Estimating Task Execution Times (2)

1.2 Cache

The cache is a fast memory inserted as a buffer between the CPU and the main memory (RAM)

Purpose: to reduce the bad effects on the speed of processes´execution stemming from the wide disparity between processor and memory cycle times

Problem: It is difficult to determine the success of a cache access (cache hit versus cache miss) since the cache contents are not easy to predict. Even extensive code analysis does not solve the problem.

Main reason: The existence of conditional branches and/or task preemption's

Solution approaches:
- Worst-case analysis resulting in a disabled cache
- Strategic Memory Allocation for Real Time (SMART)

2.1 Interrupts

In many operating systems, interrupts generated by I/O devices are served using a fixed priority scheme where the associated priorities are higher than process priorities

---> The delay introduced by the interrupt mechanism on the ET of tasks becomes unpredictable, i.e. (WC)ET of the driver task hard to predict

---> In RT systems, I/O devices must be handled differently.

---> Three possible techniques are sketched

a) Except the one for the timer, all external interrupts are disabled and all I/O devices must be handled by the application task itself

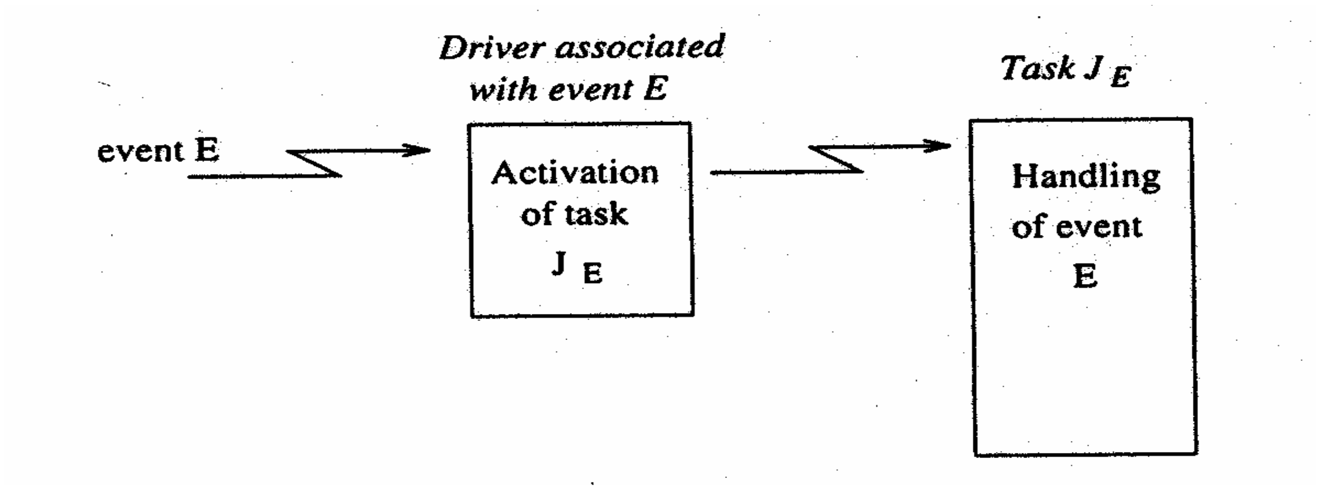
Main drawback: low processor efficiency on I/O operations because of busy waiting

Estimating Task Execution Times (3)

b) I/O devices are managed by dedicated kernel routines, periodically activated by the timer.
Main drawback: In principle, the same as in case a)

c) Leave all external interrupts enabled, but reduce the job of the associated drivers to a minimum, i.e. only to activate a proper task (which is scheduled just like another application task) that will take care of the interrupt routine.

Activation of a device-handling task:



Main advantage: Elimination of the busy wait due to I/O operations

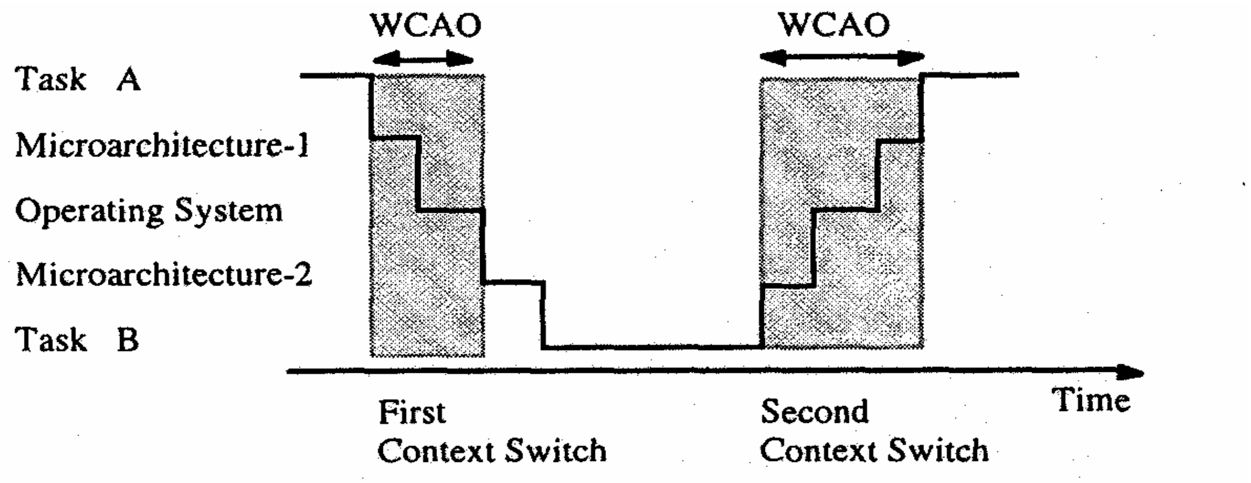
Estimating Task Execution Times (4)

2.2 Preemptions

If a task is preempted by another one, e.g. a higher priority task that must service a pending interrupt, then, its execution time is extended by:

- The ET of the preempting task
- The time required for two context switches

The administrative (OS) overhead of a task preemption (last two activities):



2.3 System calls

- Kernel calls should be characterized by a bounded execution time
- Each kernel primitive should be preemptable

2.4 Semaphores

The typical semaphore mechanism used in traditional operating systems is not suited because it is subject to the priority inversion phenomenon. Solutions for this problem are discussed more thoroughly later.

Estimating Task Execution Times (5)

2.5 Memory management

Virtual memory is not suitable to RT applications

---> Use of static allocation schemes like memory segmentation

Main drawback: Reduction of flexibility, especially in dynamic environments

3. Programming language

To meet predictability requirements in RT systems, several restrictions must be obeyed by the program:

- Time-bounded loops
- Absence of recursion
- Absence of dynamic data structures (arrays, pointers, strings etc)

4. Conclusion

At present, the systematic analysis of all the effects that determine the ET of a task is still in its infancy. However, since upper bounds on the ET (WCET) are the minimum that is needed in almost all hard RT applications, it is current practice to combine a number of diverse techniques:

- Measurement of the implementation environment (application code, task parameters, OS service times etc)
- Use of restricted HW/OS architecture and programming constructs
- Analysis of subproblems, e.g. WCET of the source program)
- Extensive testing

Real-Time (Paradigms) (32)

Problems with WCET’s:

- dependent on hardware architecture,OS, compiler, PL ---> difficult to predict
- many features serve to improve average case behavior, n o t worst case behavior

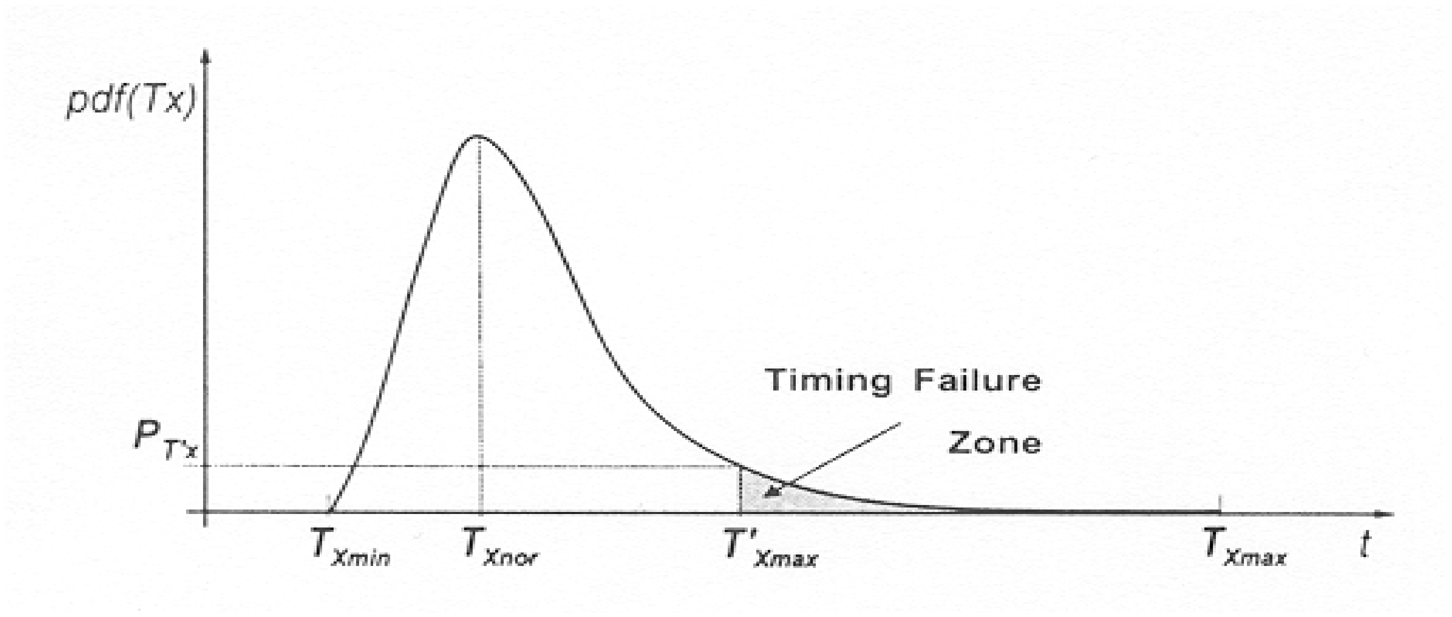
  Examples:
  - caches, pipelining, virtual memory
  - interrupt handling, preemptions
  - optimizing compilers
  - recursions

- **Even more difficult if depending on the environment (embedded systems) possibly resulting in very unpredictable release times (arrival patterns)**

Real-Time (Paradigms) (33)

Problem: What if applications with timing requirements can provide only uncertain timing parameters?

Distribution of Termination Times

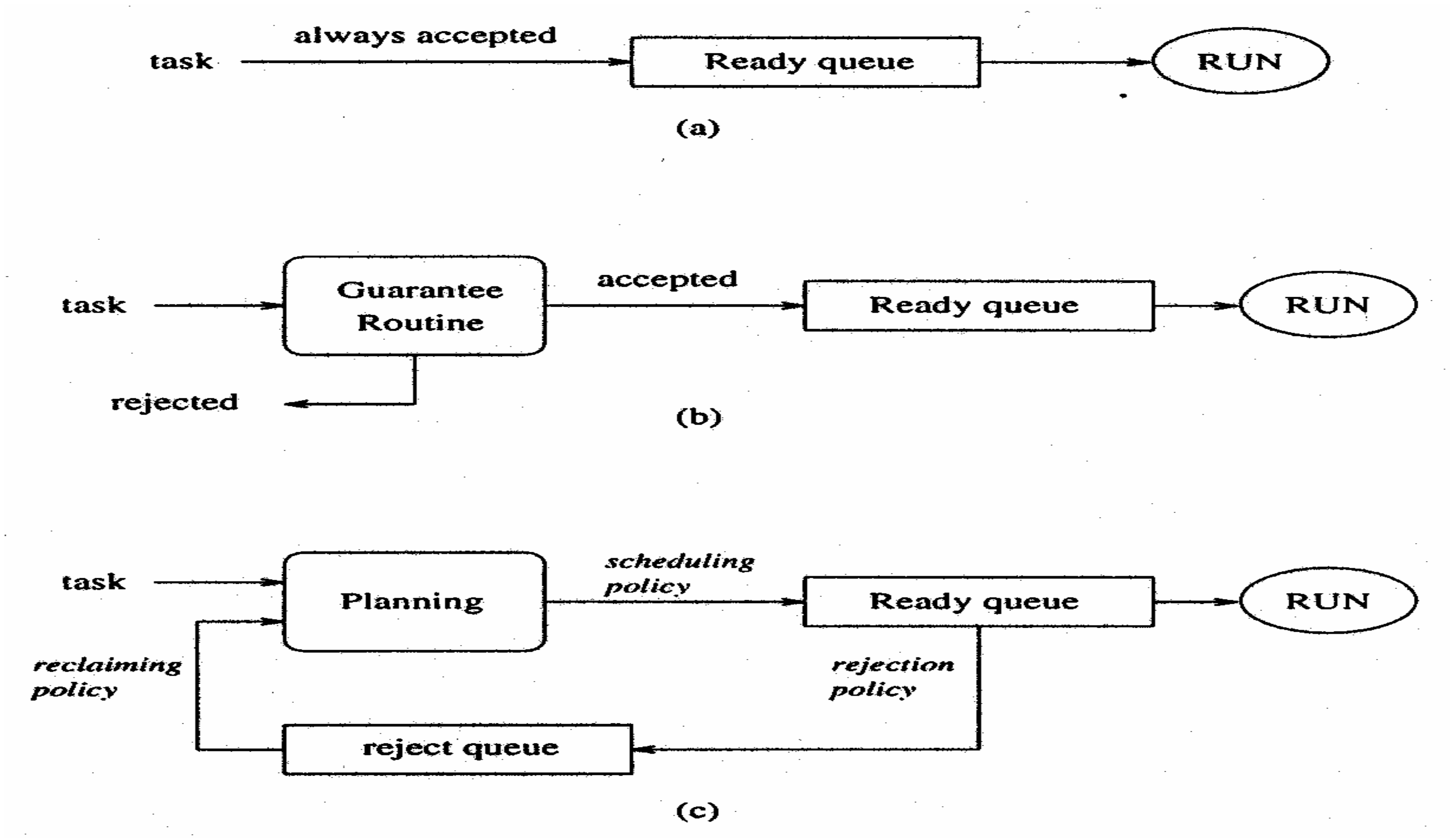


Two important classes of guarantee-based dynamic scheduling for overload situations

- *robust*
  - different policies for task acceptance and task scheduling
  - often using a reclaiming mechanism for accepted but later rejected tasks
- *fault-tolerant*
  - trading functional redundancy for predictability

Task (CPU) Scheduling (34)

Scheduling schemes for handling overload situations



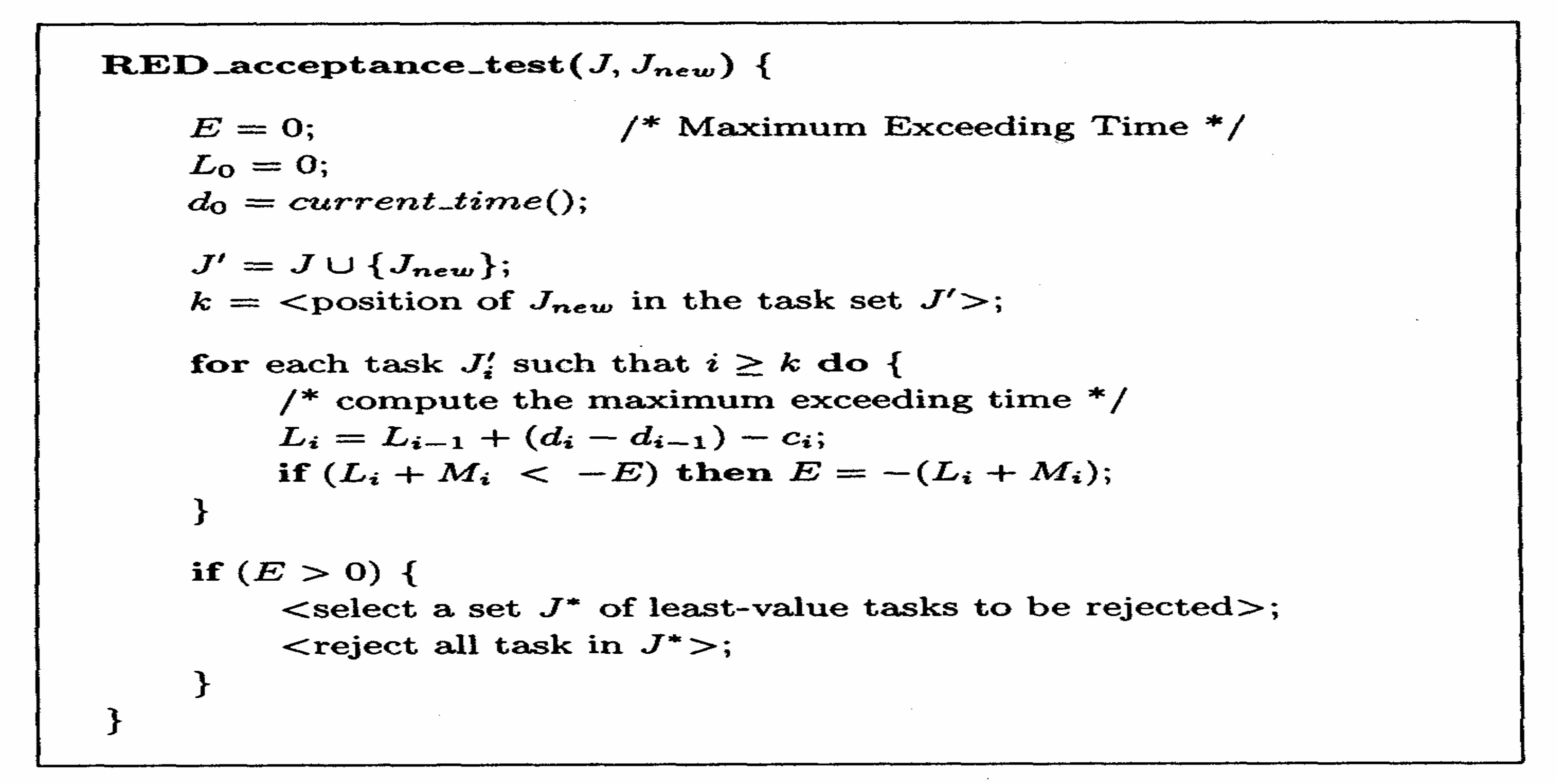
a) best effort b) simple guarantee c) robust

Task (CPU) Scheduling (35)

Example: The RED (Robust Earliest Deadline) algorithm (Buttazzo, Stankovic):

Each Task J_i is characterized by four parameters:

C_i	its worst-case execution time
D_i	its deadline
M_i	its deadline tolerance
V_i	its importance value



```
RED_acceptance_test($J$, $J_{new}$) {

    $E = 0$;                      /* Maximum Exceeding Time */
    $L_0 = 0$;
    $d_0$ = current_time();

    $J' = J \cup \{J_{new}\}$;
    $k$ = <position of $J_{new}$ in the task set $J'$>;

    for each task $J'_i$ such that $i \geq k$ do {
        /* compute the maximum exceeding time */
        $L_i = L_{i-1} + (d_i - d_{i-1}) - c_i$;
        if ($L_i + M_i < -E$) then $E = -(L_i + M_i)$;
    }

    if ($E > 0$) {
        <select a set $J^*$ of least-value tasks to be rejected>;
        <reject all task in $J^*$>;
    }
}
```