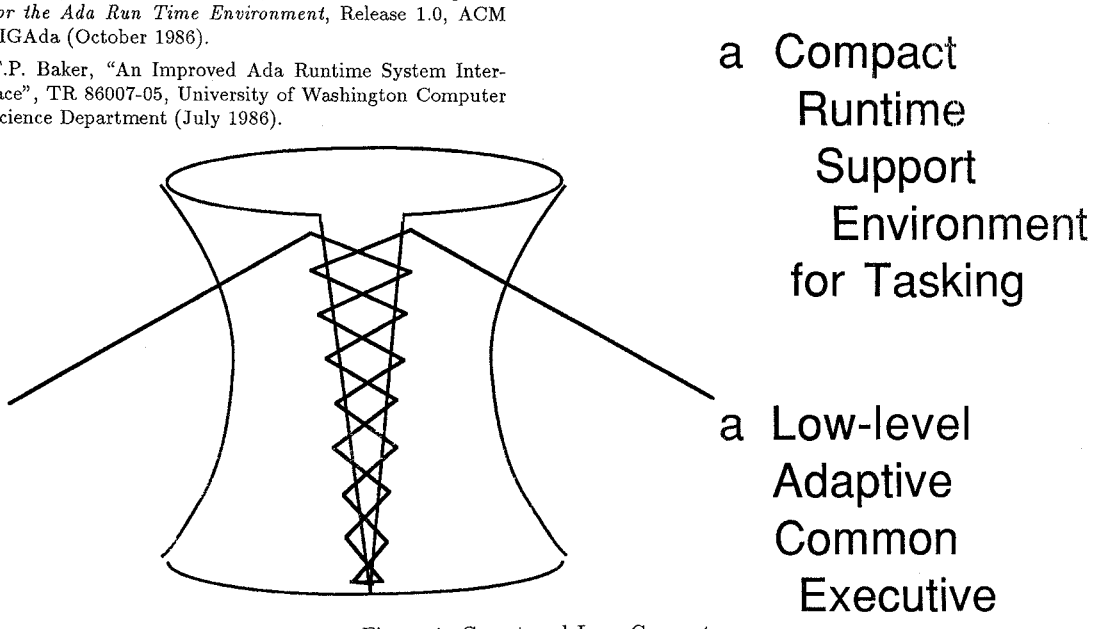

Figure 1: Corset and Lace Concept

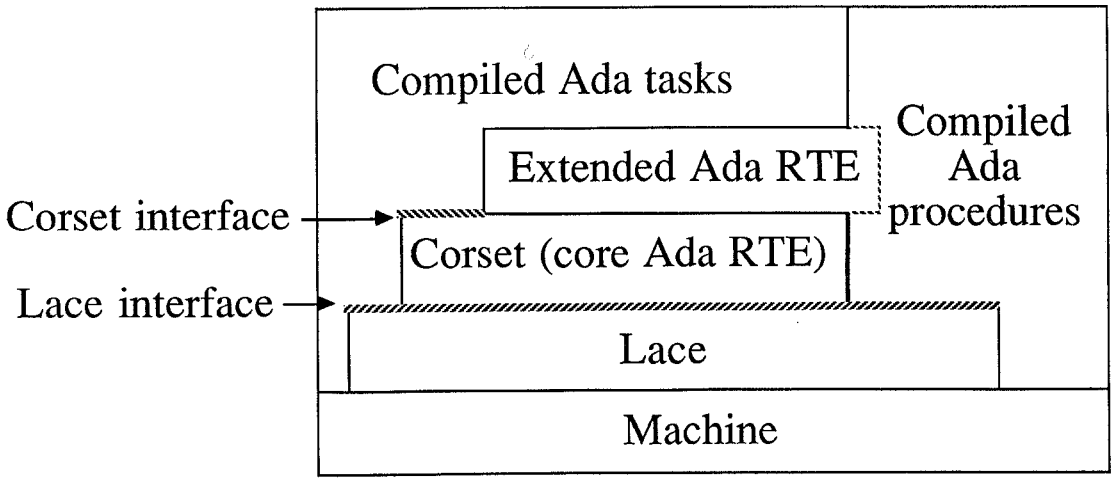

Figure 2: Corset and Lace Layers