Chapter 16
The execution environment

16.1 The role of the execution environment
16.2 Tailoring the execution environment
16.3 Scheduling models
16.4 Hardware support
Summary
Further reading
Exercises

By their very nature, real-time systems have to exhibit a timely response to events occurring in their surrounding environment. This has led to the view that real-time systems must be as fast as possible, and that any overheads introduced by language or operating system features that support high-level abstractions (such as monitors, exceptions, atomic actions and so on) cannot be tolerated. The term 'efficiency' is often used to express the quality of the code produced by a compiler or the level of abstraction provided by mechanisms supported by an operating system or run-time support system. This term, however, is not well-defined. Furthermore, efficiency is in many ways a poor metric for assessing an application and its implementation. In real-time systems, what is actually important is the meeting of deadlines or the attainment of adequate response times in a particular execution environment. This chapter considers some of the issues associated with meeting this goal. Firstly, the impact of the execution environment on the design and implementation of real-time systems is considered. Then, ways in which the software execution environment can be tailored to the needs of the application are discussed. Scheduling models of kernels are then reviewed so as to facilitate the complete schedulability analysis of an application. This is followed by illustrations of how some of the abstractions presented in this book can be supported by hardware in the execution environment.

16.1 The role of the execution environment

In Chapter 13, schedulability analysis was considered to be essential for predicting the real-time properties of software. It is difficult to undertake this analysis unless the details

of the proposed execution environment are known. The term 'execution environment' is used to mean those components that are used together with the application's code to make the complete system: the processors, networks, operating systems and so on. The nature of the proposed execution environment will dictate whether a particular design will meet its real-time requirements. Clearly, the more 'efficient' use of the execution environment, the more likely the requirements will be met. But this is not always the case, and a poorly structured design may fail to meet its requirements irrespective of how efficiently it is implemented. For example, a design suffering from significant priority inversion will fail to meet attainable deadlines irrespective of how efficiently it is implemented – as was so poignantly illustrated in the Mars Pathfinder mission (Jones, 1997; Reeves, 1997).

The design process can be viewed as a progression of increasingly specific **commitments** and **obligations**. The commitments define properties of the system design which designers operating at a more detailed level are not at liberty to change. Those aspects of a design to which no commitment is made at some particular level in the design hierarchy are effectively the subject of obligations that lower levels of design must address.

The process of refining a design – transforming obligations into commitments – is often subject to **constraints** imposed primarily by the execution environment. The choice of the execution environment and how it is used may also be constrained. For example, there may be a requirement which dictates that a space-hardened processor be used, or there may be a certification requirement which dictates that the capacity of the processor (or network) shall not exceed 50%.

Many design methods distinguish between a logical and physical design. The logical design focuses on meeting the functional requirements of the application and assumes a sufficiently fast execution environment. The physical architecture is the result of combining the functional and the proposed execution environment to produce a complete software and hardware architecture design.

The physical architecture forms the basis for asserting that all the application's requirements will be met once the detailed design and implementation have taken place. It addresses timing (and even dependability) analysis that will ensure (guarantee) that the system once built will (within some stated failure hypotheses) satisfy the real-time requirements. To undertake this analysis, it will be necessary to make some initial estimations of resource usage of the proposed system (hardware and software). For example, initial estimates of the timing properties of the application can be made and schedulability analysis undertaken to give confidence that deadlines will be met once the final system has been implemented.

The focus of physical architecture design is to take the functional architecture and to map it on to the facilities provided by the execution environment. Any mismatch between the assumptions made by the functional architecture and the facilities provided by the execution environment must be addressed during this activity. For example, the functional architecture might assume that all functions are immediately visible from all other functions. When functions are mapped to processors in the physical architecture, there may be no direct communication path provided by the infrastructure, and consequently it may be necessary to add extra application-level router functions. Furthermore, the infrastructure may only support low-level message passing, whereas the functions

may communicate using procedure calls; consequently it will be necessary to provide an application-level RPC facility. There is clearly a trade-off between the sophistication of the execution environment and the need to add extra application facilities to the functional architecture during the production of the physical architecture. However, it is also important not to provide sophisticated mechanisms in the execution environment if they are not needed by the application, or worse, the application needs more primitive facilities which it must then try to construct from high-level ones. This is often called **abstraction inversion**.

Once the initial architectural design activities are complete, the detailed design can begin in earnest and all components for the application produced. When this has been achieved, each component must be analyzed using tools to measure characteristics of the application such as its worst-case execution time (or its complexity, for example, if dependability is being considered) to confirm that estimated worst-case execution times are accurate, (or that certain modules were complex and therefore prone to software errors, resulting in design diversity being needed). If these estimations were not accurate (which will usually be the case for a new application), then either the detailed design must be revisited (if there are small deviations), or the designer must return to the architectural design activities (if serious problems exist). If the estimation indicates that all is well, then testing of the application proceeds. This should involve measuring the actual timing of code, the number of bugs found and so on. The process is illustrated in Figure 16.1 (this is actually the life cycle supported by the HRT-HOOD design methods considered in Chapter 2 and used in the case study presented in Chapter 17).

What is important, therefore, is not so much efficiency of compiled code or operating systems overheads, but rather that timing analysis be undertaken as early on in the life cycle as possible. Having said this, a grossly inefficient compiler would clearly be an inappropriate tool to employ; such inefficiencies being an indication of a poorly engineered product.

16.2 Tailoring the execution environment

Modern operating systems, and the run-time support systems associated with languages like Ada, are awash with functionality because they try to be as general purpose as possible. Clearly, if a particular application does not use certain features of an operating system, it would be advantageous to customize its run-time existence. This ability is essential for three main reasons:

(1) it avoids unnecessary resource usage, be it processor time or memory;

(2) it reduces the amount of software whose correctness has to be argued during any certification process;

(3) many development standards require ‘dead code’ to be removed.

This section considers the facilities provided by Ada and POSIX that aid this process. Real-Time Java does not, in general, support optional components as it is contrary to

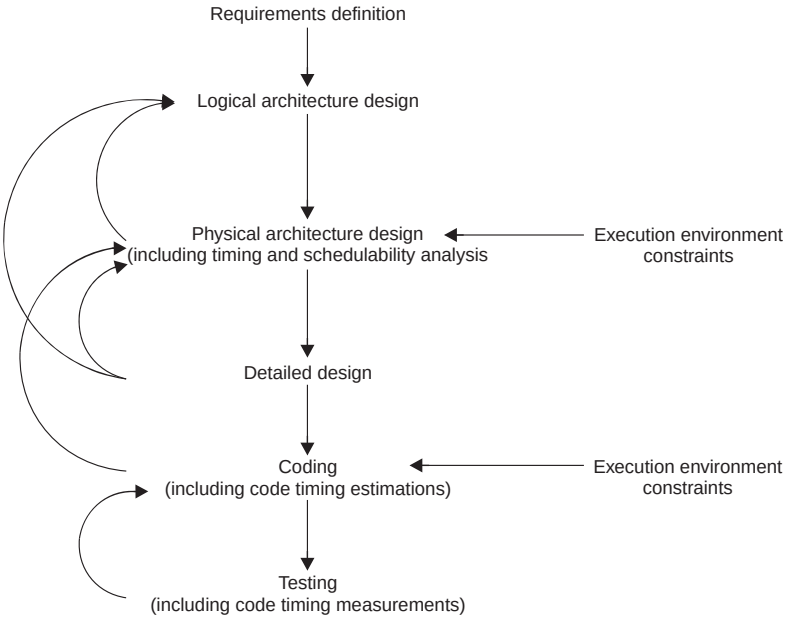

Figure 16.1 A hard real-time life cycle.

the *Write Once Run Anywhere* principle. However, it does accept that some components cannot be implemented if the underlying support system does not provide that functionality. The most obvious case being the class that interfaces to POSIX signals.

16.2.1 Restricted tasking in Ada

The Real-Time Annex of Ada allows the programmer to specify a set of restrictions that a run-time system should recognize and 'reward' by giving more effective support. The following are examples of restrictions which are identified by pragmas, and are checked and enforced before run-time:

- `No_Task_Hierarchy` – This significantly simplifies the support required for task termination.
- `No_Abort_Statement` – This affects all aspects of the run-time support system, as there is no need to worry about a task being aborted while in a rendezvous, in a protected operation, propagating an exception, waiting for child termination and so on.
- `No_Terminate_Alternatives` – Again simplifies the support required for task termination.

- `No_Task_Allocators` – Allows the run-time system to be configured for a static number of tasks and removes a need for dynamic memory allocation.
- `No_Dynamic_Priorities` – Simplifies many aspects of the support for task priorities as priority will not change dynamically (other than via the use of ceiling priorities).
- `No_Asynchronous_Control` – This affects all aspects of the run-time support system as there is no need to worry about a task receiving an asynchronous event while in a rendezvous, in a protected operation, propagating an exception, waiting for child termination and so on.
- `Max_Select_Alternatives` – Allows the use of static data structures and removes a need for dynamic memory allocation.
- `Max_Task_Entries` – Again allows the use of static data structures and removes a need for dynamic memory allocation. A value of zero indicates that no rendezvous are possible.
- `Max_Protected_Entries` – Again allows the use of static data structures and removes a need for dynamic memory allocation. A value of zero indicates that no condition synchronization is allowed for protected objects.
- `Max_Tasks` – Specifies the maximum number of tasks and therefore allows the run-time to provide a fixed amount of static support structures.

Note that Ada also has a Safety and Security Annex which sets all the above restrictions to zero (that is *no tasking*!). It also introduces a further restriction that disallows protected types and objects. Current practice, in the safety critical application area, forbids the use of tasks or interrupts. This is unfortunate, as it is possible to define a subset of the tasking facilities that is both predictable and amenable to analysis. It is also possible to specify run-time systems so that they can be implemented to a high level of integrity.

One of the challenges facing Ada practitioners over the coming decade is to demonstrate that concurrent programming is an effective and safe technique for even the most stringent of requirements. Towards this goal, the 8th International Real-Time Ada Workshop (Burns, 1999) defined a tasking profile (known as the *Ravenscar Profile*) for use in high-integrity or performance-sensitive applications. In the Ravenscar Profile, use of the following features is forbidden:

- Task types and object declarations other than at the library level. Thus, there is no hierarchy of task types.
- Unchecked deallocation of protected and task objects (and hence finalization). Dynamic allocation of such objects may be allowed, but only if the sequential part of the high-integrity language profile allows dynamic allocation of other objects.
- Requeue.
- ATC (asynchronous transfer of control via the *select then abort* statement).
- Abort statements.
- Task entries.

- Dynamic priorities.
- Package `Calendar`.
- Relative delays.
- Protected types other than at the library level.
- Protected types with more than one entry.
- Protected entries with barriers other than a single boolean variable declared within the same protected type.
- Attempts to queue more than one task on a single protected entry.
- Locking policies other than *Ceiling locking*.
- Scheduling policies other than *FIFO within priorities*.
- All forms of the select statement.
- User-defined task attributes.

In addition to these restrictions, an implementation can make the assumption that none of the program's tasks will terminate. Note that most, but not all, of these constraints can be defined using the `Restrictions` pragma. Even with these limitations, an application conforming to the Ravenscar Profile still has:

- Task objects, restricted as above.
- Protected objects, restricted as above.
- Suspension objects.
- Atomic and volatile pragmas.
- 'Delay until' statements.
- *Ceiling locking* policy and *FIFO within priority* dispatching.
- The Count attribute (but not within entry barriers).
- Task identifiers.
- Task discriminants.
- The `Real_Time` package.
- Protected procedures as interrupt handlers.

Protected types with only subprogram interfaces are provided for simple mutual exclusion. The special form of protected entry (that is, only one per protected object and a maximum of one possible caller of that entry) is available as an event signalling mechanism to allow aperiodic or sporadic tasks to be supported.

In addition to the features described above, the Real-Time Systems Annex defines a number of implementation requirements, documentation requirements and metrics. The metrics allow the costs (in processor cycles) of the run-time system to be obtained. They also indicate which primitives can lead to blocking, and which must not.

The timing features (that is, real-time clock and delay primitives) are defined precisely. It is thus, for example, possible to know the maximum time between a task's

delay value expiring and it being placed on the run queue. All this information is needed for the analysis of an application within the context of its execution environment.

16.2.2 POSIX

POSIX consists of a variety of standards. There is the base standard, the real-time extensions, the threads extensions and so on. If implemented in a single system, it would contain a huge amount of software. To help produce more compact versions of operating systems which conform to the POSIX specifications, a set of application environment **profiles** have been developed, the idea being that implementors can support one or more profiles. For real-time systems, four profiles have been defined:

- PSE50 – Minimal real-time system profile – intended for small single/multiprocessor embedded systems controlling one or more external devices; no operator interaction is required and there is no file system. Only a single multithreaded process is supported.
- PSE51 – Real-time controller system profile – an extended PSE50 for potentially multiple processors with a file system interface and asynchronous I/O.
- PSE52 – Dedicated real-time system profile – an extension of PSE50 for single or multiple processor systems with memory management units; includes multiple multithreaded processes, but no file system.
- PSE53 – Multi-purpose real-time system profile – capable of running a mix of real-time and non real-time processes executing on single/multiprocessor systems with memory management units, mass storage devices, networks and so on.

Table 16.1 illustrates the type of functionality provided by PSE50, PSE51 and PSE52.

In general, a POSIX system is also free not to support any of the optional units of functionality it chooses, and so much finer control over the supported functionality is possible. All of the real-time and the thread extensions are optional. However, conforming to one of the profiles means that all the required units of functionality must be supported.

16.3 Scheduling models

The execution environment has a significant impact on the timing properties of an application. Where there is a software kernel, the overheads incurred by the kernel must be taken into account during the schedulability analysis of the application. The following characteristics are typical of many real-time software kernels:

- The cost of a context switch between processes is not negligible and may not be a single value. The cost of a context switch to a higher-priority periodic process (following, for example, a clock interrupt) may be higher than a context switch

Functionality	PSE50	PSE51	PSE52
pthreads	√	√	√
fork	×	×	√
semaphores	√	√	√
mutexes	√	√	√
message passing	√	√	√
signals	√	√	√
timers	√	√	√
synchronous I/O	√	√	√
asynchronous I/O	×	√	√
priority scheduling	√	√	√
shared memory objects	√	√	√
file system	×	√	×

Table 16.1 POSIX real-time profile functionality.

from a process to a lower-priority process (at the end of the high-priority process's execution). For systems with a large number of periodic processes, an additional cost will be incurred for manipulating the delay queue (for periodic tasks when they execute, say, an Ada 'delay until' statement).

- All context switch operations are non preemptive.
- The cost of handling an interrupt (other than the clock) and releasing an application sporadic process is not insignificant. Furthermore, for DMA and channel-program controlled devices, the impact of shared-memory access can have a non-trivial impact on worst-case performance; such devices are best avoided in hard real-time systems.
- A clock interrupt (say every 10 ms) could result in periodic processes being moved from a delay queue to the dispatch queue. The costs for this operation varies depending on the number of processes to be moved.

In addition to the above, the scheduling analysis must take into account the features of the underlying hardware, such as the impact of the cache and pipeline.

16.3.1 Modelling non-trivial context switch times

Most scheduling models ignore context switch times. This approach is, however, too simplistic if the total cost of the context switches is not trivial when compared with the application's own code. Figure 16.2 illustrates a number of significant events in the execution of a typical periodic process.

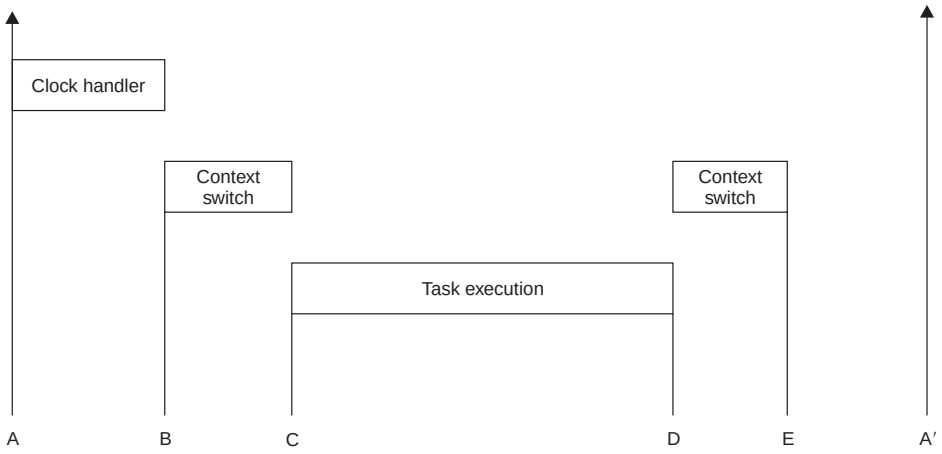

Figure 16.2 Overheads when executing processes.

A – the clock interrupt that designates the notional time at which the process should start (assuming no release jitter or non-preemptive delay, if the interrupts were disabled due to the operation of the context switch then the clock handler would have its execution delayed; this is taken into account in the scheduling equations by the blocking factor B).

B – the earliest time that the clock handler can complete, this signifies the start of the context switch to the process (assume it is the highest priority runnable process)

C – the actual start of the execution of the process

D – the completion of the process (the process may be preempted a number of times between C and D)

E – the completion of the context switch away from the process

A′ – the next release of the process

The typical requirement for this process is that it completes before its next release (that is, D $<$ A′), or before some deadline prior to its next release. Either way, D is the significant time, not E. Another form of requirement puts a bound on the time between the start of execution and termination (that is, D–C). This occurs when the first action is an input and the last an output (and there is a deadline requirement between the two). While these factors affect the meaning of the process's own deadline (and hence its response time) they do not affect the interference this process has on lower-priority processes; here the full cost of both context switches counts. Recall that the basic

scheduling equation (13.7) has the form:

$$R_i \;=\; C_i \;+\; B_i \;+\; \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

This now becomes (for periodic processes only):

$$R_i \;=\; CS^1 \;+\; C_i \;+\; B_i \;+\; \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 \;+\; CS^2 \;+\; C_j) \qquad \textbf{(16.1)}$$

where CS^1 is the cost of the initial context switch (to the process) and CS^2 is the cost of the context switch away from each process at the end of its execution. The cost of putting the process into the delay queue (if it is periodic) is incorporated into C_i. Note that in practice this value may depend on the size of the queue; a maximum value would need to be incorporated into C_i.

This measure of the response time is from point B in Figure 16.2. To measure from point C, the first CS^1 term is removed. To measure from point A (the notional true release time of the process) requires the clock behaviour to be measured (see Section 16.3.3).

16.3.2 Modelling sporadic processes

For sporadic processes released by other sporadic processes, or by periodic processes, Equation (16.1) is a valid model of behaviour. However, the computation time for the process, C_i, must include the overheads of blocking on the agent that controls its release.

When sporadics are released by an interrupt, priority inversion can occur. Even if the sporadic has a low priority (due to it having a long deadline) the interrupt itself will be executed at a high hardware priority level. Let Γs be the set of sporadic processes released by interrupts. Each interrupt source will be assumed to have the same arrival characteristics as the sporadic that it releases. The additional interference these interrupt handlers have on each application process is given by:

$$\sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

where IH is the cost of handling the interrupt (and returning to the running process, having released the sporadic process).

This representation assumes that all interrupt handlers give rise to the same cost; if this is not the case then IH must be defined for each k. Equation (16.1) now becomes:

$$R_i \quad = \quad CS^1 \;+\; C_i \;+\; B_i \;+\; \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 \;+\; CS^2 \;+\; C_j)$$

Queue state	Clock handling time, μs
No processes on queue	16
Processes on queue but none removed	24
One process removed	88
Two processes removed	128
Twenty five processes removed	1048

Table 16.2 Clock handling overheads.

$$+ \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH \qquad \textbf{(16.2)}$$

16.3.3 Modelling the real-time clock handler

To support periodic processes, the execution environment must have access to a real-time clock that will generate interrupts at appropriate times. An ideal system will use an interval timer, and will interrupt only when a periodic process needs to be released. The more common approach, however, is one in which the clock interrupts at a regular rate (say once every 10 ms) and the handler must decide if none, one, or a number of periodic processes must be released. The ideal approach can be modelled in an identical way to that introduced for sporadic processes (see Section 16.3.2). With the regular clock method, it is necessary to develop a more detailed model as the execution times of the clock handler can vary considerably. Table 16.2 gives possible times for this handler (for a clock period of 10 ms). Note that if the worst case was assumed to occur on all occasions over 100% of the processor would have to be assigned to the clock handler. Moreover, all this computation occurs at a high (highest) hardware priority level, and hence considerable priority inversion is occurring. For example, with the figures given in the table, at the LCM (least common multiple) of 25 periodic processes $1048\,\mu$s of interference would be suffered by the highest priority application process that was released. If the process was released on its own then only $88\,\mu$s would be suffered. The time interval is represented by B–A in Figure 16.2.

In general, the cost of moving N periodic processes from the delay queue to the dispatch queue can be represented by the following formulae:

$$C_{clk} = CT^c + CT^s + (N - 1)CT^m$$

Where CT^c is the constant cost (assuming there is always at least one process on the delay queue), CT^s is the cost of making a single move, and CT^m is the cost of each subsequent move. This model is appropriate due to the observation that the cost of moving just one process is often high when compared with the additional cost of moving extra processes. With the kernel considered here, these costs were:

CT^c	24 μs
CT^s	64 μs
CT^m	40 μs

To reduce the pessimism of assuming that a computational cost of C_{clk} is consumed on each execution of the clock handler, this load can be spread over a number of clock ticks. This is valid if the shortest period of any application process, T_{min} is greater than the clock period, T_{clk}. Let M be defined by:

$$M = \left\lceil \frac{T_{min}}{T_{clk}} \right\rceil$$

If M is greater than 1 then the load from the clock handler can be spread over M executions. In this situation, the clock handler is modelled as a process with period T_{min} and computation time C'_{clk}:

$$C'_{clk} = M(CT^c + CT^s) + (N - M)CT^m$$

This assumes $M <= N$.

Equation (16.2) now becomes

$$\begin{aligned} R_i = {} & CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \\ & + \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH \\ & + \left\lceil \frac{R_i}{T_{min}} \right\rceil C'_{clk} \end{aligned} \qquad \textbf{(16.3)}$$

To give further improvements (to the model) requires a more exact representation of the clock handlers actual execution. For example, using just CT^c and CT^s the following equation can easily be derived:

$$\begin{aligned} R_i = {} & CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \\ & + \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{clk}} \right\rceil CT_c \\ & + \sum_{g \in \Gamma p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s \end{aligned} \qquad \textbf{(16.4)}$$

where Γp is the set of periodic processes.

It is left as an exercise for the reader to incorporate the three-parameter model of clock handling (see Exercise 16.2).

16.3.4 The impact of the cache on worst-case execution time analysis

The problems of undertaking WCET analysis for processes executing on modern processors has already been mentioned in Section 13.12.1. In particular, it is necessary to model the behaviour of the processor's cache and pipeline. In Equation (16.4), C_i and C_j are the values that are affected. If these values are calculated following a detailed analysis of the processor's architecture, preemptions caused by interrupts need to be taken into account in the scheduling equations. Otherwise, the values used will be optimistic. Fortunately, for hard real-time systems, it is necessary to place bounds on how often interrupts can occur. Each interrupt handler is treated as a sporadic process of a high priority and is considered in the same way as a periodic process of higher priority. Equation (16.4) already specifies the number of preemptions that can occur while process i is executing. It is simply the number of times each higher-priority process can be released during process's i response time. Each preemption will potentially flush the cache and the pipeline. This leads to the following approaches to integrating the preemption penalties. Assume C_i is the worst-case value calculated using models which account for the benefits gained by caches and pipelining in *the absence* of interrupt. Calculate γ, the maximum possible penalty that can be accrued from an interrupt, this being the time taken to refill the cache and the pipeline. Equation (16.4) can now be modified to calculate the effect of interrupts on process i (Busquets and Wellings, 1996):

$$
\begin{aligned}
R_i \quad = \quad & CS^1 \; + \; C_i \; + \; B_i \; + \; \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 \; + \; CS^2 \; + \; C_j) \\
& + \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil (IH \; + \; \gamma) + \left\lceil \frac{R_i}{T_{clk}} \right\rceil (CT_c \; + \; \gamma) \\
& + \sum_{g \in \Gamma p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s
\end{aligned}
\qquad \textbf{(16.5)}
$$

Of course, this is quite pessimistic because not all preemptions will require the whole cache to be refilled. Furthermore, some memory blocks which are replaced would have been replaced anyway. A less pessimistic approach tries to identify the number of cache blocks.

16.4 Hardware support

When concurrent processes are introduced into the solution to any real-time problem, overheads with scheduling, interprocess communication, and so on, occur. Section 16.3.3 has attempted to model these overheads in the schedulability analysis. Several attempts have also been made to reduce these overheads by providing direct hardware support. This section briefly considers two hardware kernels. The first is the transputer which was designed to execute occam2 programs efficiently, and the second is the Ada Tasking Coprocessor (ATAC).

In more recent years, there have been moves to support the Java Virtual Machine directly in hardware (for example, Sun Microsystems' picoJava processor (Sun Microsystems, 2000) or the aJ-100 from aJile Systems Inc (aJile Systems, 2000)). This support goes beyond the support for just concurrent execution and attempts to improve the performance over interpreting Java byte code.

16.4.1 The transputer and occam2

The transputer was designed as an occam2 machine which, on a single chip, has a 32-bit processor, a 64-bit floating-point coprocessor, internal memory and a number of communication links for direct connection to other transputers. An address bus joins external memory to the internal provision by means of a continuous address space. Typically a transputer will have 16 kbyte of internal memory; this acts as, in effect, a large collection of non-sharable registers for the executing processes.

The links are connected to the main processor via link interfaces. These interfaces can independently manage the communications of the link (including direct access to memory). As a result, a transputer can simultaneously communicate on all links (in both directions), execute an internal process and undertake a floating-point operation.

The transputer has a reduced instruction set but with an operations stack of only three registers. Each instruction has been designed to be of use in the code generation phase of the occam2 compiler; direct programming in assembler, although allowed, has not been taken into account in the design of the instruction set. Being a reduced instruction machine, not all instructions are immediately available; those that are directly accessible are precisely those that are commonly generated from real occam2 programs.

Unfortunately the transputer can only support a limited priority model. But by this restriction, a run-time support system that is essentially cast in silicon can be provided. The result of this architecture (plus the axiom that context switches only take place when the operations stack is empty) is very small context switch times.

Although the operational characteristics of a single transputer are impressive it is only when they are grouped together that their full potential is realized. Transputers use point-to-point communication, which has the disadvantage that a message may have to be forwarded to its destination via intermediates if no direct link is available. Nevertheless link transfer rates are very high and transmission failure rates very low, giving a real-time engine of considerable power and reliability.

16.4.2 ATAC and Ada

There have been several attempts to produce Ada machines, for example (Ericsson, 1986; Runner and Warshawsky, 1988). The one considered here is an Ada tasking coprocessor (ATAC) designed by Roos (1991).

The ATAC is a hardware device designed to support the Ada 83 tasking and clock models. It also anticipated some of the Ada 95 features such as support for priority inheritance and 'delay until'. It goal is to remove the burden of supporting Ada tasking from the application CPU, thereby allowing tasks to proceed efficiently without the overheads normally incurred by the Ada run-time support system.

Communication between CPU and the ATAC is based on standard memory-mapped read and write instructions. A set of primitive operations provide the interface; they include:

- `CreateTask` – create a new task;
- `ActTasks` – activate one or more created tasks;
- `Activated` – signal activation to creating task;
- `EnterTBlock` – enter a new task block;
- `ExitTBlock` – wait for dependent tasks to exit a task block;
- `EntryCall` – make an entry call;
- `TimedECall` – make a timed entry call;
- `SelectArg` – open a select alternative;
- `SelectRes` – choose an alternative in a select;
- `RndvCompl` – set caller runnable after the rendezvous is complete;
- `Activate` – make a suspended task runnable;
- `Suspend` – suspend the current task;
- `Switch` – perform a reschedule;
- `Delay` – delay a task.

The ATAC also fields all interrupt and interrupts the CPU only if a higher-priority task becomes runnable. An internal timer is used to support the Ada delay facilities and package calendar.

The overall goal of ATAC is to increase the performance of Ada tasking by two orders of magnitude over a pure software run-time system.

Summary

The execution environment is a key component of any implemented real-time system. It supports the application, but also introduces overheads and constrains the facilities that the application can use. A full-blown operating system (OS) could be used to provide the execution environment, but this is usually rejected due to:

- size of the OS (that is, memory occupancy);
- efficiency of key functions (such as context switching);
- complexity and hence reliability of the complete OS.

In this chapter it has been shown how an execution environment can be tailored to an application's specific needs, how its overheads can be modelled, and how hardware support can be provided. Other parts of the book have also introduced issues of significance to the execution environment. For example:

- its role in providing damage confinement (that is, firewalls);
- its role in error detection;
- its role in supporting communications in a distributed system.

The second issue has a number of facets. Various aspects of the application's execution can be monitored (array bounds violation, memory violation, time overruns). Also 'built-in test' facilities can be run in background mode to exercise parts of the hardware in order to isolate faulty components and generate maintenance data for fault removal.

As many features of an execution environment are important to a wide range of applications, there is a need to reuse trusted components and to move towards the provision of standard environments. The use of standardized languages and operating system interfaces will help to bring this about.

Further reading

Allen, R. K., Burns, A. and Wellings, A. J. (1995) Sporadic Tasks in Hard Real-Time Systems. *Ada Letters*, XV(5), 46–51.

Burns, A., Tindell, K. and Wellings, A. J. (1995) Effective Analysis for Engineering Real-Time Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 21(5), 475–480.

Venners, B. (1999) *Inside the Java 1.2 Virtual Machine*. New York: Osborne McGraw-Hill.

Exercises

16.1 Should the real-time system's programmer be aware of the implementation cost of all the implementation language's features?

16.2 Develop a model of clock handling which incorporates the three parameters CT^c, CT^s and CT^m (see Section 16.3.3).

16.3 Rather than using a clock interrupt to schedule periodic processes, what would be the ramifications of only having access to a real-time clock?

16.4 A periodic process of period 40 ms is controlled by a clock interrupt that has a granularity of 30 ms. How can the worst-case response time of this process be calculated?