Scalable Applications for Energy-Aware processors

Giorgio Buttazzo

University of Pavia
Italy

Summary

- Context
- Objectives
- Problem identification
- Possible solutions
- Conclusions

Context

- Battery operated systems are very common today and will increase in the future:

  cell phones, video games, GPS, wearable computing, portable TV, videocameras, …

- Most of such systems operate under timing constraints to exhibit a desired performance

- Power consumption is also important for achieving long lifetime

Power consumption

- In CMOS circuits, the power consumption increases with the supply voltage:

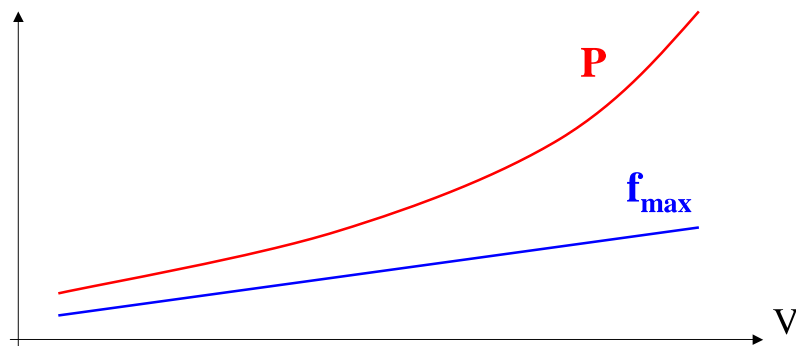
$$P \propto C_{load} \cdot f_c \cdot V_{dd}^2$$

- Moreover, the supply voltage also affects the circuit delay (hence the max clock frequency):

$$D \propto \frac{V_{dd}}{(V_{dd} - V_t)^2}$$

Voltage and speed

- Hence, the energy consumed by a system can be controlled by the speed and the voltage at which the processor operates:



Voltage variable processors

- To exploit such a possibility, next generation processors will be designed to work under different voltage levels

GOAL

Minimize the energy consumption while meeting task timing constraints

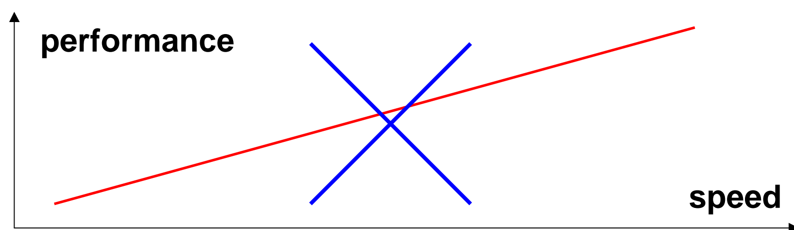
Existing results

- **[Yao et al, 1995]**
  Off-line scheduling to minimize total energy consumption
- **[Aydin 2001]**
  On line speed changes for periodic tasks with different power consumption characteristics
- **Melhem et al. (2002)**
  proposed several algorithms for reducing energy consumption in power-aware systems

Implicit assumption

All these algorithms implicitly assume that

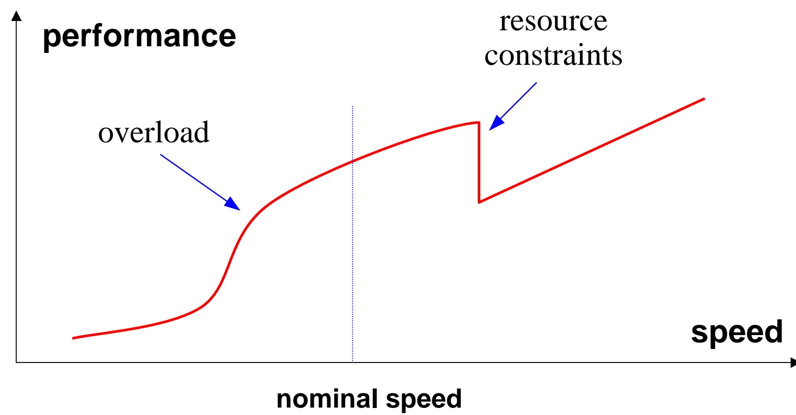
system performance monotonically increases with the processor speed:



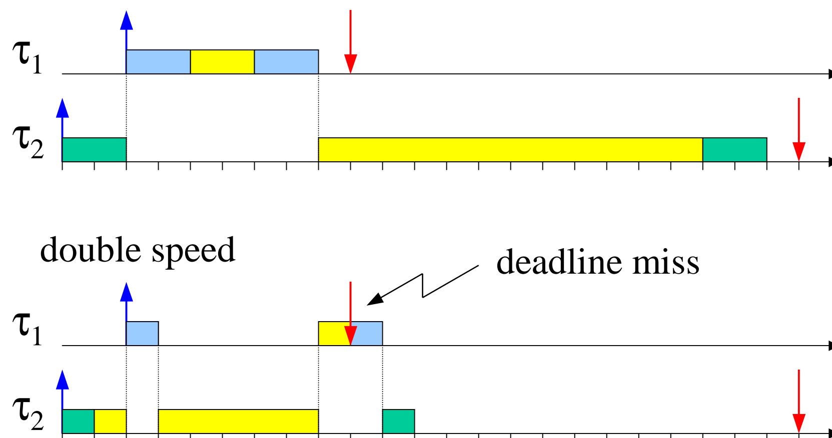
Unfortunately this assumption is WRONG!

Actual situation

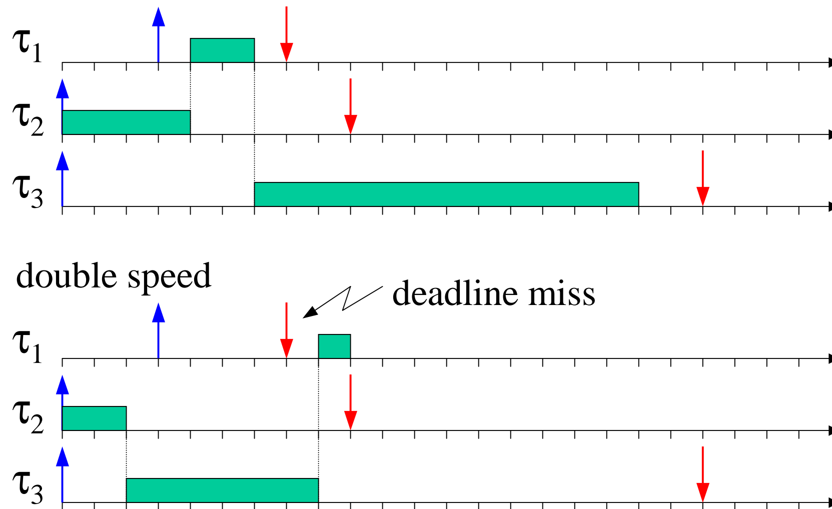
Performance may have abrupt changes and even be discontinuous



Running faster does not always improves performance



A case with non preemptive tasks



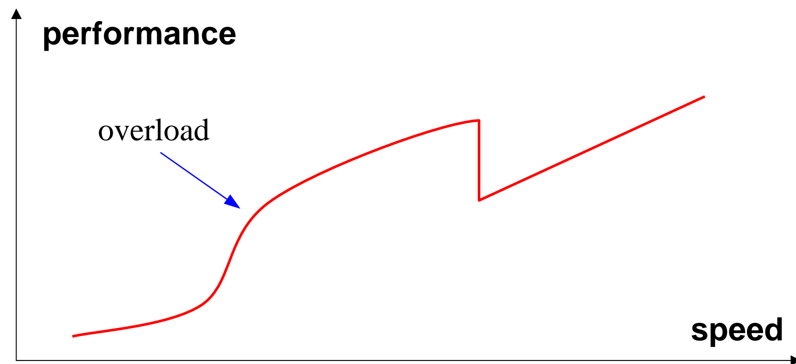
Note that

Non preemption is a special case of resource constraints

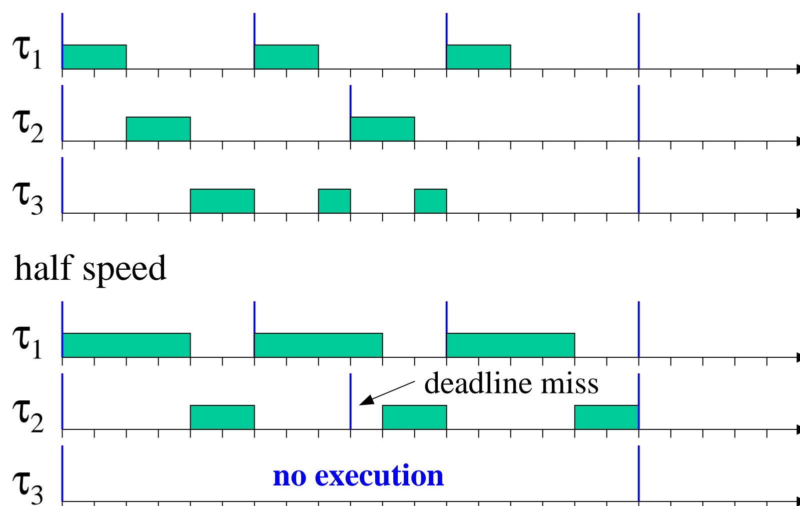
- Non preemptive tasks can be considered as preemptive tasks that share a single resource for their entire execution.

Effects of overload

Performance may have abrupt changes and even be discontinuous



Negative effects of overload



What do we learn?

To achieve scalability of performance with respect to speed, we need to avoid:

- non preemptive sections
- blocking (i.e., critical sections)
- overload conditions

How can we achieve these goals?

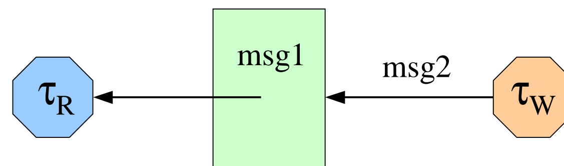
Avoiding blocking

Blocking for mutual exclusion can be avoided by using asynchronous communication buffers:

- ***State message*** semantics: messages are overridden by the sender and are not consumed by the receiver:
  - ⇒ a new message overwrites the previous one;
  - ⇒ the same message can be read more times.

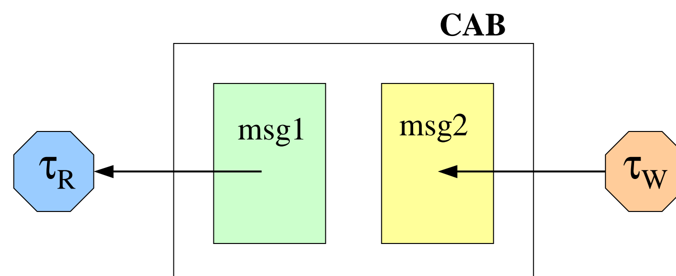
This is not enough

State message semantics avoids blocking for an empty/full queue but does not avoid blocking due to simultaneous accesses:



Cyclic Asynchronous Buffers (CABs)

If a writer task τ_W arrives while a task τ_R is reading, the new message is written in a new buffer:



Dimensioning a CAB

- To avoid blocking, if a CAB is shared by **N tasks**, it must have at least **N +1 buffers**.
- The (N+1)-th buffer is needed for keeping the most recent message in the case all the other buffers are used.

Accessing a CAB

- CABs are accessed through a memory pointer.
- Hence, a reader is not forced to copy the message in its memory space.
- More tasks can simultaneously read the same message.
- At each instant, a pointer (**mrd**) points to the most recent message stored in the CAB.

Writing Protocol

To write a message in a CAB a task must:

- **get a pointer** to a free buffer;
- **copy the message** into the buffer using the pointer;
- **release the pointer** to the CAB to make the message accessible to the next reader.

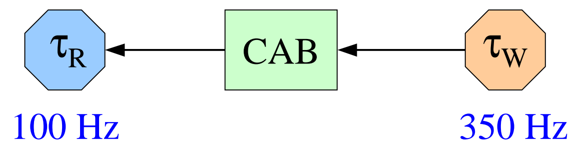
Reading Protocol

To read a message from a CAB a task must:

- **get the pointer** to the most recent message in the CAB;
- **process the message** through the pointer;
- **release the pointer**, to allow the CAB to recycle the buffer if it is not used.

NOTE THAT

- CABs can effectively be used to exchange messages among periodic tasks running at different rates:



- This type of mechanism is not considered in today's OS standards (e.g., OSEK).

Overload handling

Overload conditions in periodic systems due to speed reduction can be avoided by:

- Admission control (too drastic)
- Reducing precision of results
- Job skipping
- Rate adaptation

Reducing precision

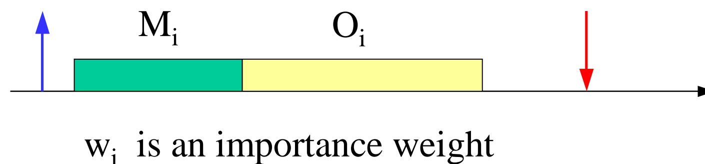
In many applications, computation can be performed at different level of precision: the higher the precision, the longer the computation. Examples are:

- binary search algorithms
- image processing and computer graphics
- neural learning

Imprecise computation

In this model, each task τ_i (C_i, D_i, w_i) is divided in two portions:

- a **mandatory** part: τ^m_i (M_i, D_i)
- an **optional** part: τ^o_i (O_i, D_i)

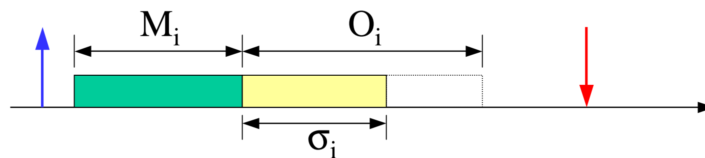


w_i is an importance weight

Imprecise computation

In this model, the workload can be reduced by aborting the optional part at any time:

error: $\varepsilon_i = O_i - \sigma_i$ **average error**: $\varepsilon_a = \sum_{i=1}^{n} w_i \varepsilon_i$



GOAL: minimize the average error

Job skipping

Periodic load can also be reduced by skipping some jobs, once in a while.

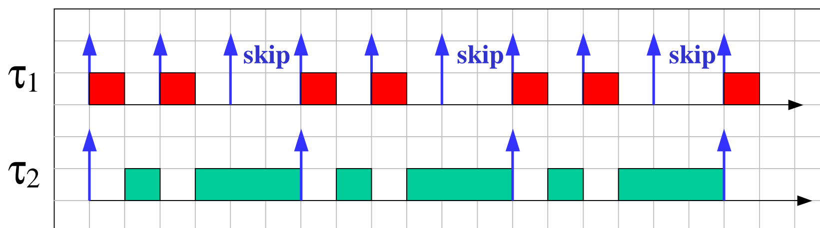
Many systems tolerate skips, if they do not occur too often:

- multimedia systems (video reproduction)
- inertial systems (robots)
- monitoring systems (sporadic data loss)

Example

The system is overloaded, but tasks can be schedulable if τ_1 skips one instance every 3:

$$U_p = \frac{1}{2} + \frac{4}{6} = 1.17 > 1$$



Rate adaptation

- The idea is to reduce the load by increasing deadlines and/or periods.
- Each task must specify a range of values for its period.
- Periods are increased during overloads, and reduced when the overload is over.

Example

task	C_i	T_{i0}
τ_1	10	20
τ_2	10	40
τ_3	15	70

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} = 0.96$$

Load adaptation

If τ_4 arrives with: $\mathbf{C_4 = 5, T_4 = 30}$ the system is not schedulable any more:

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} + \frac{5}{30} = 1.13$$

However, there exists a feasible schedule within the specified ranges:

$$U_p = \frac{10}{23} + \frac{10}{50} + \frac{15}{80} + \frac{5}{30} = 0.99$$

Finding feasible rates

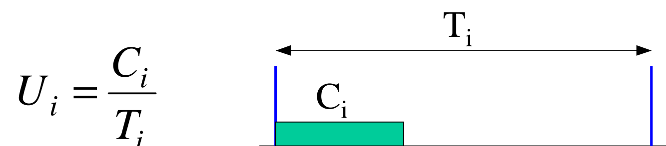
- There is a huge number of possible rate configurations giving a feasible schedule.
- How to find one efficiently?

THE ELASTIC TASK MODEL

Elastic task model

- Tasks' utilizations are treated as elastic springs and can be changed by period variations:

$$U_i = \frac{C_i}{T_i}$$

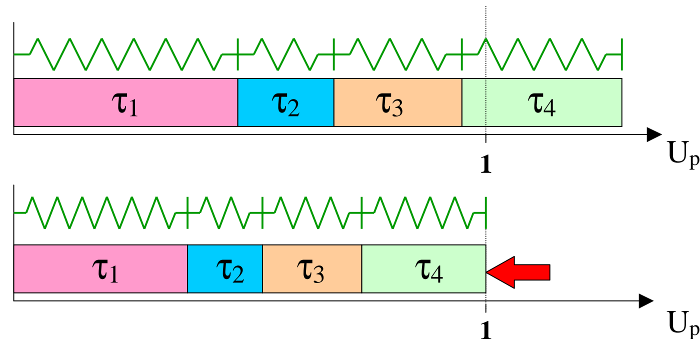


- The resistance of a task to a period variation is controlled by an **elastic coefficient E_i**:

  ⇒ the greater E_i the greater the elasticity

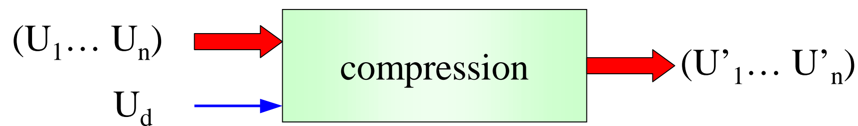
Compression algorithm

During overloads, utilizations must be compressed to bring the load below one.



Elastic task model

- Once new utilizations are computed, we can derive periods **OR** computation times



$$T'_i = \frac{C_i}{U'_i} \qquad \textbf{OR} \qquad C'_i = T_i \, U'_i$$

Conclusions

- To really exploit voltage variable processors, applications should be scalable
  (i.e., performance $\propto$ speed)
- Scalable applications can be developed if:
  - tasks are fully preemptive;
  - communication is non blocking (CABs);
  - overloads can be efficiently handled (e.g. elastic).