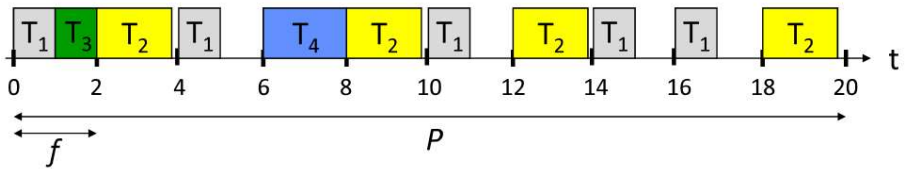
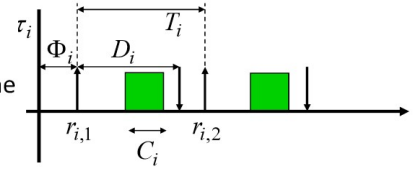
Time-Triggered Cyclic Executive Scheduler



→ Period P is partitioned into frames of length f

→ Definitions:

Γ : denotes the set of all periodic tasks
τ_i : denotes a periodic task
$\tau_{i,j}$: denotes the *j*th instance of task *i*
$r_{i,j}$, $d_{i,j}$: denote the release time and absolute deadline of the *j*th instance of task *i*
Φ_i : phase of task *i* (release time of its first instance)
D_i : relative deadline of task *i*



→ Assumptions: all tasks are periodic

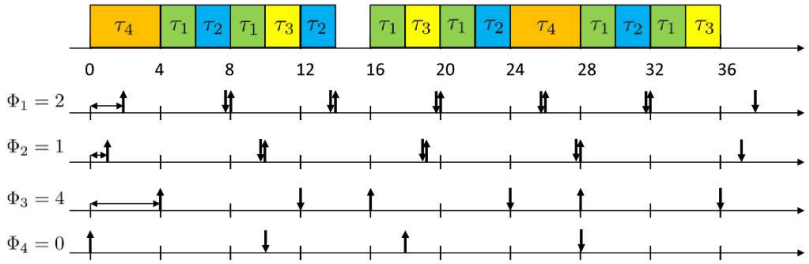
⇒ release time: $r_{ij} = \Phi_i + (j-1)T_i$

⇒ deadline: $d_{ij} = \Phi_i + (j-1)T_i + D_i$

- Constant WCET for all instances of a task i (WCET(i))

→ Example:

- $\tau_1 : T_1 = 6, D_1 = 6, C_1 = 2$ $\quad \tau_2 : T_2 = 9, D_2 = 9, C_2 = 2$
  $\tau_3 : T_3 = 12, D_3 = 8, C_3 = 2$ $\quad \tau_4 : T_4 = 18, D_4 = 10, C_1 = 4$
- $P = 36, f = 4$



→ Conditions:

- P is a multiple of f
- P is least common multiple of all periods T_i
- A task executes at most once in a frame

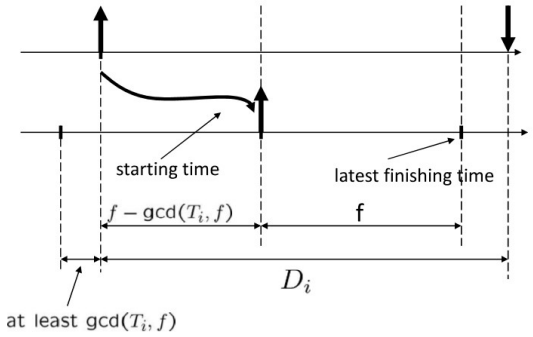
$$f \leq T_i \quad \forall \text{ tasks } \tau_i$$

- Tasks complete within a frame

$$f \geq C_i \quad \forall \text{ tasks } \tau_i$$

- At least one full frame between release and deadline

$$2f - \gcd(T_i, f) \leq D_i \quad \forall \text{ tasks } \tau_i$$

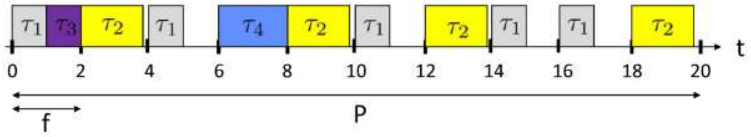


→ Check for correctness:

- f_{ij} = Frame number in which the j^{th} instance of task τ_i executes
- Phase $\Phi_i = \min_{1 \leq j \leq P/T_i} \{(f_{ij} - 1)f - (j-1)T_i\}$ $\forall$ tasks τ_i
- Respect all deadlines

$$(j-1)T_i + \Phi_i + D_i \geq f_{ij} f \quad \forall \text{ tasks } \tau_i,\ 1 \leq j \leq P/T_i$$

Γ	T_i	D_i	C_i
τ_1	4	4	1.0
τ_2	5	5	1.8
τ_3	20	20	1.0
τ_4	20	20	2.0

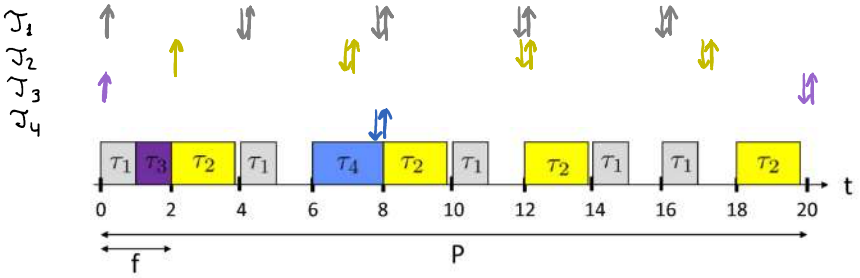


$$\Phi_1 = \min_{1 \le j \le 5} \{(f_{1j} - 1) f - (j-1) T_1\} = \min \begin{cases} 0-0 \\ 4-4 \\ 10-8 \\ 16-12 \\ 18-16 \end{cases} = 0$$

$$\Phi_2 = \min_{1 \le j \le 4} \{(f_{2j} - 1) f - (j-1) T_2\} = \min \begin{cases} 2-0 \\ 10-5 \\ 14-10 \\ 20-15 \end{cases} = 2$$

$$\Phi_3 = \min_{1 \le j \le 1} \{(f_{3j} - 1) f - (j-1) T_3\} = 0 \cdot f - 0 \cdot T_3 = 0$$

$$\Phi_4 = \min_{1 \le j \le 1} \{(f_{4j} - 1) f - (j-1) T_4\} = 8 - 0 \cdot T_4 = 8$$



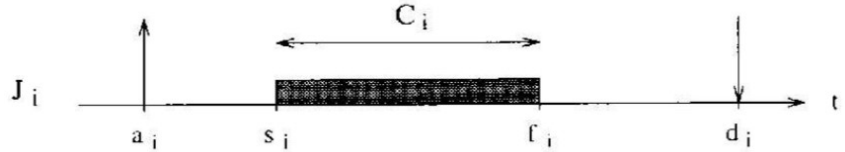
Aperiodic and Periodic Scheduling

→ Real-time Systems

- **Hard**: Missing a deadline may be catastrophic (sensor data acquisition, detection of critical conditions, actuator servoing etc.)
- **Soft**: Meeting a deadline is desirable but not necessary (user interface, printing information on the screen etc)

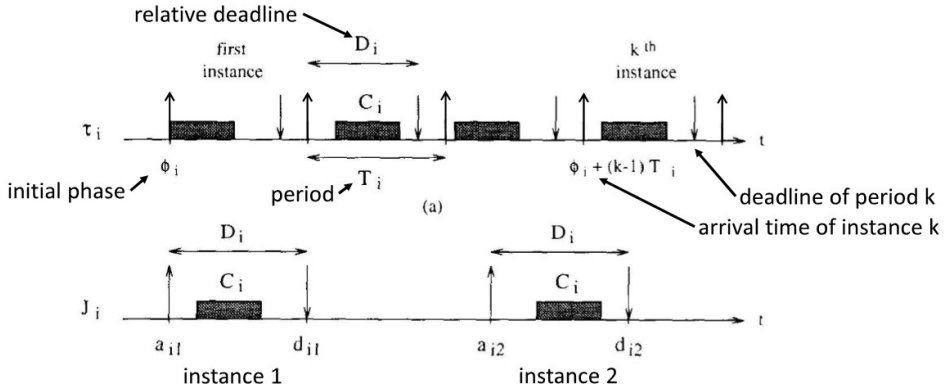
→ Schedule and Timing

- ***feaseable*** = task can be completed according to a set of specified constraints
- ***schedulable*** = there exists at least one algorithm that gives a feasable solution
- *Release time* r_i, a_i = When a task becomes ready for execution
- *Computation time* C_i = Processing time of a task
- *Deadline* d_i = When a task should be completed
- *Start time* s_i = When a task starts its execution
- *Finishing time* f_i = When a task finishes its execution
- *Lateness* L_i = $L_i := f_i - d_i < 0$ for task finishing before deadline
- *Tardiness* E_i = $E_i = \max(0, L_i)$
- *Laxity* X_i = $X_i := d_i - a_i - C_i$



$\Rightarrow E_i = 0$ (since $L_i < 0$)

- **Periodic task** τ_i: Periodic instances activated at a constant rate with period T_i. First activation happens at the phase Φ_i.



→ Classification of Scheduling Algorithms

- ***Preemptive***: Running tasks can be interrupted at anytime to assign the processor to another active task.
- ***Non-preemptive***: A task, once started, is executed until completion
- ***Static***: Scheduling decisions are based on fixed parameters, assigned to tasks before their activation
- ***Dynamic***: Scheduling decisions are based on dynamic parameters that may change during system execution

→ Metrics

- *Average response time*: $\overline{t}_r = \frac{1}{n} \sum_{i=1}^{n} (f_i - r_i)$
- *Total completion time*: $t_c = \max_i (f_i) - \min_i (r_i)$
- *Weighted sum of response time*: $\frac{\sum_{i=1}^{n} w_i (f_i - r_i)}{\sum_{i=1}^{n} w_i}$

- Maximum lateness: $L_{max} = \max_i (f_i - d_i)$
- Late tasks: $N_{late} = \sum_{i=1}^{n} miss(f_i)$, $miss(f_i) = \begin{cases} 0, & f_i \leq d_i \\ 1, & \text{otherwise} \end{cases}$

1. Aperiodic Scheduling

	Equal arrival times	Arbitrary arrival times
Independent tasks	EDD	EDF
Dependent tasks	LDF	EDF*

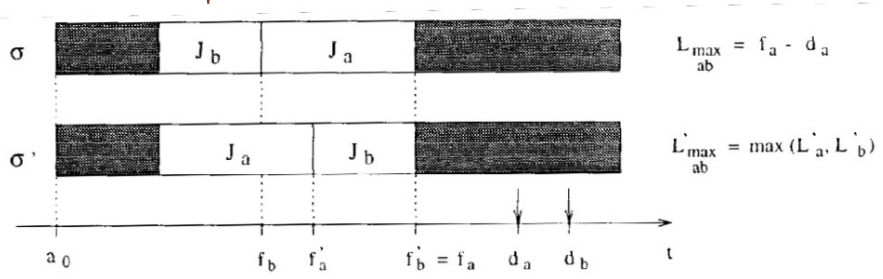
→ Earliest Deadline Due (EDD)

- For equal arrival times

> *Jackson's rule:* Given a set of n tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

- Just order tasks in increasing deadline

Proof of concept

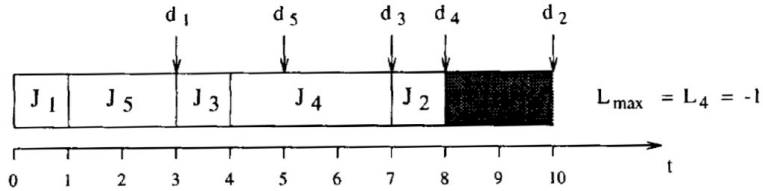


if $(L'_a \geq L'_b)$ then $L'_{max}_{ab} = f'_a - d_a < f_a - d_a$

if $(L'_a \leq L'_b)$ then $L'_{max}_{ab} = f'_b - d_b < f_a - d_a$

in both cases: $L'_{max}_{ab} < L_{max}_{ab}$

	J_1	J_2	J_3	J_4	J_5
C_i	1	1	1	3	2
d_i	3	10	7	8	5

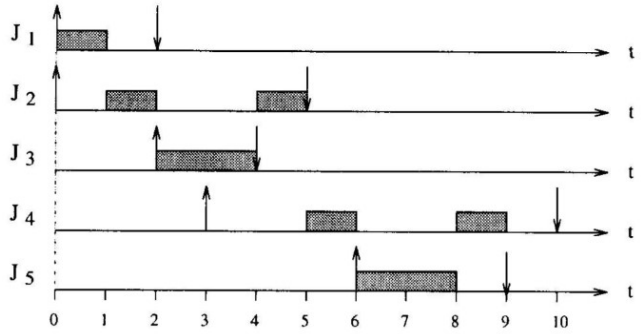


→ Earliest Deadline First (EDF)

- For arbitrary arrival times

> *Horn's rule:* Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9



Proof of concept

For each time interval $[t, t+1)$ it is verified, whether the actual running task is the one with the earliest absolute deadline. If this is not the case, the task with the earliest absolute deadline is executed in this interval instead. This operation cannot increase the maximum lateness.

- Worst case finishing time of task τ_i: $f_i = t + \sum_{k=1}^{i} C_k(t)$
- EDF guarantee condition: $\forall i = 1, \ldots, n \quad t + \sum_{k=1}^{i} C_k(t) \leq d_i$

→ Earliest Deadline First (EDF*)

- Schedule n tasks with precedence constraints

  * a task starts execution not earlier than its release time and not earlier than the finishing times of its predecessors (precedence constraint)

- Modification of deadlines and release times

  $J_a \rightarrow J_b \Rightarrow$ task τ_b depends on task τ_a

$$d_i^* = \min\left(d_i, \min\left(r_j^* - C_i : J_i \rightarrow J_j\right)\right)$$

$$r_j^* = \max\left(r_j, \max\left(r_i^* + C_i : J_i \rightarrow J_j\right)\right)$$

Algorithm for modification of release times:

1. For any initial node of the precedence graph set $r_i^* = r_i$
2. Select a task j such that its release time has not been modified but the release times of all immediate predecessors i have been modified. If no such task exists, exit.
3. Set $r_j^* = \max\left(r_j, \max\left(r_i^* + C_i : J_i \rightarrow J_j\right)\right)$
4. Return to step 2

Algorithm for modification of deadlines:

1. For any terminal node of the precedence graph set $d_i^* = d_i$
2. Select a task i such that its deadline has not been modified but the deadlines of all immediate successors j have been modified. If no such task exists, exit.
3. Set $d_i^* = \min\left(d_i, \min\left(d_j^* - C_j : J_i \rightarrow J_j\right)\right)$
4. Return to step 2

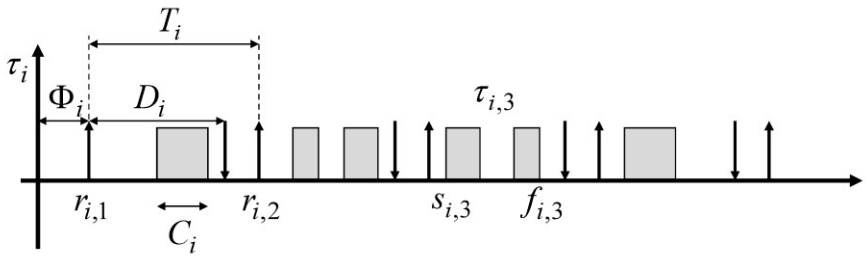
2. Periodic Scheduling

	Deadline equals period	Deadline smaller than period
Static priority	RM	DM
Dynamic priority	EDF	EDF*

→ Assumptions

- Periodic task activation $r_{i,j} = \Phi_i + (j-1)T_i$
- Same worst case execution time C_i and deadline D_i

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i \xrightarrow{D_i = T_i} d_{i,j} = \Phi_i + jT_i$$



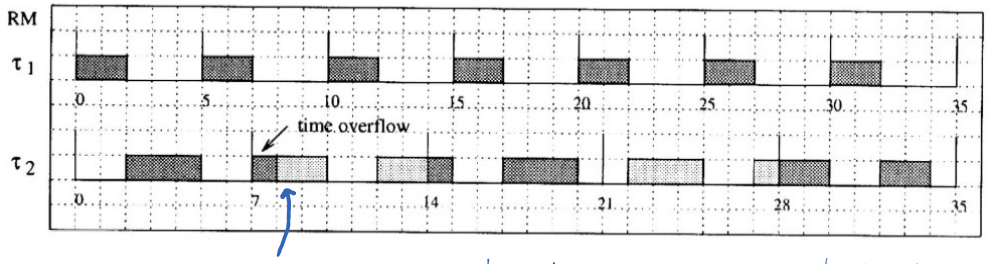
→ Rate Monotonic Scheduling (RM)

- Static priority assignment → assigned before execution
- Current task is preempted by a task with higher priority
- $D_i = T_i$
- Higher priorities given by shorter periods

Rate-Monotonic Scheduling Algorithm: Each task is assigned a priority. Tasks with higher request rates (that is with shorter periods) will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.

$\tau_1 \rightarrow C_1 = 2,\ T_1 = 5 \Rightarrow$ higher priority

$\tau_2 \rightarrow C_2 = 4,\ T_2 = 7$



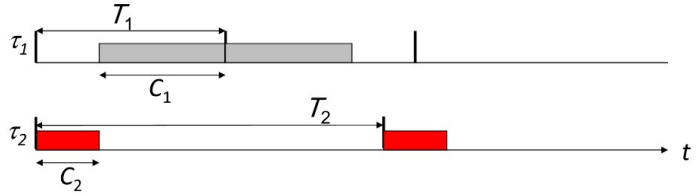
Since τ_1 has higher priority it will preempt task τ_2 (τ_2 will not manage to finish before the deadline)

Optimality: RM is optimal among all fixed-priority assignments in the sense that not other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM.

Proof of concept

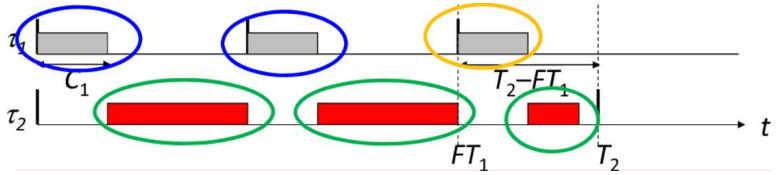
- $T_1 < T_2$ and $F := \lfloor T_2/T_1 \rfloor$

Case A: no RM



Schedule is feasable if $C_1 + C_2 \leq T_1$ and $C_2 \leq T_2$

Case B: with RM



Schedule is feasable if $FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$ (B)

Show (A) $\Rightarrow$ (B)

i. $C_1 + C_2 \leq T_1$ (multiply by F)

$F(C_1 + C_2) \leq FT_1 \Rightarrow FC_1 + FC_2 \leq FC_1 + C_2 \leq T_1$

ii. $FC_1 + C_2 \leq T_1$ (add $\min(T_2 - FT_1, C_1)$)

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq FT_1 + \min(T_2 - FT_1, C_1)$$
$$\leq \min(T_2, C_1)$$
$$\leq T_2$$

$C_1 + C_2 \leq T_1,\ C_2 \leq T_2 \Rightarrow FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2$

(A) (B)

■

- Schedulability Analysis: There exists a feasible schedule if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

↳ processor utilization factor U

* **It is sufficient but not necessary**

$U \leq n(2^{1/n} - 1) \Rightarrow$ feasible schedule exists

$U > n(2^{1/n} - 1) \Rightarrow$ there can be a feasable schedule

→ Deadline Monotonic Scheduling (DM)

- Assumption: $C_i \leq D_i \leq T_i$
- Current task is preempted by a task with higher priority
- Higher priorities given by shorter relative deadlines D_i not d_i as in EDF ↵

> *Algorithm:* Each task is assigned a priority. Tasks with smaller relative deadlines will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.

- Schedulability Analysis: There exists a feasible schedule if

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{1/n} - 1\right)$$

↳ ! not U (as in RM)

* It is sufficient but not necessary

- Schedulability Analysis 2: There exists a feasible schedule if and only if

$R_i \leq D_i \quad \forall$ task T_i

Find R_i that minimizes and fulfills $R_i = C_i + \underbrace{\sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j}_{I_i}$

	τ_1	τ_2	τ_3
C_i	1	3	2
T_i	3	8	9

Priority order: T_3, T_2, T_1

$R_3 = C_3 = 2 \qquad I_3 = \left\lceil \frac{R_3}{3} \right\rceil 1 + \left\lceil \frac{R_3}{8} \right\rceil 3 = 1 + 3 = 4 \qquad 2 \neq 2 + 4$

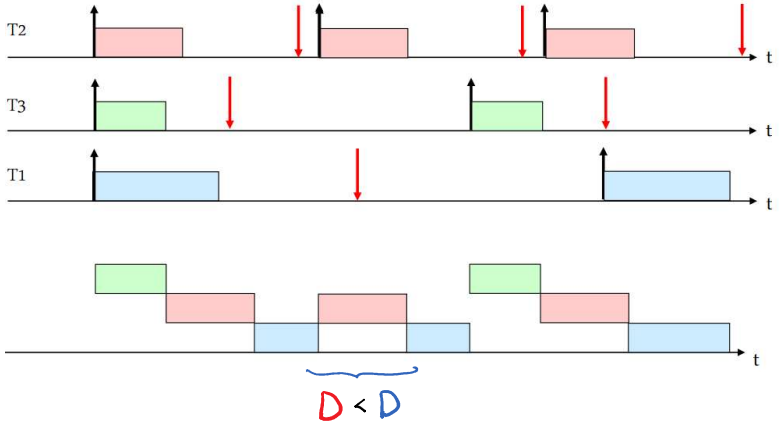
$R_3 = 2 + 4 = 6 \qquad I_3 = \left\lceil \frac{6}{3} \right\rceil 1 + \left\lceil \frac{6}{8} \right\rceil 3 = 2 + 3 = 5 \qquad 6 \neq 2 + 5$

$R_3 = 2 + 5 = 7 \qquad I_3 = \left\lceil \frac{7}{3} \right\rceil 1 + \left\lceil \frac{7}{8} \right\rceil 3 = 3 + 3 = 6 \qquad 8 = 2 + 6 \checkmark$

$\Rightarrow R_3 = 7 < 9 \checkmark$

→ Calculate all R_i and check $R_i \leq D_i$

* I_i considers only tasks with higher priorities
* Also valid for RM
* It is sufficient and necessary



→ Earliest Deadline First (EDF)

- Higher priorities given by shorter deadlines $d_{i,j}$

$$d_{i,j} = \Phi_i + (j-1)\,T_i + D_i$$

- No other algorithm can schedule a set of periodic tasks if the set cannot be scheduled by EDF
- Schedulability Analysis

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

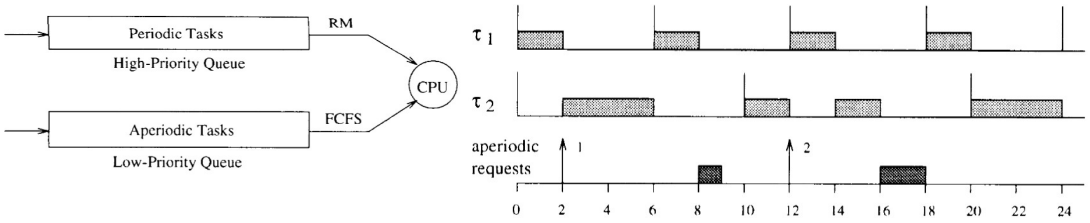
* $D_i = T_i$ It is sufficient and necessary
* $D_i < T_i$ It is sufficient but not necessary

3. Mixed Task Set

- *Periodic tasks: time-driven*, execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates.
- *Aperiodic tasks: event-driven*, may have hard, soft, non-real-time requirements depending on the specific application.
- *Sporadic tasks:* Offline guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is by assuming a *maximum arrival rate* for each critical event. Aperiodic tasks characterized by a minimum interarrival time are called sporadic.

→ Background Scheduling: Simple solution for RM and EDF

- Process aperiodic tasks if there are no pending periodic requests



→ Rate Monotonic Polling Server

- Introduce an artificial periodic tas whose purpose is to service aperiodic tasks as soon as possible
- Polling server (PS)
  - * At regular intervals equal to T_s, a PS task is instantiated.
  - * Maximum capacity of C_s
  - * Its priority (RM → period) can be chosen to match the response requirements
- Disadvantage: Long waiting if an aperiodic request arrives just after the server has suspended
- Schedulability Analysis: There exists a feasible schedule if

$$\frac{C_s}{T_s} + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq (n+1)\left(2^{\frac{1}{(n+1)}} - 1\right)$$

*** It is sufficient but not necessary**

- Guarantee the response time of aperiodic requests: Computation time C_a and deadline D_a

$$\left(1 + \left\lceil \frac{C_a}{C_s} \right\rceil\right) T_s \leq D_a$$

disadvantage: aperiodic request arrived just after server was suspended

How many server intervals are needed to process the aperiodic request

→ EDF - Total Bandwidth Server

- When the kth aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

C_k = execution time of the request

U_s = server utilization factor

- Once a deadline is assigned, the request is inserted into the ready queue as any other periodic instance
- Given a set of n periodic tasks with process utilization U_p and a total bandwidth server utilization U_s, the whole set is schedulable by EDF if and only if

$$U_p + U_s \leq 1$$

Shared Resources

→ Example: data structures, variables, set of registers, I/O unit...

→ No simultaneous access allowed ⇒ mutual exclusion
⇒ exclusive resources

• Some methods to protect exclusive resources include disabling interrupts, using semaphores and mutex.

→ **Semaphores**: Protect exclusive resources using semaphore S_i

J_i

≡

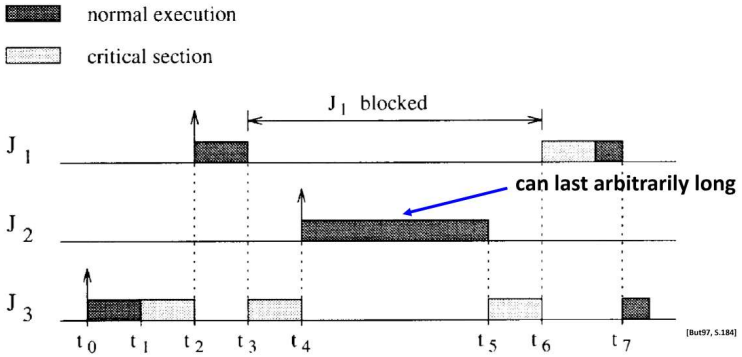
wait(S_1) → Protect resource: or spin (blocked) until it is released by the task using it

|

signal(S_1) → Release resource:

≡

→ Priority Inversion



• Solution: disallow preemption during execution of a critical section

→ Resource Access Protocols: Modify the priority of tasks that cause blocking

• When a task J_1 blocks one or more higher priority tasks, it temporarily assumes a higher priority

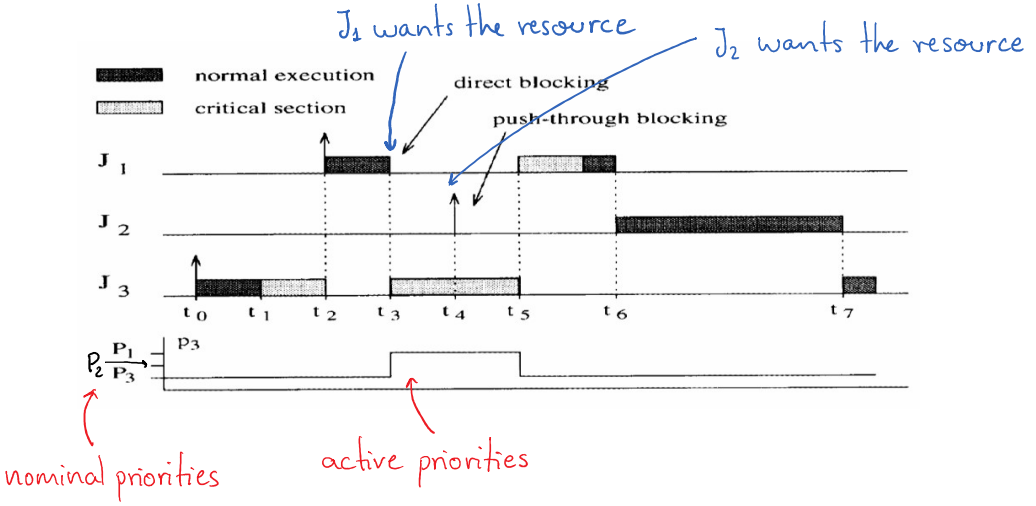
→ *Priority Inheritance Protocol* (PIP)

• **Assumptions**: n tasks which cooperate through m shared resources and all critical sections begin with wait(S_i) and end with signal(S_i)

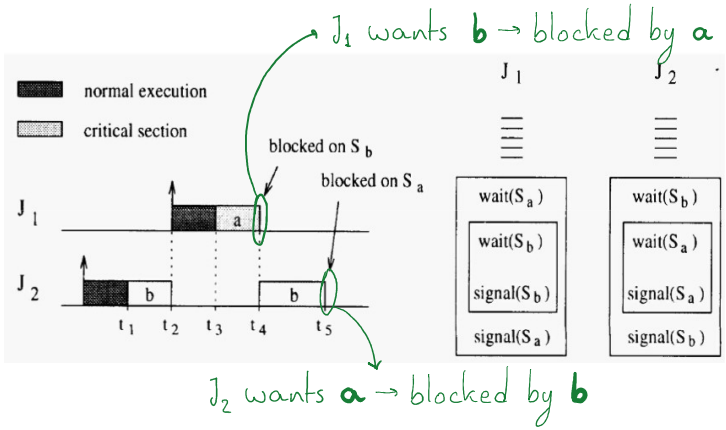
• **Idea**: When a task J_i blocks one or more higher priority tasks, it temporarily assumes the highest priority of the blocked tasks.

Algorithm:

- Jobs are scheduled based on their *active priorities*. Jobs with the same priority are executed in a FCFS discipline.
- When a job J_i tries to *enter a critical section* and the resource is blocked by a lower priority job, the job J_i is blocked. Otherwise it enters the critical section.
- When a job J_i is *blocked*, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k=p_i$ (it *inherits* the priority of the highest priority of the jobs blocked by it).
- When J_k exits a critical section, it *unlocks* the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k, then p_k is set to P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.
- Priority inheritance is *transitive*, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.



- **Problem: Deadlock**

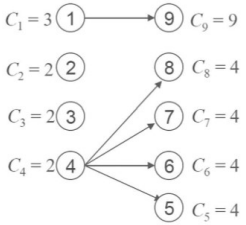


* Both J_1 and J_2 are blocked waiting for b and a
⇒ Deadlock

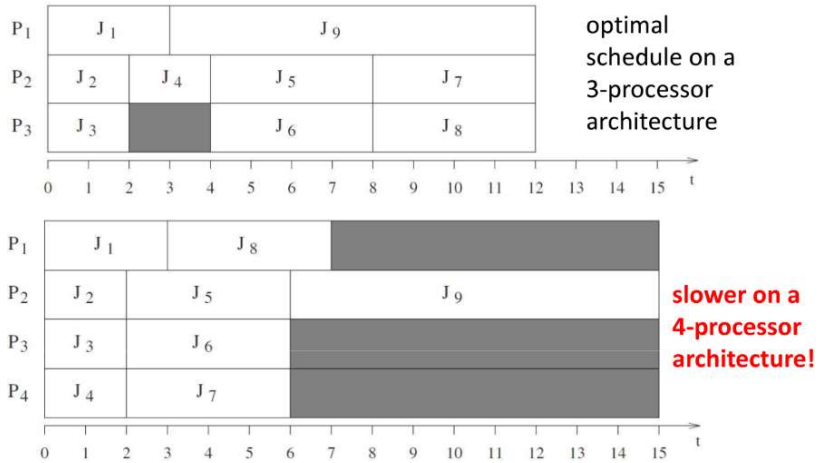
→ **Timing Anomalies**: Processing time and schedule feasability are not always improved when using faster or more processing units

Example: Richard's Anomalies

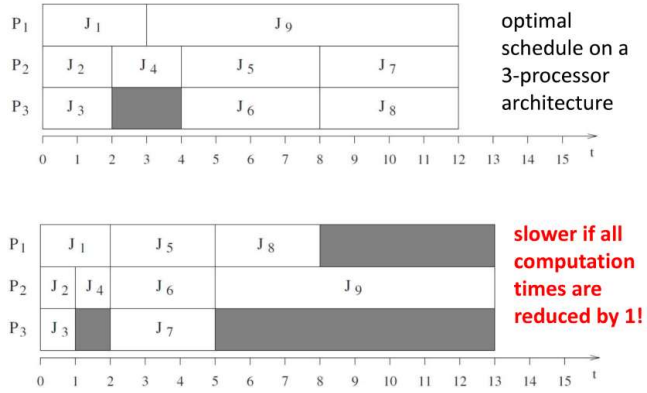
Consider the following procedence constraints



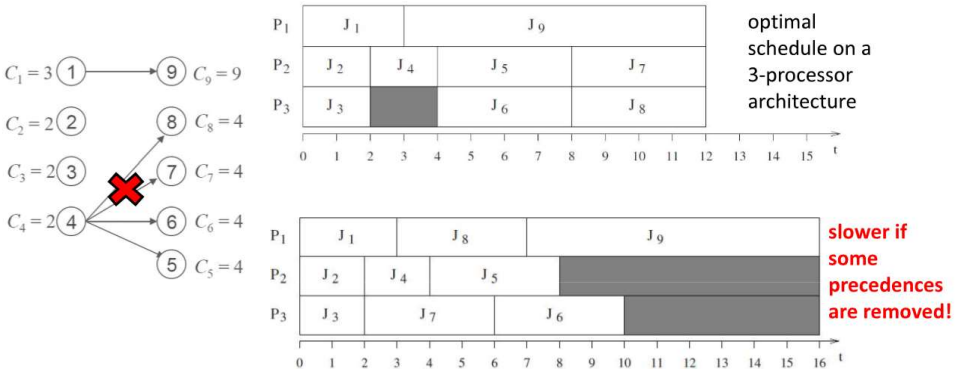
- More cores



- Faster processors

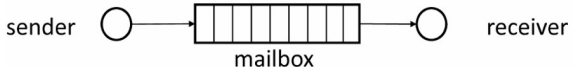


- Remove precedences



→ Communication and Synchronization

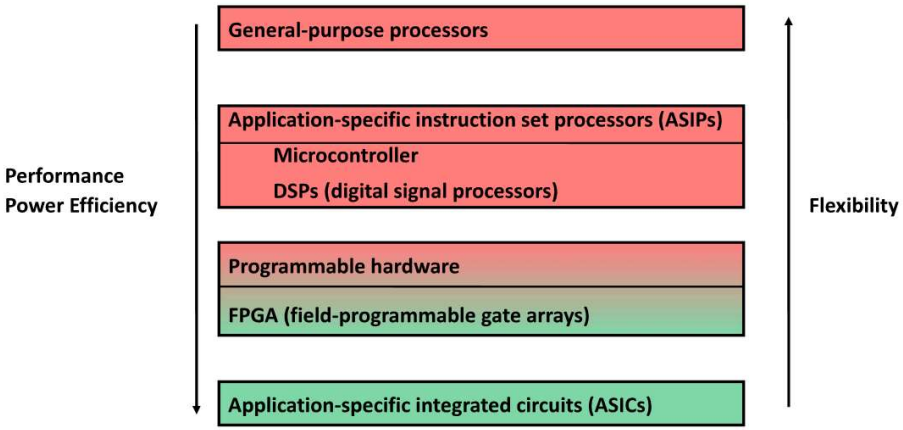
- **Synchronous communication**: Devices must be synchronized (wait for each other → rendez-vous)
  * **Problem**: estimating the blocking time for a process rendez-vous is difficult
- **Asynchronous communication**: Don't wait for each other. Deposit the message in a channel (queue) [limited space, read/write using FIFO principle)
  * **Problem**: Blocking behavior when queue is full.



Power and Energy

→ *„Power is considered as the most important constraint in embedded systems."* [in: L. Eggermont (ed): Embedded Systems Roadmap 2002, STW]

→ *"Power demands are increasing rapidly, yet battery capacity cannot keep up."* [in Diztel et al.: Power-Aware Architecting for data-dominated applications, 2007, Springer]



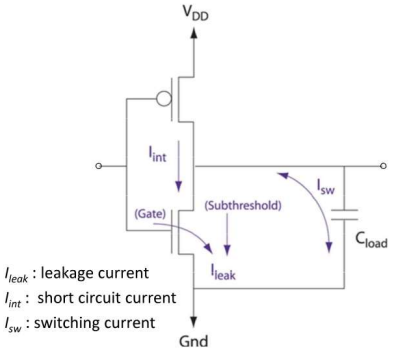
→ Power: Energy over time

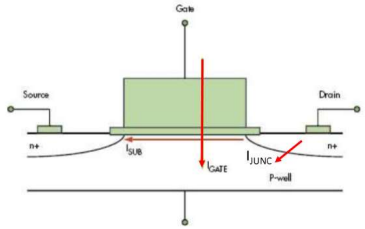
$$P(t) = \frac{dE}{dt} \Rightarrow E = \int P(t)\,dt$$

→ Minimizing **power consumption** is important for
- design of the power supply and voltage regulator
- dimensioning of interconnect
- cooling (high cost, limited space)

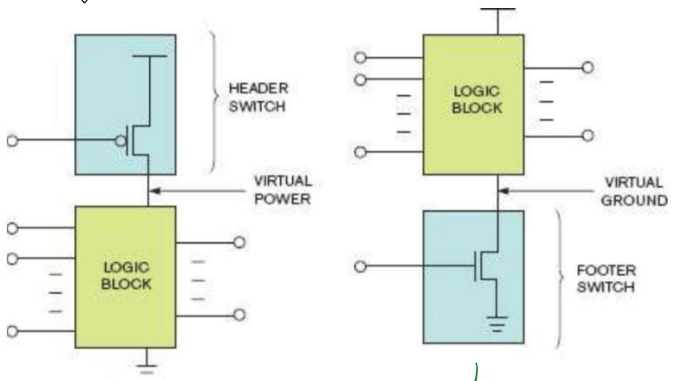
→ Minimizing **energy consumption** is important for
- restricted availability (mobile)
- limited battery capacities
- high costs, long lifetimes, low temperatures

→ Power Consumption of a CMOS Gate





- Dynamic power consumption
  - * Charging/discharging capacitors
  - * short circuit path between supply rails during switching
- Leakage and static power
  - * gate-oxide/subthreshold/junction leakage
  - * Material imperfections
- Reducing power: Power Supply Grating
  - * One of the most effective ways of reducing static power consumption (leakage)



Cut-off power supply to inactive units/components

1. Dynamic Voltage Scaling (DVS)

Average power consumption

$$P \sim \alpha C_L V_{dd}^2 f$$

Delay of a CMOS circuit

$$\tau \sim C_L \frac{V_{dd}}{(V_{dd} - V_T)^2}$$

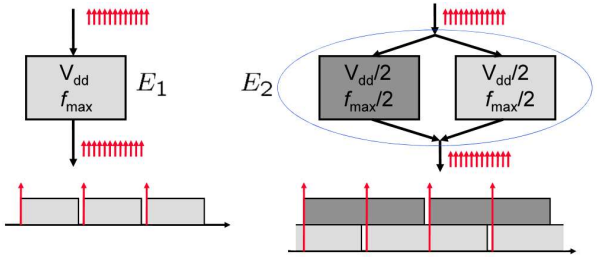
V_{dd} : supply voltage
α : switching activity
C_L : load capacity
f : clock frequency
V_T : threshold voltage ($V_{dd} \gg V_T$)

$$\Rightarrow E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#\text{cycles})$$

- Power: Quadratic dependence → V_{dd}
- Saving energy
  - → Reduce V_{dd}
  - → Reduce switching activity (α)
  - → Reduce load capacity (C_L)
  - → Reduce # cycles

→ Techniques to reduce dynamic power

- Minimize the dynamic power consumption by considering the quadratic voltage dependency
- Consider constant task effort (constant voltage and frequency for all tasks running)
- Parallelism

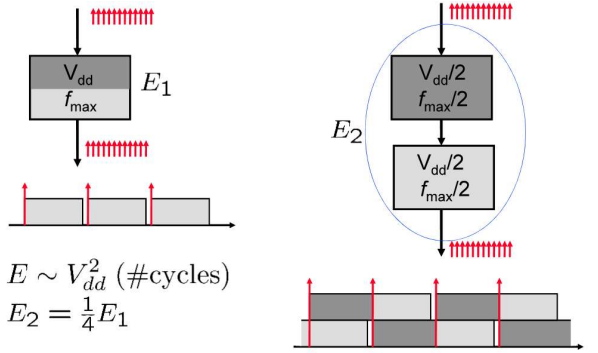


$$E_2 = 2 \cdot \left(\alpha C_L \left(\frac{V_{dd}}{2}\right)^2 \frac{f}{2}\right) = \frac{1}{4} \alpha C_L V_{dd}^2 f$$

$$E_2 = \frac{1}{4} E_1$$

- Pipelining

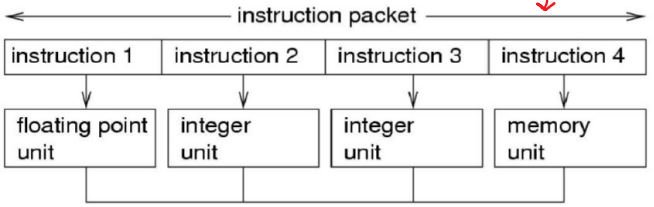
$$E_2 = \frac{1}{4} E_1$$



- VLIW (Very Long Instruction Word) Architecture
  - * Large degree of parallelism
  - * Simple hardware architecture (parallelization is done offline)

optimized by the compiler

all 4 instructions are executed in parallel



2. Dynamic Voltage and Frequency Scaling (DVFS)

→ Energy: $P \sim \alpha C_L V_{dd}^2 f \Rightarrow E \sim \alpha C_L V_{dd}$ (#cycles)

energy per cycle

→ Maximum frequency: $f \sim \frac{1}{\tau} \sim V_{dd}$ → reduce voltage ⇒ reduce clock frequency

gate delay

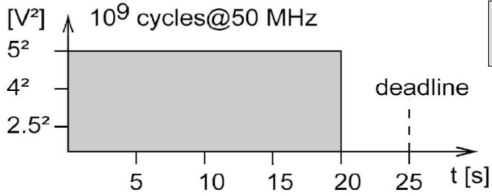
maximum frequency

→ Dynamically change voltage and frequency for a set of cycles

- Example:

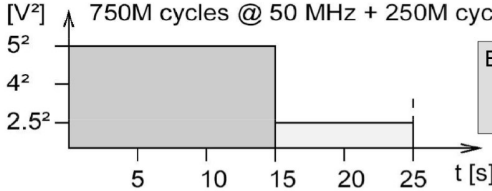
V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

a) [V²] 10^9 cycles@50 MHz



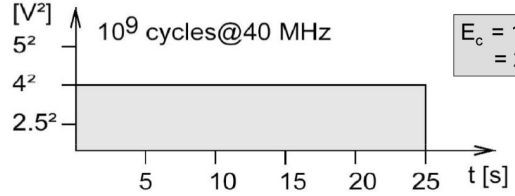
$E_a = 10^9 \times 40 \times 10^{-9} = 40$ [J]

b) [V²] 750M cycles @ 50 MHz + 250M cycles @ 25



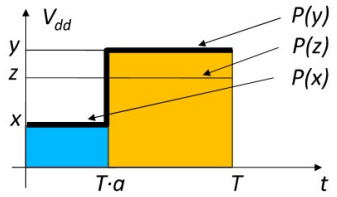
$E_b = 750\ 10^6 \times 40 \times 10^{-9} + 250\ 10^6 \times 10 \times 10^{-9} = 32.5$ [J]

c) [V²] 10^9 cycles@40 MHz



$E_c = 10^9 \times 25 \times 10^{-9} = 25$ [J]

→ Optimal strategy

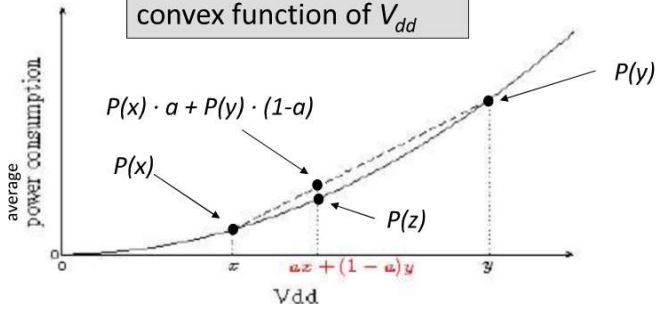


$z = a \cdot x + (1-a) \cdot y$

case A: Execute voltage x for a·T time units and at voltage y for (1-a)·T time units → $E_{tot} = T(P(x)a + P(y)(1-a))$

case B: Execute at voltage $z = ax + (1-a)y$ for T time units → $E_{tot} = TP(z)$

Dynamic power is a convex function of V_{dd}



- Running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling

⇒ **case A** is always worse if the power consumption is a convex function of the supply voltage (like $P \sim V_{dd}^2$)

→ Real-time offline scheduling

- Consider independent tasks $v_i \in V$ (absolute)
  - * Computation c_i, arrives at time a_i, deadline constraint d_i
- Objective: schedule all tasks such that all tasks finish before their deadline and energy consumption is minimized

⇒ YDS Algorithm

If possible, running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling.

→ YDS Optimal DVFS Algorithm for Offline Scheduling

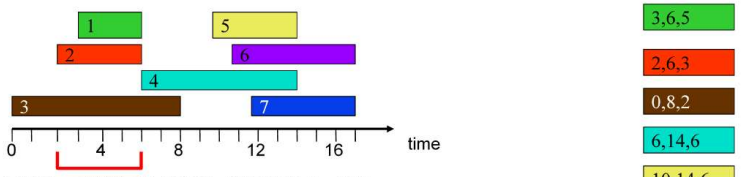
- Define *intensity* $G([z,z'])$ in some time interval $[z,z']$:
- Average accumulated execution time of all tasks that have arrival and deadline in $[z,z']$ relative to the length of the interval

$$V'([z,z']) = \{ v_i \in V \mid z \le a_i < d_i \le z' \}$$

$$G([z,z']) = \sum_{v_i \in V'([z,z'])} \frac{c_i}{z'-z}$$

• **Step 1**: Calculate all possible G([z,z']) combinations, where z and z' have to be an arrival and deadline time respectively

* Take the interval where G([z,z']) takes its highest value



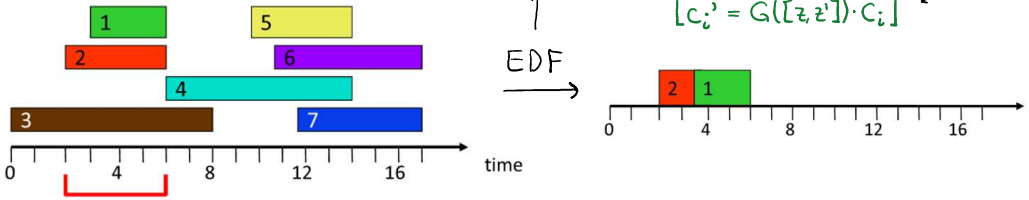
3,6,5
2,6,3
0,8,2
6,14,6
10,14,6
11,17,2
12,17,2
a_i,d_i,c_i

G([0,6]) = (5+3)/6=8/6, G([0,8]) = (5+3+2)/ (8-0) = 10/8,
G([0,14]) = (5+3+2+6+6)/14=11/7, G([0,17]) = (5+3+2+6+6+2+2)/17=26/17
highest G → G([2, 6]) = (5+3)/(6-2)=2 G([2,14]) = (5+3+6+6) / (14-2) = 5/3,
G([2,17]) = (5+3+6+6+2+2)/15=24/15
G([3,6]) =5/3, G([3,14]) = (5+6+6)/(14-3) = 17/11, G([3,17])=(5+6+6+2+2)/14=21/14
G([6,14]) = 12/(14-6)=12/8, G([6,17]) = (6+6+2+2)/(17-6)=16/11
G([10,14]) = 6/4, G([10,17]) = 10/7, G([11,17]) = 4/6, G([12,17]) = 2/5

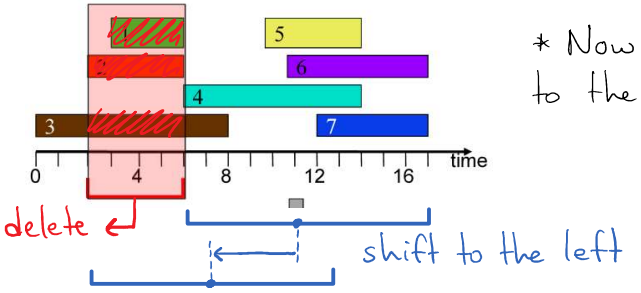
• **Step 2**: Schedule the tasks which are completely inside [z,z'] using EDF (for the G([z,z']) of Step 1) where G([z,z']) is the frequency

for 1 and 2

$[c_i' = G([z,z']) \cdot c_i]$

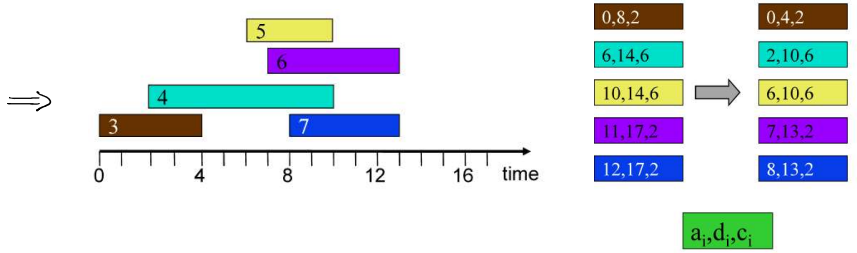


• **Step 3**: Delete everything that was inside this critical section [z,z']. This includes arrival, computation and deadline times



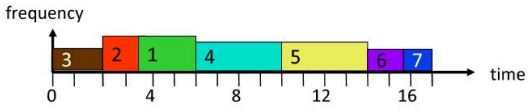
* Now shift what remains to the left (up t = z)

* Calculate the new arrival and deadline times



• **Step 4**: Repeat all these steps until all tasks are scheduled. These steps only give us the frequency at which each task should run

⋮

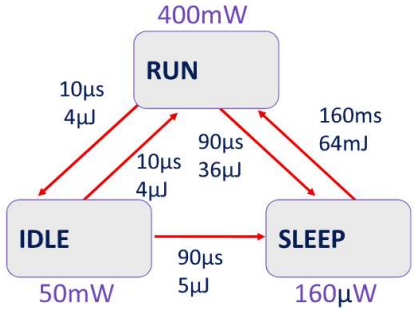


	v_1	v_2	v_3	v_4	v_5	v_6	v_7
frequency	2	2	1	1.5	1.5	4/3	4/3

3. Dynamic Power Management

→ Dynamic power management tries to assign optimal power saving states during program execution

• Example:

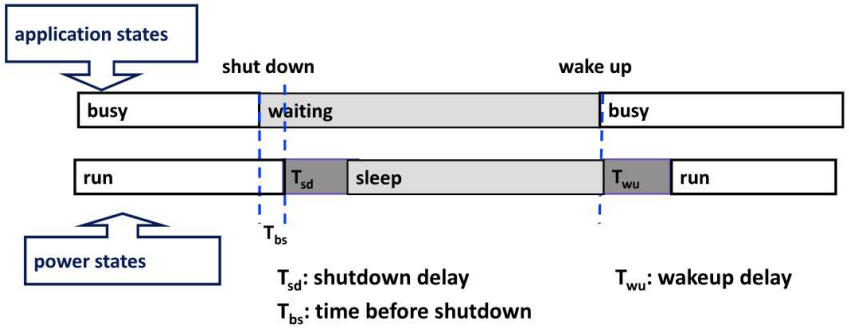


RUN: operational

IDLE: a SW routine may stop the CPU when not in use, while monitoring interrupts

SLEEP: Shutdown of on-chip activity

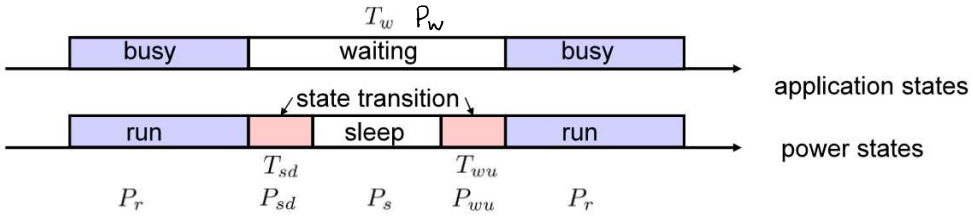
- Desired: Shutdown only during long idle times. This leads to a tradeoff between energy saving and overhead



- Because of all the delays, it's not always ideal to go to sleep mode when waiting

→ **Break-even Time**: The minimum idle time required to compensate the cost of entering an inactive (sleep) mode

- Enter an inactive state is beneficial only if the idle time is longer than the break-even time



no transition: $E_1 = T_w \cdot P_w$

transition: $E_2 = T_{sd} P_{sd} + T_{wu} P_{wu} + (T_w - T_{sd} - T_{wu}) P_s$

break-even: $E_2 \leq E_1 \Rightarrow T_w \geq \frac{T_{sd}(P_{sd} - P_s) + T_{wu}(P_{wu} - P_s)}{P_w - P_s}$

[time constraint: $T_w \geq T_{sd} + T_{wu}$]

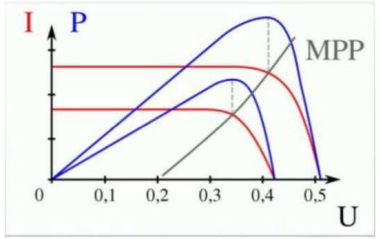
4. Battery-Operated Systems and Energy Harvesting

→ Battery operation: No continuous power source available, mobility

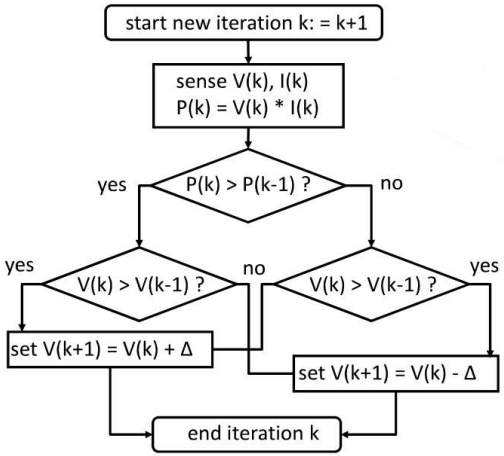
→ Energy harvesting: Prolong lifetime of battery-operated devices, rechargable batteries, autonomous operation

→ Power Point Tracking

- Maximize output power by changing the voltage
- Simple tracking algorithm (assume constant illumination)

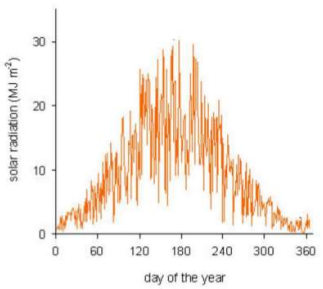


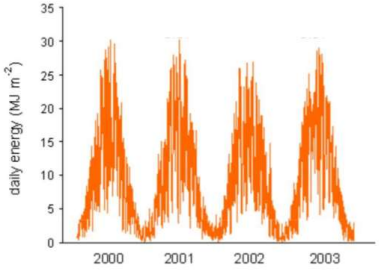
red: current for different light intensities
blue: power for different light intensities
grey: maximal power
tracking: determine optimal impedance seen by the solar panel



→ Challenges in (Solar) harvesting systems

- What is the optimal maximum battery capacity?
- What is the optimal area of the solar cell?
- How can we control the application such that a continuous system operation is possible, even under varying input energy?



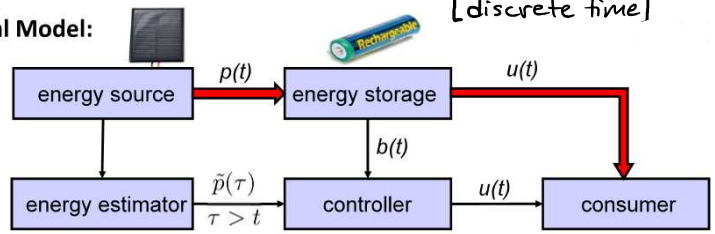


→ Application Control

- The controller can adapt the service of the consumer (sampling rate, transmission rate of information etc.) to proportionally change the power consumption.
  - Precondition: never run out of energy
  - Optimality: Maximize the lowest energy flow to the consumer

[discrete time]

Formal Model:



harvested/used energy in $[t, t+1]$: $p(t)$, $u(t)$

battery model: $b(t+1) = \min\{b(t) + p(t) - u(t), B\}$

failure state: $b(t) + p(t) - u(t) < 0$

utility: $U(t_1, t_2) = \sum_{t_1 \le \tau \le t_2} \mu(u(\tau))$ → μ is a strict concave function higher used energy gives a reduced reward for the overall utility

- Find an optimal control $u^*(t)$ for some interval with
  - i. $\forall\ 0 \le t < T : b^*(t) + p(t) - u^*(t) \ge 0$
  - ii. There is no feasable $u(t)$ with a larger minimal energy
    $\forall u : \min_{0 \le t \le T}\{u(t)\} \le \min_{0 \le t \le T}\{u^*(t)\}$
  - iii. We consider battery periodicity $b^*(0) = b^*(T)$

- Theorem: Given a use function $u^*(t)$ such that the system never enters a failure state. $u^*(t)$ is optimal with respect to maximizing the minimal used energy among all use functions and maximizes the utility $U(t,T)$ if and only if [for all $\tau \in (t,T)$]

$$u^*(\tau-1) < u^*(\tau) \implies b^*(\tau) = 0 \rightarrow \text{empty}$$
and
$$u^*(\tau-1) > u^*(\tau) \implies b^*(\tau) = B \rightarrow \text{full}$$

- Algorithm: linear programming
  - Suppose the utility is simply $U(0,T) = \sum_{0 \le \tau < T} u(\tau)$
  - Linear program has the form

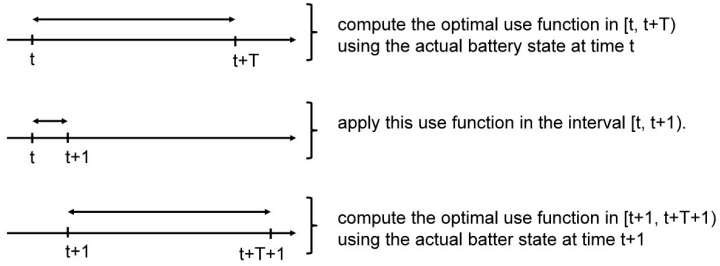
$$\begin{aligned}
&\text{maximize} \sum_{0 \le \tau < T} u(\tau) \\
&\forall \tau \in [0,T) \ : \ b(\tau+1) = b(\tau) - u(\tau) + \tilde{p}(\tau) \\
&\forall \tau \in [0,T) \ : \ 0 \le b(\tau+1) \le B \\
&\forall \tau \in [0,T) \ : \ u(\tau) \ge 0 \\
&b(T) = b(0) = b_0
\end{aligned}$$

- Algorithm: Finite horizon control

i. At time t, we compute the optimal control using $b(t)$ with predictions $\tilde{p}(\tau)$ for all $t \le \tau < t+T$ and $b(t+T) = b(t)$

ii. From the computed optimal $u(\tau)$ for all $t \le \tau < t+T$ we just take the first use value $u(t)$ in order to control the application

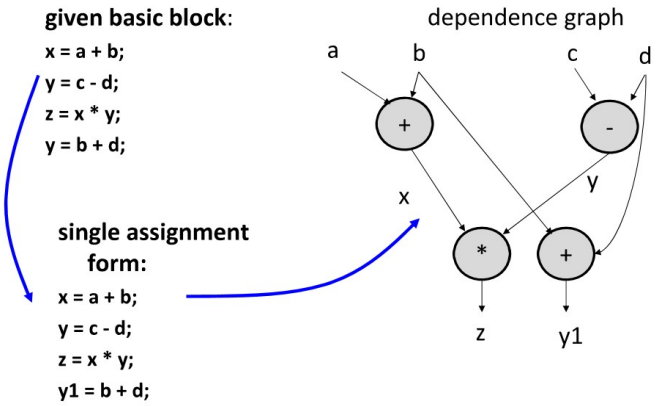
iii. At the next time step, we take as initial battery state the actual state. Therefore we take mispredictions into account. For the estimated future energy, we also take the new estimations



Architecture Synthesis

→ Major tasks of architecture synthesis

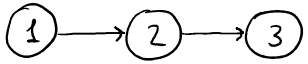
- *allocation* - determine the necessary hardware components
- *scheduling* - determine the timing of individual operations
- *binding* - determine the relation between individual operations of the algorithm and hardware resources



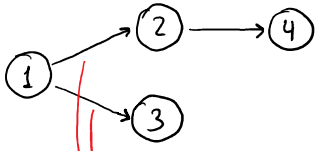
1. Task Graph or Dependence Graph (DG)

→ A *dependence graph* is a directed graph $G = (V, E)$ in which $E \subseteq V \times V$ is a partial order

execution order is not unique



... unique



who executes first ?

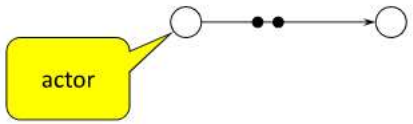
... not unique

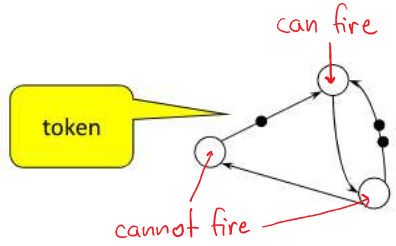
- If $(v_1, v_2) \in E$, then v_1 is called an *immediate predecessor* of v_2 and v_2 is called an *immediate successor* of v_1
- A dependence graph describes order relations for the execution of single operations or tasks.
- Nodes correspond to tasks or operations, edges correspond to relations („executed after")
- A dependence graph is **acyclic**
- Often, there are additional quantities associated to edges or nodes
  * execution times, deadlines, arrival times
  * communication demand

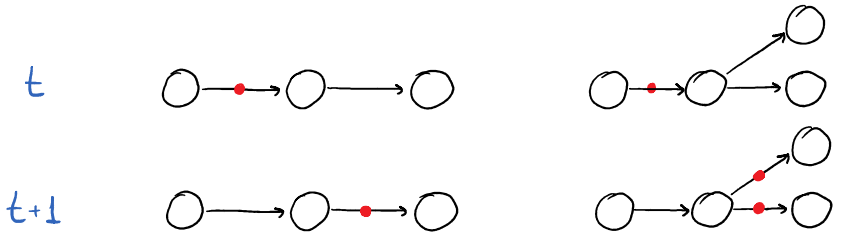
2. Marked Graphs (MG)

→ A *marked graph* $G = (V, A, del)$ consists of

- nodes (actors) $v \in V$
- edges $a = (v_1, v_2) \in A$, $A \subseteq V \times V$
- number of initial tokens on edges $del: A \to \mathbb{Z}^{\geq 0}$





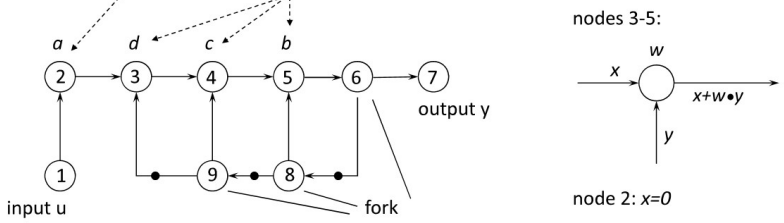


- the token on the edges correspond to data (FIFO queues)
- a node is **activated** if there is at least one token on every input edge
- The *firing* of a node removes from each input edge a token and adds a token to each output edge

- Filter equation:

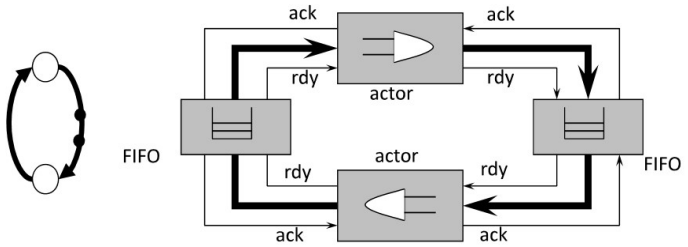
$$y(l) = a \cdot u(l) + b \cdot y(l-1) + c \cdot y(l-2) + d \cdot y(l-3)$$

- Possible model as a *marked graph*:



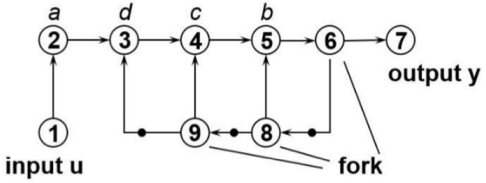
→ *Hardware Implementation*

- Edges correspond to synchronously clocked shift registers (FIFO)



→ *Software Implementation* [with static scheduling]

- At first, a feasible sequence of actor firing is determined which ends in the starting state (initial distribution of tokens)



(1, 2, 3, 9, 4, 8, 5, 6, 7)

```
while(true) {
      t1 = read(u);
      t2 = a*t1;
      t3 = t2+d*t9;
      t9 = t8;
      t4 = t3+c*t9;
      t8 = t6;
      t5 = t4+b*t8;
      t6 = t5;
      write(y, t6);}
```

→ *Software Implementation* [with dynamic scheduling]

- Scheduling is done using a real-time operating system
- Actors correspond to tasks (threads)
- After firing, the thread is put into the wait state. It is put into the ready state if all inputs are present

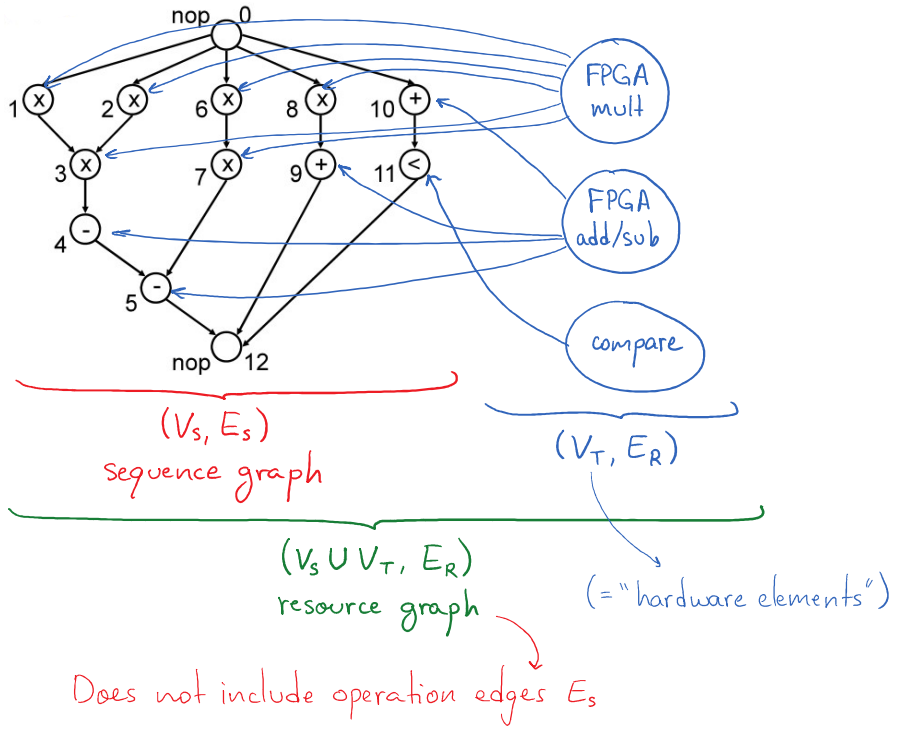
3. Models for Architecture Synkesis

→ A **sequence graph** $G_S(V_S, E_S)$ is a dependence graph with a single start node and a single end node. V_S denotes the operations of the algorithm and E_S denotes the dependence relations

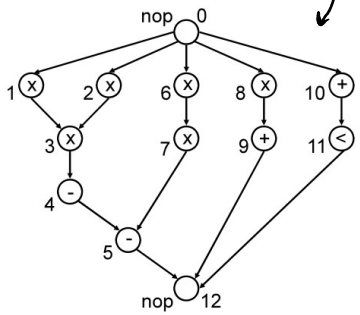
→ A **resource graph** $G_R(V_R, E_R)$, $V_R = V_S \cup V_T$ models resources and bindings. V_T denote the resource types of the architecture and G_R is a bipartite graph. An edge $(v_s, v_T) \in E_R$ represents the availability of a resource type v_T for an operation v_s.

→ **cost function** $c: V_T \to \mathbb{Z}$

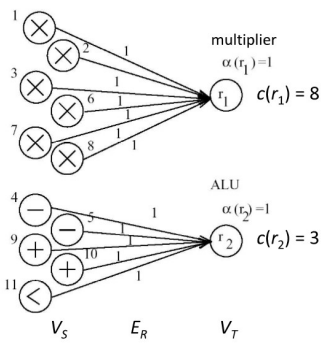
→ **Execution times** $w: E_R \to \mathbb{Z}^{\geq 0}$ are assigned to each edge $(v_s, v_T) \in E_R$ and denote the execution time of operation $v_T \in V_T$ on resource type $v_s \in V_S$



```
int diffeq(int x, int y, int u, int dx, int a) {
  int x1, u1, y1;
  while ( x < a ) {
    x1 = x + dx;
    u1 = u - (3 * x * u * dx) - (3 * y * dx);
    y1 = y + u * dx;
    x = x1;
    u = u1;
    y = y1;
  }
  return y;
}
```



sequence graph



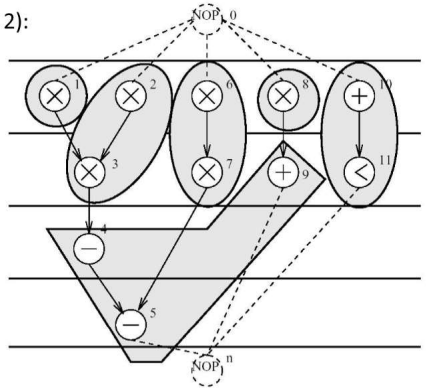
resource graph

→ **Allocation**: An allocation is a function $\alpha: V_T \to \mathbb{Z}^{\geq 0}$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances

→ **Binding**: A binding is defined by functions $\beta: V_S \to V_T$ and $\gamma: V_S \to \mathbb{Z}^{>0}$. Here, $\beta(v_s) = v_t$ and $\gamma(v_s) = r$ denote that operation $v_s \in V_S$ is implemented on the rth instance of resource type $v_t \in V_T$

Example binding ($\alpha(r_1) = 4, \alpha(r_2) = 2$):

$\beta(v_1) = r_1, \gamma(v_1) = 1,$
$\beta(v_2) = r_1, \gamma(v_2) = 2,$
$\beta(v_3) = r_1, \gamma(v_3) = 2,$
$\beta(v_4) = r_2, \gamma(v_4) = 1,$
$\beta(v_5) = r_2, \gamma(v_5) = 1,$
$\beta(v_6) = r_1, \gamma(v_6) = 3,$
$\beta(v_7) = r_1, \gamma(v_7) = 3,$
$\beta(v_8) = r_1, \gamma(v_8) = 4,$
$\beta(v_9) = r_2, \gamma(v_9) = 1,$
$\beta(v_{10}) = r_2, \gamma(v_{10}) = 2,$
$\beta(v_{11}) = r_2, \gamma(v_{11}) = 2$



↳ use second instance of resource r_2

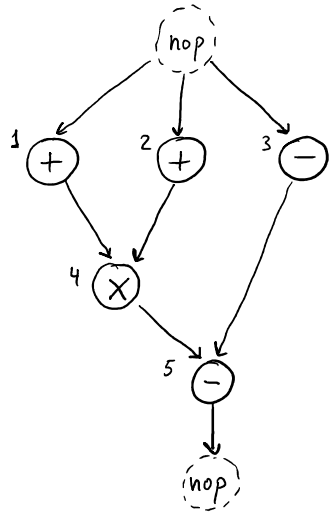
$\gamma(v_i) \leq \alpha(\beta(v_i))$

→ **Scheduling**: A schedule is a function $\tau: V_S \to \mathbb{Z}^{\geq 0}$ that determines the starting times of operations. A schedule is feasible if the conditions

$$\tau(v_j) - \tau(v_i) \geq \omega(v_i) \quad \forall (v_i, v_j) \in E_S$$

are satisfied. $\omega(v_i) = \omega(v_i, \beta(v_i))$ denotes the execution time of operation v_i

→ **Latency**: The latency L of a schedule is the time difference between start node v_0 and end node v_n. $L = \tau(v_n) - \tau(v_0)$



considering $\omega(v_i) = 2 \ \forall \ i$ and one single resource for all operations

$\tau(4) \geq \tau(1) + 2$

$\tau(4) \geq \tau(2) + 2$

$\tau(5) \geq \tau(4) + 2$

$\tau(5) \geq \tau(3) + 2$

$\tau(0) = 0. \quad \tau(1) = 2. \quad \tau(2) = 4. \quad \tau(3) = 6. \quad \tau(4) = 8. \quad \tau(5) = 10$

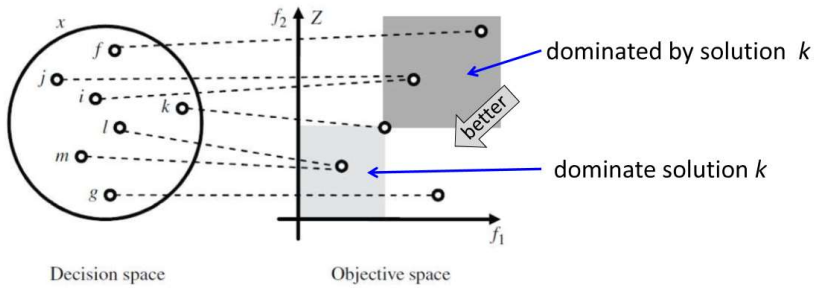
$\Rightarrow \tau(v_n) = 12 \Rightarrow L = \tau(v_n) - \tau(0) = \underline{12}$

4. Multiobjective Optimization

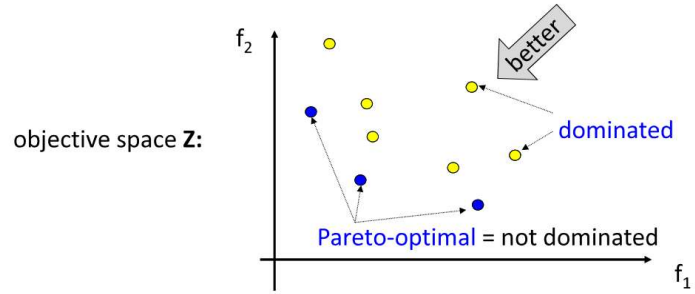
→ Architecture Synthesis is an optimization problem with more than one objective

- Latency of the algorithm, hardware cost, power and energy consumption

→ *Pareto-Dominance*: A solution $a \in X$ weakly Pareto-dominates a solution $b \in X$, denotes as $a \preceq b$, if it is at least as good in all objectives, i.e., $f_i(a) \leq f(b)$ for all $1 \leq i \leq n$. Solution a is better than b, denoted as $a \prec b$, iff $(a \preceq b) \wedge (b \not\preceq a)$



- A solution is named *Pareto-optimal*, if it is not Pareto-dominated by any other solution in X



→ **Synthesis Algorithms**

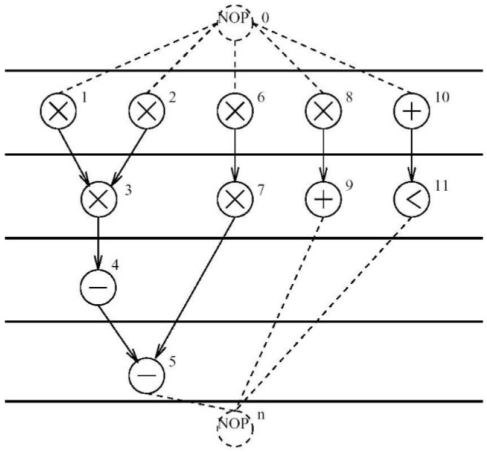
- *Unlimited resources*: No constraints in terms of available resources are defined
- *Iterative algorithms*: An initial solution to the architecture synthesis is improved step by step

→ **Scheduling without resource constraints**

- Given is a sequence graph $G_S = (V_S, E_S)$ and a resource graph $G_R = (V_R, E_R)$. Then the latency minimization without resource constraints is defined as

$$L = \min \{ \tau(v_n) : \tau(v_j) - \tau(v_i) \geq \omega(v_i) \ \forall (v_i, v_j) \in E_S \}$$

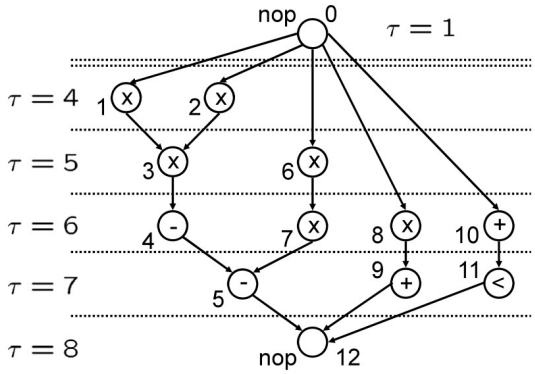
- *ASAP* (as soon as possible)




```
ASAP(G_S(V_S, E_S), w) {
    τ(v_0) = 1;
    REPEAT {
        Determine v_i whose predec. are planed;
        τ(v_i) = max{τ(v_j) + w(v_j) ∀(v_j, v_i) ∈ E_S}
    } UNTIL (v_n is planned);
    RETURN (τ);
}
```

→ $\omega(v_i) = 1$

- *ALAP* (as late as possible)




```
ALAP(G_S(V_S, E_S), w, L_max) {
    τ(v_n) = L_max + 1;
    REPEAT {
        Determine v_i whose succ. are planed;
        τ(v_i) = min{τ(v_j) ∀(v_i, v_j) ∈ E_S} − w(v_i)
    } UNTIL (v_0 is planned);
    RETURN (τ);
}
```

→ $L_{max} = 7$
$\omega(v_i) = 1$

→ **Scheduling with time constraints**

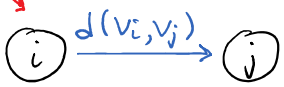
- *deadline* (latest finishing times of operations) like $\tau(v_2) + \omega(v_2) \leq 5$
- *release times* (earliest starting times) like $\tau(v_3) > 4$
- *relative constraints*: difference between starting times of a pair of operations) like

$$\tau(v_6) - \tau(v_7) \geq 4$$
$$\tau(v_4) - \tau(v_1) \leq 2$$

- We can model all types of constraints using only relative constraints

* minimum	$\tau(v_j) \geq \tau(v_i) + l_{ij}$	$\Rightarrow$	$\tau(v_j) - \tau(v_i) \geq l_{ij}$
* maximum	$\tau(v_j) \leq \tau(v_i) + l_{ij}$	$\Rightarrow$	$\tau(v_i) - \tau(v_j) \geq -l_{ij}$
* equality	$\tau(v_j) = \tau(v_i) + l_{ij}$	$\Rightarrow$	$\tau(v_j) - \tau(v_i) \geq l_{ij}$ $\wedge$ $\tau(v_i) - \tau(v_j) \geq -l_{ij}$

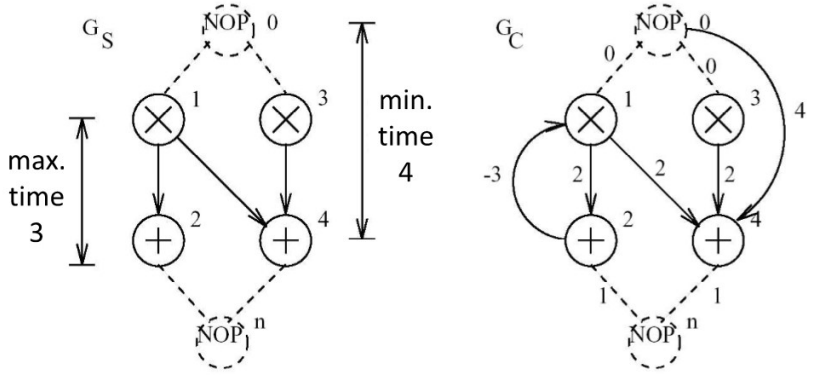
- **Weighted constraint graph**: A weighted constraint graph $G_c = (V_c, E_c, d)$ related to a sequence graph $G_s = (V_s, E_s)$ contains nodes $V_c = V_s$ and a weighted edge for each timing constraint. An edge $(v_i, v_j) \in E_c$ with weight $d(v_i, v_j)$ denotes the constraint $\tau(v_j) - \tau(v_i) \geq d(v_i, v_j)$



$w(v_1) = w(v_3) = 2 \qquad w(v_2) = w(v_4) = 1$

$\tau(v_0) = \tau(v_1) = \tau(v_3) = 1,\ \tau(v_2) = 3,$

$\tau(v_4) = 5,\ \tau(v_n) = 6,\ L = \tau(v_n) - \tau(v_0) = 5$



- Given is a sequence graph $G_s(V_s, E_s)$, a resource graph $G_R(V_R, E_R)$ and an associated allocation α and binding β Then the minimal latency is defined as

$$L = \min \{ \tau(v_n) : (\tau(v_j) - \tau(v_i) \geq \omega(v_i, \beta(v_i))\ \forall (v_i, v_j) \in E_s) \wedge (|\{v_s : \beta(v_s) = v_t \wedge \tau(v_s) \leq t < \tau(v_s) + \omega(v_s, v_t)\}| \leq \alpha(v_t)\ \forall v_t \in V_T,\ \forall 1 \leq t \leq L_{max}) \}$$

„Dependencies are respected. There are not more than the available resources in use at any moment in time and for any resource type."
(L_{max} = upper bound on the latency)

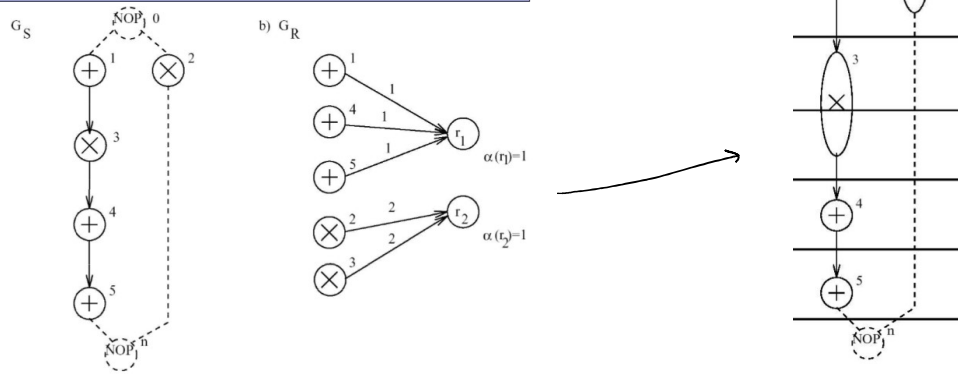
5. List Scheduling

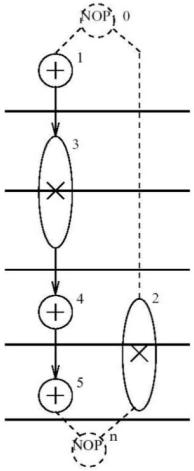
→ List Scheduling is one of the most widely used algorithms for scheduling under resource constraints

- To each operation there is a priority assigned which denotes the urgency of being scheduled

```
LIST(G_S(V_S,E_S),G_R(V_R,E_R),α,β,priorities){
  t = 1;
  REPEAT {
    FORALL v_k ∈ V_T {        (resource types)
      determine candidates to be scheduled U_k;   (v ∈ V_S with β(v) = v_k)
      determine running operations T_k;
      choose S_k ⊆ U_k with maximal priority
        and |S_k| + |T_k| ≤ α(v_k);
      τ(v_i) = t  ∀v_i ∈ S_k;   }
    t = t + 1;
  } UNTIL (v_n planned)
  RETURN (τ);   }
```

U_k = ready operations
T_k = running operations
S_k = starting operations



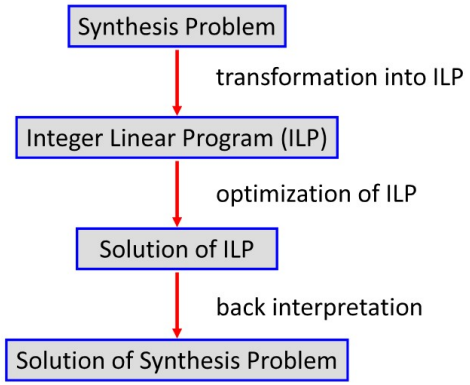


- List scheduling is a **heuristic** algorithm
  ⇒ It does not always yield the minimal latency.
- The optimal solution for the example given would be the schedule shown on the left

6. Integer Linear Programming

→ Yields optimal solution to syntesis problems as it is based on an exact mathematical description of the problem

- Solves scheduling, binding and allocation simultaneously



→ Assumptions

- The binding is determined already, i.e., every operation has a unique execution time $\omega(v_i)$
- We know the earliest and latest starting times of operations v_i as l_i and h_i. L_{max} is chose so that a feasible schedule exists

$$
\begin{aligned}
&\text{minimize:} && \tau(v_n) - \tau(v_0) \\
&\text{subject to} && x_{i,t} \in \{0,1\} \quad \forall v_i \in V_S \;\; \forall t : l_i \le t \le h_i && (1)
\end{aligned}
$$

$$\sum_{t=l_i}^{h_i} x_{i,t} = 1 \quad \forall v_i \in V_S \qquad (2)$$

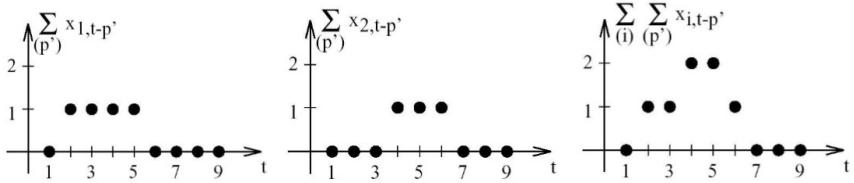
$$\sum_{t=l_i}^{h_i} t \cdot x_{i,t} = \tau(v_i) \quad \forall v_i \in V_S \qquad (3)$$

$$\tau(v_j) - \tau(v_i) \ge w(v_i) \quad \forall (v_i, v_j) \in E_S \qquad (4)$$

$$\sum_{\forall i:(v_i,v_k)\in E_R} \; \sum_{p'=\max\{0,t-h_i\}}^{\min\{w(v_i)-1,\,t-l_i\}} x_{i,t-p'} \le \alpha(v_k)$$
$$\forall v_k \in V_T \;\; \forall t : 1 \le t \le \max\{h_i : v_i \in V_S\} \qquad (5)$$

- (1) declares variables x to be binary .
- (2) makes sure that exactly one variable $x_{i,t}$ for all t has the value 1, all others are 0.
- (3) determines the relation between variables x and starting times of operations τ. In particular, if $x_{i,t} = 1$ then the operation v_i starts at time t, i.e. $\tau(v_i) = t$.
- (4) guarantees, that all precedence constraints are satisfied.
- (5) makes sure, that the resource constraints are not violated. For all resource types $v_k \in V_T$ and for all time instances t it is guaranteed that the number of active operations does not increase the number of available resource instances.
- (5) The first sum selects all operations that are mapped onto resource type v_k. The second sum considers all time instances where operation v_i is occupying resource type v_k :

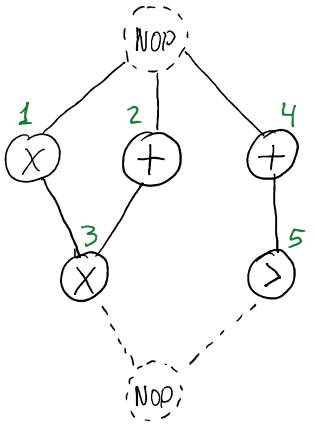
$$\sum_{p'=0}^{w(v_i)-1} x_{i,t-p'} = \begin{cases} 1 & : \quad \forall t : \tau(v_i) \le t \le \tau(v_i) + w(v_i) - 1 \\ 0 & : \quad \text{sonst} \end{cases}$$



Example

Consider the sequence graph $G_s = (V_s, E_s)$

- For the execution times: A multiplication takes 2 time units and all other operations take 1 time unit each.
- There is one unit of each resource type
- Formulate the problem of latency minimization with restricted resources as an integer linear program (ILP)



i. Perform ASAP and ALAP → Consider $L_{max} = 4$

	ASAP(l_i)	ALAP(h_i)
V_1	1	1
V_2	1	2
V_3	3	3
V_4	1	3
V_5	2	4

→ means that V_2 can only run between timestamps $t = 1$ and $t = 2$

$\Rightarrow 1 \leq \mathcal{T}(V_2) \leq 2$

ii. Formulate ILP

(1) Objective function

$$\min \{ L = \mathcal{T}(V_n) - \mathcal{T}(V_0) \}$$

(2) Binary variables $x_{i,t}$ (node #id, timestamp) $t \in [l_i, h_i]$

* One sum for each node for time $t \in [l_i, h_i]$ (where it can run)

V_1: $x_{1,1} = 1$

V_2: $x_{2,1} + x_{2,2} = 1$

V_3: $x_{3,3} = 1$

V_4: $x_{4,1} + x_{4,2} + x_{4,3} = 1$

V_5: $x_{5,2} + x_{5,3} + x_{5,4} = 1$

(3) Starting time of operations $t \in [l_i, h_i]$

* One sum for each node for time $t \in [l_i, h_i]$ (where it can run)

V_1: $x_{1,1} = \mathcal{T}(V_1)$

V_2: $x_{2,1} + 2x_{2,2} = \mathcal{T}(V_2)$

V_3: $3x_{3,3} = \mathcal{T}(V_3)$

V_4: $x_{4,1} + 2x_{4,2} + 3x_{4,3} = \mathcal{T}(V_4)$

V_5: $2x_{5,2} + 3x_{5,3} + 4x_{5,4} = \mathcal{T}(V_5)$

(4) Time constraints

$\mathcal{T}(V_3) - \mathcal{T}(V_1) \geq 2$

$\mathcal{T}(V_3) - \mathcal{T}(V_2) \geq 1$

$\mathcal{T}(V_5) - \mathcal{T}(V_4) \geq 1$

$\mathcal{T}(V_n) - \mathcal{T}(V_3) \geq 2$

$\mathcal{T}(V_n) - \mathcal{T}(V_5) \geq 1$

$\mathcal{T}(V_1, V_2, V_3, V_4, V_5) \geq 1$

(5) Resource constraints

* One sum for each timestamp $t \in [1, \max\{h_i\}]$ and resource type $v_t \in V_T$
* See which tasks can run at each timestamp and sort by resource type

	r_1 (mult)	r_2 (add, sub, comp)
$t = 1$:	$x_{1,1} \leq 1$	$x_{2,1} + x_{4,1} \leq 1$
$t = 2$:		$x_{2,2} + x_{5,2} \leq 1$
$t = 3$:	$x_{3,3} \leq 1$	$x_{4,3} \leq 1$
$t = 4$:		$x_{5,4} \leq 1$

→ at $t = 1$, only V_2 and V_4 use r_2. Since there is only one r_2 available

$\Rightarrow x_{2,1} + x_{4,1} \leq 1$

7. Iterative Algorithms

→ *Iterative Algorithms* consists of a set of indexed equations that are evaluated for all values of an index l:

$$x_i[l] = F_i[\ldots, x_j[l-d_{ji}], \ldots] \quad \forall l \, \forall i \in I$$

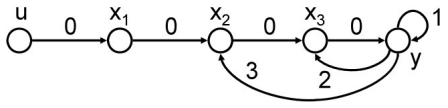
- Representations
  - * One indexed equation with constant index dependencies

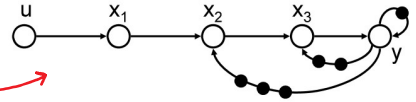
$$y[l] = au[l] + by[l-1] + cy[l-2] + dy[l-3]$$

  - * Equivalent set of indexed equations

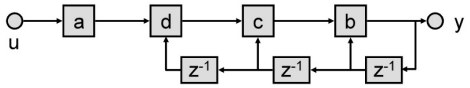
$$\begin{aligned} x_1[l] &= au[l] && \forall l \\ x_2[l] &= x_1[l] + dy[l-3] && \forall l \\ x_3[l] &= x_2[l] + cy[l-2] && \forall l \\ y[l] &= x_3[l] + by[l-1] && \forall l \end{aligned}$$

- Extended sequence graph $G_s = (V_s, E_s, d)$: To each edge $(v_i, v_j) \in E_s$ there is associated the index displacement d_{ij}. An edge $(v_i, v_j) \in E_s$ denotes that the variable corresponding to v_j depends on variable corresponding to v_i with displacement d_{ij}





marked graph



signal flow graph

```
while(true) {
      t1 = read(u);
      t5 = a*t1 + d*t2 + c*t3 + b*t4;
      t2 = t3;
      t3 = t4;
      t4 = t5;
      write(y, t5);}
```

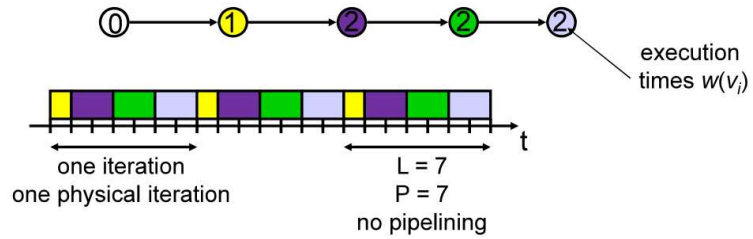
loop program

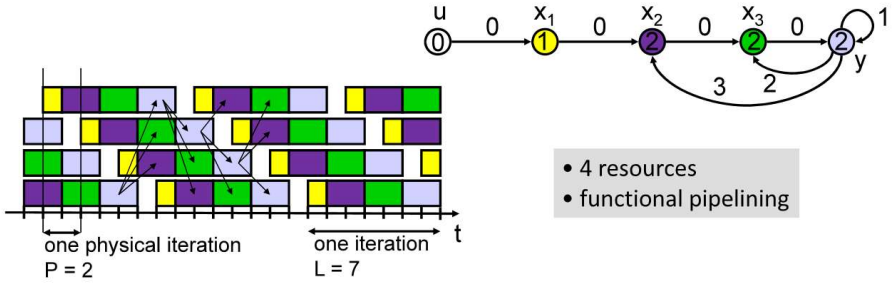
→ An *iteration* is the set of all operations necessary to compute all variables for a fixed index l.

→ The *iteration interval* P is the time distance between two successive iterations of an iterative program. $1/P$ denotes the *throughput* of the implementation

→ The *Latency* L is the maximal time distance between the starting and the finishing times of operations belonging to one iteration

→ In a pipelined implementation (*functional pipelining*), there exist time instances where the operations of different iterations l are executed simultaneously.





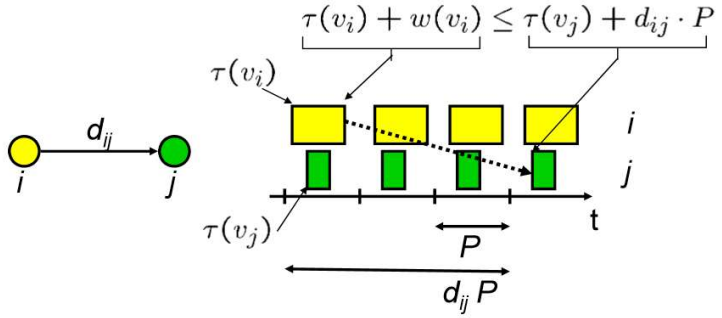
→ Solving the synthesis problem using integer linear programming

- Starting point is the ILP formulation given for simple sequence graphs (6.).
- Now, we use the **extended sequence graph** (including displacements d_{ij})
- ALAP and ASAP scheduling for upper and lower bounds h_i and l_i use only edges with $d_{ij} = 0$ (remove dependencies accross iterations)

- We suppose that a suitable iteration interval P is chosen beforehand. If it is too small, no feasible solution to the ILP exists and P needs to be increased
- Equation (4) is replaced by

$$\tau(v_j) - \tau(v_i) \geq \omega(v_i) - d_{ij} P \quad \forall (v_i, v_j) \in E_s$$

Proof of correctness



- Equation (5) is replaced by

$$\sum_{\forall i:(v_i,v_k)\in E_R} \sum_{p'=0}^{\omega(v_i)-1} \sum_{\forall p: l_i \leq t-p'+pP \leq h_i} x_{i,t-p'+pP} \leq \alpha(v_k) \quad \forall l \leq t \leq P, \forall v_k \in V_T$$

Proof of correctness: An operation v_i starting at $\tau(v_i)$ uses the corresponding resource at time steps t with

$$t = \tau(v_i) + p' - pP$$
$$\forall p', p : 0 \leq p' < \omega(v_i) \wedge l_i \leq t - p' + pP \leq h_i$$
$$\Rightarrow \sum_{p'=0}^{\omega(v_i)-1} \sum_{\forall p: l_i \leq t-p'+pP \leq h_i} x_{i,t-p'+pP}$$