Embedded Systems Summary
Andreas Biri, D-ITET 10.07.15

1. Introduction

Embedded Systems (ES): information processing systems embedded into a larger project

Cyber-physical system (CPS): must operate dependably, safely, securely, efficiently and in real-time

Characteristics of Embedded Systems (1-19)
- dependable: reliable, maintainable, available, safe
- efficient: energy, code, run-time, weight, cost
- specialized: dedicated towards certain application
- real-time: must meet constraints of environment
- not programmable by end-user
- fixed run-time requirements (additional power useless)
- criteria: cost, power consumption, predictability
- energy & temperature constrains (often independent)
- energy harvesting important (e.g. zero power systems)

Hard real-time constrain: not meeting that constrain could result in a catastrophe; answer arriving too late is wrong

Hybrid system: analog and digital system components

Reactive system: in continual interaction with environment executes at pace determined by environment

MPSOcs: Multiprocessor systems-on-a-chip (e.g. phone)

2. Software Introduction

Real-Time Systems (2-15)

ES are expected to finish tasks reliably within time bounds

Hard constrain: missing a deadline results in catastrophe
- often in safety-critical applications (aeronautics, brakes)

Soft constrain: missing deadline is undesirable but not fatal

Worst-Case Execution Time (WCET): upper bound on execution time of all tasks statically known
- difficult to calculate because of parallelism (branch prediction, speculation, pipelines) & caches

Best-Case Execution Time (BCET): lower bound for it

Programming Paradigms (2-25)

Time triggered approaches (2-26)
- periodic
- cyclic executive
- generic time-triggered scheduler
- no interrupts except by timer
- deterministic behaviour at run-time
- interaction with environment through polling

Summary
+ deterministic schedule (computed before run-time)
+ shared resources pose no problem
- external communication only via polling
- inflexible (no adaptation to environment)
- long processes have to be split into subtasks

Extension
- allow arbitrary interrupts (not deterministic anymore!)
- allow preemtable background processes

Simple Periodic Time-Triggered Scheduler

Timer interrupts regularly with period $P$ (same for all processes)

- unpredictable starting times for later processes
- mutually exclusive, no sync required for communication

\[ \sum_t \text{WCET}(T_i) < P \]

Time-Triggered Cyclic Executive Scheduler

processes may have different periods

- period $P$ portioned into frames of length $f$
- terrible for long processes (need to be split)

Conditions
- Process executes at most once within a frame
  \[ f \leq p(k) \quad \forall k \]
- Period $P$ is least common multiple of all periods $p(k)$
- Periods start and complete within a single frame:
  \[ f \geq \text{WCET}(k) \quad \forall k \]
- at least one frame boundary between release & deadline
  \[ 2f - \gcd(p(k), f) \leq D(k) \quad \forall k \]

Generic Time-Triggered Scheduler

- precompute schedule a priori offline (if purely TT)

Task-Descriptor List (TDL): contains cyclic schedule for all activities, considering required precedence and mutual exclusion -> no explicit coordination at run-time necessary
Event triggered approaches (2-36)
- non-preemptive
- preemptive (stack policy, cooperative, multitasking)

Summary
+ dynamic & adaptive
+ can react to environment by receiving interrupts
+ guarantees can be given during run-time or even off-line
- problems with respect to timing
- shared resources have to be coordinated

Non-Preemptive Event-Triggered Scheduling
events are collected in a queue and cannot be preempted
(cannot give guarantees regarding deadlines)

ISR: Interrupt service routine
- event associated with corresponding process
- events emitted by
  a) external interrupts
  b) processes themselves
- simple communication between processes
- buffer overflow if too many events are generated
- long processes prevent others from running (-> split)

Extension
- preemtable background process if event queue is empty
- timed events enter queue only after time interval elapsed

Preemptive Event-Triggered Scheduling
possible to preempt process, solves problem of long tasks

Stack-based: stack-based context mechanism of function calls
(process = C-style function with own memory space)
- LIFO: restricts flexibility, bad if waiting for external event
- no mutual exclusion; shared resources must be protected
  (e.g. disable interrupt, semaphores)

Processes and CPU (2-43)

Process: unique execution of a program (“instance”)
- has its own state (e.g. register values, memory stack)
- several copies of a program can run simultaneously

Activation record: copy of process state (includes registers)

Context switch: current CPU context goes, next comes
- context of current process is stored (registers, program counter, stack pointer)
- execution continues where other process left off

Co-operative Mutitasking (2-45)

process allows context switch at cswitch() call
+ predictable where context switch can occur
+ less errors with use of shared resources
- bad programming can stall the system (doesn’t yield)
- real-time behaviour at risk (if switch not possible)

Preemptive Mutitasking (2-60)

Scheduler (OS)
  i) controls when context switches
  ii) determines which process runs next

Scheduler is called / switch enforced by:
- use of timers / timer interrupts
- hardware or software interrupts
- direct call to OS routines to switch context

3. Real-Time Models

Hard: missing its deadline has catastrophic consequences
Soft: meeting its deadline is desirable, but not critical

Schedule: assignment of tasks to the processor
- \( \sigma(t) = 0 \) : processor is idle at time \( t \)
- \( \sigma(t) = i \) : processor is executing task \( i \) at time \( t \)

Feasible: tasks can be completed according to constraints
Schedulable: there exists at least one algorithm which can produce a feasible schedule

Schedule & Timing (3-5)

\[ J_i \] / \( \tau_i \) task / periodic task \( i \)

\[ a_i \] / \( r_i \) arrival / release time (ready for execution)

\[ C_i \] computation time (required CPU time)

\[ d_i \] / \( D_i \) absolute / relative deadline \( d_i \geq r_i + C_i \)

\[ s_i \] / \( f_i \) start / finishing time

\[ T_i \] period (for periodic tasks)

\[ \Phi_i \] phase (start of periodic task)

Derived figures

Lateness delay of a task completion

Laxity / slack time maximal time a task can be delayed on its activation to complete within deadline

\[ L_i = f_i - d_i \]

\[ E_i = \text{max}(0, L_i) \]

\[ X_i = d_i - a_i - C_i \]
Precedence Constrains: describes the interdependencies between tasks ("Which one has to be executed first?")

Classification of Scheduling Algorithms (3-11)

Preemptive algorithm: running task can be interrupted at any time to assign the processor to another active task

Non-preemptive algorithm: once started, the task is executed until completion (no interruptions)

Static algorithm: scheduling decisions are based on fixed parameters, assigned to tasks before activation (offline)

Dynamic algorithm: scheduling decisions based on dynamic parameters that may change during system execution (e.g. CPU bursts, I/O waits)

Schedule metrics (3-13)

Optimal algorithm: minimizes given cost function

Heuristic algorithm: tends to find optimal schedule

Average response time
\[ t_r = \frac{1}{n} \sum_{i=1}^{n} (f_i - r_i) \]

Total completion time
\[ t_c = \max(f_i) - \min(r_i) \]

Weighted sum of response time
\[ t_w = \sum w_i (f_i - r_i) / \sum w_i \]

Maximum lateness
\[ L_{\text{max}} = \max(f_i - d_i) \]

Number of late tasks
\[ N_{\text{late}} = \sum_{i=1}^{n} \text{miss}(f_i), \quad \text{miss}(f_i) = \begin{cases} 0 & f_i \leq d_i \\ 1 & \text{else} \end{cases} \]

4. Periodic/Aperiodic Tasks

Aperiodic Tasks (4-3)

Equal arrival times & non-preemptive
- EDD (Jackson) for independent tasks
- LDF (Lawler) for dependent tasks

Arbitrary arrival times & preemptive
- EDF (Horn) for independent tasks
- EDF* (Chetto) for dependent tasks

Earliest Deadline Due (EDD) (4-4)
equal arrival times & non-preemptive : \( O(n \log(n)) \)

Algorithm: Task with earliest deadline is processed first

Jackson’s rule: processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness

Earliest Deadline First (EDF) (4-7)
arbitrary arrival times & preemptive : \( O(n^2) \)

Algorithm: Task with earliest deadline is processed first; if new task arrives with earlier deadline, current task is interrupted (just like EDD, but with recalculation)

Horn’s rule: executing the task with the earliest absolute deadline among the ready tasks at any time is optimal with respect to minimizing the maximum lateness

Earliest Deadline First* (EDF*) (4-12)
determines a feasible schedule for tasks with precedence constrains if there exists one

Algorithm: Modify release times & deadlines, then EDF

Modification of release times:
1. Start at the top (roots to leaves)
2. Search the predecessor which takes the longest:
\[ r_j^* = \max(r_j, \max(r_i^* + c_{ij} : I_i \rightarrow J_j)) \]

Modification of deadlines:
1. Start at the bottom (leaves to roots)
2. Search the successor which starts the earliest:
\[ d_j^* = \min(d_j, \min(d_i - c_{ij} : I_i \rightarrow J_j)) \]

Latest Deadline First (LDF) (U2.2)
Non-preemptive scheduling for precedence constrains

Algorithm:
1. A precedence graph is constructed
2. Leaves to roots: Select task with latest deadline among all available tasks to be scheduled last
3. At runtime: tasks are extracted from head of the queue: first task inserted into queue will be executed last (FILO)

Shortest Job First (SJF)
Minimizes average waiting time
**Periodic Tasks (4-17)**

**Deadline equals period:**
- Rate-monotonic (RM) for static priority
- EDF for dynamic priority

**Deadline smaller than period:**
- Deadline-monotonic (DM) for static priority
- EDF* for dynamic priority

---

**Terminology**

- \( \tau_{i,j} \) denotes the \( j \)-th instance of task \( i \)
- \( r_{i,j} \) / \( s_{i,j} \) / \( f_{i,j} \) release / start / finishing time
- \( \Phi_i \) phase of task \( i \) (release time of its first instance)
- \( D_i \) relative deadline of task \( i \) (same for all instances)
- \( T_i \) period with which the task is regularly activated
- \( C_i \) worst case execution time (same for all instances)

**Rate Monotonic Scheduling (RM) (4-22)**

RM is optimal among all fixed-priority assignments, i.e. no other fixed-priority algorithm can schedule a task set which cannot be scheduled with RM
- static priority assignment (offline, as not changed)
- preemptive (by a task with higher priority)
- deadlines equals to the period \( (C_i = D_i = T_i) \)

**Algorithm:**

- tasks with higher request rate / shorter period will have higher priorities and interrupt lower ones

**Critical Instant:**

- task is release simultaneously with all higher priority tasks / release creates largest response time

**Schedulability analysis**

- Sufficient but not necessary: \( \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n \left( 2^{1/n} - 1 \right) \)

**Earliest Deadline First (EDF) Scheduling (4-41)**

**Algorithm:**

- Active task with earliest deadline has highest priority
- dynamic priority assignment
- preemptive
- \( D_i \leq T_i \)

**Schedulability test only for \( D_i = T_i \):**

- Necessary & sufficient: schedulable with EDF if and only if
  \[ \sum_{i=1}^{n} \frac{C_i}{T_i} = U \leq 1 \]

**Problem of Mixed Tasks Sets (4-47)**

**Periodic tasks:**

- time-driven, execute regular critical control activities with hard timing constrains

**Aperiodic tasks:**

- event-driven; hard, soft or no real-time

**Sporadic tasks:**

- aperiodic task characterized by a minimum interarrival time (enables offline guarantee on constrains)

---

**Background scheduling (4-48)**

RM & EDF scheduling of periodic tasks: processing of aperiodic tasks in the background / when no periodic one

---

**RM Polling Server (PS) (4-50)**

**Idea:**

- Introduce artificial periodic task which services aperiodic requests as soon as possible

**Function of polling server (PS):**

- instantiated at regular intervals \( T_S \)
- serves any pending aperiodic requests
  - If none, the process is suspended (time not preserved!)

**Disadvantage:**

- if an aperiodic request arrives just after the server is suspended, it must wait for next polling period

**Schedulability analysis:**

- just like RM, suff. but not necessary

- \( \frac{C_S}{T_S} + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq (n+1) \left( 2^{1/(n+1)} - 1 \right) \)

- Sufficient if aperiodic task finishes before a new arrives

\[ \left( 1 + \left[ \frac{C_a}{C_S} \right] \right) T_S \leq D_a \]
EDF – Total Bandwidth Server (4-55)

When k-th aperiodic request arrives at time \( t = r_k \), it receives a deadline

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

\( U_s = 1 - U_p \) : server utilization factor / bandwidth

Once a deadline is assigned, the request is inserted into the ready queue as any other periodic instance

Schedulability test: necessary & sufficient

\[ U_p + U_s \leq 1 \]

5. Resource Sharing

Common resources: data structures, variables, main memory area, file, set of registers, I/O unit

Critical section: piece of code, in which access to shared resources requires mutual exclusion

blocked: task waits for an exclusive resource to be freed

holds: task is in possession of said resource

free: exclusive resource after leaving critical section

Semaphores (5-5)

\( S_i \) protects each exclusive resource \( R_i \)

\( \text{wait}(S_i) \): start of critical section, requests entrance

\( \text{signal}(S_i) \): end of critical section, frees resource

Priority Inversion (5-7)

- low-priority task holds resource which prevents high-priority task from running
- meanwhile, a medium-priority task can preempt the low-priority task and execute with the high-priority blocked

“Solution”: disallow preemption in critical sections

- unnecessary blocking of unrelated tasks with higher priority

Priority Inheritance Protocol (PIP) (5-10)

assume priority of highest blocked task in critical section

\( P_i \) : nominal priority

\( p_i \geq P_i \) : active priority

Direct Blocking: lower-priority task blocks higher task

Push-through Blocking: medium-priority task is blocked by low-priority task which has inherited a higher priority

6. Real-Time OS

Deficits of Desktop OS

- monolithic kernel too feature rich, takes too much space
- not modular, fault-tolerant, configurable, modifiable
- not power optimized
- timing uncertainty too large

Advantages of Embedded OS

- OS can be fitted to each individual need: remove unused functions, conditions compilation depending on hardware, replace dynamic data by static data, advanced compiling
- improved predictability (everything through scheduler)
- interrupts can be employed by all processes
- software tested and considered reliable (no protection)

Real-Time OS (6-10)

Requirements

- predictability of time-behaviour
  - upper bound on the execution time of tasks
  - almost all activities controlled by scheduler

- management of timing and scheduling
  - inclusion of deadlines
  - OS must provide precise time services
  - speed

Main functionality of RTOS-Kernels (6-13)

Process management (6-13)

- execution of quasi-parallel tasks
  - maintain process states & process queues
  - preemptive scheduling (fast context switch)
  - quick interrupt handling

- CPU scheduling: guarantee deadlines & fairness

- Process synchronization (semaphores, mutual exclusion)

- Inter-process communication (buffering)

- real-time clock for internal time reference
Process States (6-15)

run: starts executing on the processor
ready: ready to execute but not assigned yet
wait: task is waiting for a semaphore for access
idle: completed execution & waiting for next period

Threads (6-17)
A basic unit of CPU utilization, similar to a process
- typically shared: memory
- typically owned: registers, stack

Process: difficult to communicate, think they are alone
Thread: communicate via memory, knows there are others
- multiple threads for each distinct activity of process
- faster to switch between threads (no major OS operation)
- Thread Control Block (TCB) stores information

Communication Mechanisms (6-20)
Problem: the use of shared memory for message passing may cause priority inversion and blocking

Synchronous communication (“rendez-vous”)
- when communicating, they have to wait for each other
- causes problems for maximum blocking time
- in static RT environments solved offline by transforming synchronous interactions into precedence constrains

Asynchronous communication (“mailbox”)
- sender deposits message into channel, receiver reads
- done by shared memory buffer, FIFO queue (fixed size)

7. System Components

General-purpose Processors (7-7)
- high performance
  - highly optimized circuits and technology
  - use of parallelism (pipelining, predictions)
  - complex memory hierarchy
- not suited for real-time applications as highly unpredictable execution times due to intensive resource sharing and dynamic decisions
- good average performance for large application mix
- high power consumption

Multicore Processors
- higher execution performance through parallelism
- useful in high-performance embedded systems
- interference on shared resources (buses, cache etc.)

Communication Mechanisms (6-20)

Microcontrollers / Control Dominated Systems (7-22)
- Reactive systems with event driven behavior
- system description: Finite State Machines or Petri Nets

Microcontrollers connect interfaces (no computation)
- support process scheduling and synchronization
- preemption (interrupt), context switch
- short latency times
- low power consumption
- peripheral units often integrated (timer, buses, AD/DA-C)
- suited for real-time applications

Digital Signal Processors (DSPs) / Data Dominated Systems (7-26)
- Streaming-oriented systems with periodic behaviour
- input description: flow graphs

DSPs are for computation (signal processing, controlling)
- optimized for data-flow, only simple control-flow
- parallel hardware units (VLIW), specialized instruction set
- high data throughput, zero-overhead loops
- suited for real-time applications

Very Long Instruction Word (VLIW): detection of possible parallelism by compiler, combine multiple functional units

Field Programmable Gate Array (FPGA) (7-34)
- “program hardware by software”
- granularity of logic units: gate, tables, memory, blocks
- communication network: crossbar, hierarchical mesh
- reconfiguration: dynamically adjustable at runtime

Application-Specific Circuits (ASICS) (7-41)
- custom-designed circuits for mass production
- long design times, lack of flexibility, high design costs

System-on-Chip (SoC) (7-43)
8. Communication

Requirements
- performance (bandwidth & latency, real-time)
- efficiency (cost, low power)
- robustness (fault tolerance, maintainability, safety)

Time Multiplex Communication (8-5)

Random Access (8-6)
No access control, requires low medium utilization
Improved variant: slotted random access

TDMA (Time Division Multiple Access) (8-7)
Communication in statically allocated time slots
- synchronization among all nodes necessary
- master node sends out a synchronization frame

CSMA/CD (Carrier Sense MA / Collision Detection) (8-8)
Try to avoid and detect collisions
- before transmitting, check whether channel is idle
- if collision detected, back off / wait
- repeated collisions result in increasing backoff times

Token Protocol (Token Ring) (8-9)
Token value determines which node is transmitting
- only the token holder may transmit

CSMA/ Collision Avoidance – Flexible TDMA (FTDMA) (8-11)
Reserve $s$ slots for $n$ nodes; if slot is used, it becomes slice
- node start transmitting message only during assigned slot
- $s = n$: no collision; $s \leq n$: statistical collision avoidance

CSMA/ Collision Resolution (CSMA/CR) (8-12)
Each node is assigned a unique identification number
- all nodes wishing to transmit send a binary signal based on their identification number; if node detects a dominant state while transmitting a passive one, it drops out
- node with the lowest identification value wins

Flex Ray (8-14)
Operation principle: Cycle is subdivided into static and dynamic segment. Static segment bases on fixed allocation of time slots, dynamic segment for ad-hoc communication

Flex Ray (8-14)
Operation principle: Cycle is subdivided into static and dynamic segment. Static segment bases on fixed allocation of time slots, dynamic segment for ad-hoc communication

Flex Ray (8-14)
Operation principle: Cycle is subdivided into static and dynamic segment. Static segment bases on fixed allocation of time slots, dynamic segment for ad-hoc communication

Network Topologies (8-24)
Ad-hoc networks
- all nodes are potentially mobile
- dynamic emergence of connections
- hierarchical structure (scatternet) of small nets (piconet)

Piconet
- contains 1 master and maximally 7 slaves
- all nodes inside use the same frequency hopping scheme
- (determined by device address of master BD_ADDR)
- connections exist:
  - one-to-one
  - master and all slaves (broadcast)

Scatternet
- formed by several piconets with overlapping nodes
- node can be master in at most one and slave in other nets

Addressing (8-30)
Packet format
- Access Code / BD_ADDR : 82bits, identifies packets
- Header / AM_ADDR : 54bits, identifies connection
- Payload : 0 – 2745 bits

Bluetooth Device Address BD_ADDR : 48 Bits, unique

Active Member Address AM_ADDR :
- 3 bits for maximally 7 active slaves in piconet
- Address “Null” is broadcast to all slaves

Parked Member Address PM_ADDR : 8 bits
- in low power state: waiting for communication

Bluetooth (Frequency Multiplex Communication) (8-20)
Design goals
- small size, low cost, low energy
- secure & robust transmission (interference with WLAN)

Technical Data
- 2.4 GHz (spectral bandwidth 79 MHz)
- 10-100m transmission range, 1 Mbit/s bandwidth
- simultaneous transmission of multimedia & data
- ad hoc network (de-centralized, dynamic connections)
Connection Types (8-31)

Synchronous Connection-Oriented (SCO)
- point-to-point full duplex between master & slaves
- master reserves slots for transmission regularly

Asynchronous Connection-Less (ACL)
- asynchronous service, no slot reservation
- master transmits spontaneous, slaves answer next

Frequency Hopping / Time Multiplexing (8-32)
- packet of the master is followed by a slave packet
- after each packet, channel / frequency is switched
- master can only start sending in even slot numbers
- packets have length of 1, 3 or 5 slots (same frequency)

Modes and States (8-35)

Modes of operation

Inquiry: master identifies addresses of neighbors
Page: master attempts connection with slave
Connected: connection is established

States in connection mode
- active: active in connection to master
- hold: does not process data packets
- sniff: awakens at regular intervals, checks whether there are packets
- park: passive, only synchronized

Synchronization in Connection Mode: channel sequence & phase of a piconet is determined (by BD_ADDR) of master

Synchronization in Page Mode: 3-way-handshake to synchronize between master and slave; prerequisite for establishing a connection
1. Page: master transmits own & slave address
2. Page scan: slave listens
3. Master page response: slave answers with own address
4. Slave page response: master sends FHS-packet, which includes channel sequence & phase of piconet

From Standby to Connection (8-40)

Protocol Hierarchy (8-44)

Baseband specification: defines packet formats, physical & logical channels, error correction, synchronization and modes of operations
Audio specification: defines coding & decoding
Link manager (LM): authentication & encryption, management, connection initiation, transitions
Host controller interface (HCI): interface host - node
Link layer control & adaptation layer (L2CAP): interface for data communication
RFCOMM: simple transport protocol for serial connection

9. Low Power Design

Power is most important constrain in Embedded Systems

Power and Energy (9-9)

\[ E = \int P(t) \, dt \]

Minimizing power consumption is important for
- design of the power supply & voltage regulators
- the dimensioning of interconnect
- cooling (decrease temporary heating)

Minimizing energy consumption is important due to
- restricted availability of energy (mobile systems)
- limited battery capacities & long lifetimes needed
- very high costs of energy (solar panels, in space)

Power Consumption of CMOS Processor (9-12)

Dynamic power consumption: charging & discharging \( C_L \)
Short circuit power consumption: switching causes shorts
Leakage: leaking diodes & translators, causes static current

Power \( P \approx C_L \frac{V_{dd}^2}{f} \)
Energy \( E \approx C_L V_{dd}^2 \left( \frac{1}{f} + \frac{1}{f_T} \right) \)
Delay \( \tau \approx C_L \frac{V_{dd}^2}{(V_{dd} - V_T)^2} \)

\( V_{dd} \): supply voltage
\( V_T \ll V_{dd} \): threshold voltage
\( \alpha \): switching activity (1 : switch every cycle)
\( C_L \): load capacity
\( f \approx \frac{1}{\tau} \approx \frac{V_{dd}}{C_L} \): clock frequency
Basic Techniques (9-17)

Power Supply Gating: minimize static power consumption (leakage) by cutting off power supply while unit inactive

Parallelism (9-18)

Parallelism

Pipelining

\[ E \sim V_{zd}^2 \text{ (#cycles)} \]
\[ E_2 = \frac{1}{4} E_1 \]

VLIW Architectures (9-22)

Large degree of parallelism & many computational units:
- explicit parallelism (parallel instruction set) by compiler
- parallelization through hardware (difficult & expensive)

Translation of instruction set
- done with optimized compiler (no compatibility)
- on processor with decoder (translation in HW)
- on processor with dynamic compiler in SW (Transmeta)

Dynamic Voltage Scaling (DVS) (9-26)

Adapt voltage & frequency to situation to save energy

Optimal Strategy: running at a constant frequency/voltage minimizes energy consumption for dynamic voltage scaling
- if a task finishes on deadline, the chosen frequency (voltage) is optimal in terms of energy efficiency
- if only discrete voltage levels, choose directly above and below the ideal voltage to minimize energy consumption

YDS Algorithm for Offline Scheduling (9-36)

Schedule without missing deadlines & minimal energy \( O(N^3) \), \( N: \) number of tasks in \( V \)

Intensity \( G \) in time interval \([ z, z' ]\): average accumulated execution time of all tasks inside the interval

\[
V'(z, z') = \{ v_i \in V : z \leq a_i < d_i \leq z' \}
\]

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

1. Find critical interval (i.e. interval with highest intensity) and schedule tasks inside with EDF

\[
C_{eff} = \frac{C_{tot}}{G}, \quad f = G * f_{nominal}
\]

2. Adjust arrival times and deadlines by excluding interval

3. Run algorithm for revised input and put pieces together

Online algorithm: run algorithm with known tasks, if new ones arrive, update schedule; maximally uses 27 times the minimal energy consumption of optimal offline solution

Dynamic Power Management (DPM) (9-46)

DPM tries to assign optimal power saving states

RUN: operational

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

SLEEP: shutdown of on-chip activity

DVS Critical frequency (voltage): running at any frequency (voltage) below is not worthwhile for execution

Procrastination Schedule: execute only voltages higher or equal to the critical voltage (round up lower ones)
- procrastinate task execution & sleep as long as possible

Dependence Graph (DG) (10-4)

directed graph \( G = (V, E), \ E \subseteq V \times V \)

\( (v_1, v_2) \in E \):
- \( v_1 \): (immediate) predecessor of \( v_2 \)
- \( v_2 \): (immediate) successor of \( v_1 \)
- nodes represent tasks, edges represent relations
- describes order relations for execution of single tasks
- represents parallelism, not branches in control flow

Control-Data Flow Graph (CDFG) (10-8)

Description of control structures & data dependencies
- combines control flow & dependence representation

Control Flow Graph: finite state machine which represents the sequential control flow of the program (i.e. branches)
- operations within state are written as dependence graph

Dependence Graph/ Data Flow Graph (DFG):
- NOP operations represent start and end point (polar)
11. Architecture Synthesis

Determine a hardware architecture that efficiently executes a given algorithm

- allocation (determine necessary hardware)
- scheduling (determine timing of operations)
- binding (determine relations between parts)

Models (11-5)

Sequence Graph \( G_S = (V_S, E_S) \) : \( V_S \) denotes operations of the algorithm, \( E_S \) the dependence relations

Resource Graph \( G_R = (V_R, E_R) \) , \( V_R = V_S \cup V_T \) ; \( V_T \) : resource types of architecture, \( G_R \) bipartite graph

Cost function \( c : V_T \rightarrow Z \)

Execution times \( w : E_R \rightarrow Z \)

Allocation \( \alpha : V_T \rightarrow Z \)

Binding \( \beta(v_i) = v_t \) , \( \gamma(v_s) = r \) operation \( v_s \) is implemented on \( r \)-th instance of resource \( v_t \)

Scheduling \( \tau : V_S \rightarrow Z \) determines starting times

feasible if \( \tau(v_j) - \tau(v_i) \geq w(v_i) \ \forall \ (v_i, v_j) \in E_S \)

Latency \( \tau(v_o) - \tau(v_i) \)

Multiobjective Optimization (11-13)

Mostly optimize for more than one objective:
- latency of implemented algorithm
- hardware cost (memory, communication, ALUs)
- power & energy consumption

Pareto Optimum
- Improving a given configuration without downgrading any other aspect is called a pareto improvement
- if no further improvements can be made, the configuration is called pareto optimal (nothing better in all aspects → dominates weaker configurations)

Classification of Scheduling Algorithms (11-19)

- unlimited resources \( \leftrightarrow \) limited resources
- iterative algorithm: initial solution improved step-by-step
- constructive algorithm: problem solved in one step
- transformative algorithm: initial problem formulation is transformed into a (classical) optimization problem

Scheduling without resource constrains (11-20)

Every operation gets its own resource; often used as a first step to determined upper bounds on feasible schedules

As Soon As Possible (ASAP) (11-22)
Start at top, schedule task after all predecessors finished

\[ \tau(v_i) = \max\{ \tau(v_j) + w(v_j) \} , (v_j, v_i) \in E_S \]

As Late As Possible (ALAP) (11-25)
Start at bottom, schedule task before earliest successor

\[ \tau(v_i) = \min\{ \tau(v_j) \forall (v_i, v_j) \in E_S \} - w(v_i) \]

Scheduling with Timing Constrains (11-28)

Constraints:
- deadline : latest finishing time
- release time : earliest starting time
- relative constrains : differences

Weighted Constrain Graph: \( G_C = (V_C, E_C, d) \)
Contains a weighted edge for each timing constrain

An edge \( (v_i, v_j) \in E_C \) with weight \( d(v_i, v_j) \) denotes:

\[ \tau(v_j) - \tau(v_i) \geq d(v_i, v_j) \]

Bellman-Ford-Algorithm: complexity \( O(|V_C| \cdot |E_C|) \)

Iteratively set for all \( v_i \in V_C : \)

\[ \tau(v_i) := \max\{ \tau(v_j) , d(v_i, v_j) \} , (v_j, v_i) \in E_C \]

Starting from \( \tau(v_i) = -\infty , v_i \in V_C \setminus \{v_0\} , \tau(v_0) = 1 \)
Scheduling with resource constraints (11-34)

Minimal latency is defined as
\[ L = \min \{ \tau(v_i) : (\tau(v_i) - \tau(v_j) \geq w(v_i, v_j), \forall (v_i, v_j) \in E_S) \land \{ (v_i : \beta(v_i) = v_t \land \tau(v_i) \leq t < \tau(v_j) + w(v_j, v_i)) \} \leq \alpha(v_k) \forall v_k \in V_f, \forall 1 \leq t \leq L_{\text{max}} \} \]

List Scheduling (11-36)
- static priority, which denotes urgency of being scheduled (e.g. higher priority, the further still away from end)
- algorithm schedules one time after the other and chooses from the tasks with top-priority
- heuristic algorithm, doesn’t guarantee optimal scheduling

\[ \text{LIST}(G_S(V_S, E_S), G_R(V_R, E_R), \alpha, \beta, \text{priorities}) \}
\]
\[ t = 1; \]
\[ \text{REPEAT} \{ \]
\[ \text{FOR ALL } v_k \in V_T \{ \]
\[ \text{determine candidates to be scheduled } U_k; \]
\[ \text{determine running operations } T_k; \]
\[ \text{choose } S_k \subseteq U_k \text{ with maximal priority} \]
\[ \text{and } |S_k| + |T_k| \leq \alpha(v_k); \]
\[ \tau(v_i) = t \quad \forall v_i \in S_k; \}
\[ t = t + 1; \}
\[ \text{UNTIL } (v_n \text{ planned}) \]
\[ \text{RETURN } (\tau); \}

Integer Linear Programming (11-42)
- yields optimal solution, as based on exact description
- binding already determined (know duration)
- know earliest & latest starting times from ASAP / ALAP

1. Minimize: \[ \tau(v_i) - \tau(v_j) = L \]
2. Decision variables \( x \) binary:
\[ x_{ij} \in \{0,1\}, \forall v_i \in V_f, \forall : i \leq t \leq h_i \]
3. Exactly one variable \( x_{ij} \) for all \( t \) has the value 1:
\[ \sum_{h_i} h_i x_{ij} = 1, \forall v_i \in V_f \]
4. Each task can only have one starting point.
5. If \( x_{ij} = 1 \) then the operation \( v_i \) starts at time \( t \), i.e. \( \tau(v_i) = t \).
6. Resource constraints are not violated:
\[ \sum_{i \in v_i / v_j \in E_k \land |S_k| + |T_k| \leq \alpha(v_k)} \sum_{j \in v_j / v_i \in V_T, \forall 1 \leq t = \max \{ h_i : v_i \in V_f \} } \]

Iterative Algorithms (11-49)
Consist of a set of indexed equations that are evaluated for all values of an index variable (e.g. signal flow graphs, marked graphs)

**Representation of iterative algorithms**
- one indexed equation with constant index dependencies
- equivalent set of indexed equations
- extended sequence graph denoting the displacements
- marked graph denoting displacement as data in queue
- signal flow graph (with displacement \( x^{-1} \))
- loop program

Dynamic Voltage Scaling (DVS) (11-60)
We can optimize the energy in case of DVS
- there are \(|K|\) different voltage levels
- task \( v_i \in V_f \) can use one of the execution times \( w_k(v_i) \)
and corresponding energy \( e_k(v_i) \)

1. Minimize:
\[ \sum_{i \in V_f} \sum_{k \in K} w_k(v_i) e_k(v_i) \]
Summed up all individual energies of operations.
2. Decision variables \( x_{ik} \) binary:
\[ x_{ik} \in \{0,1\}, \forall v_i \in V_f, k \in K \]
3. Exactly one implementation (voltage) \( k \in K \) is chosen for each operation \( v_i \):
\[ \sum_{k \in K} x_{ik} = 1, \forall v_i \in V_f \]
4. Precedence constraints, where the actual execution time is selected from the set of all available ones:
\[ \tau(v_j) - \tau(v_i) \geq \sum_{k \in K} x_{ik} + s_k(v_i), \forall (v_i, v_j) \in E_f \]
5. Guarantee deadlines:
\[ \tau(v_i) + \sum_{k \in K} x_{ik} s_k(v_i) \leq d(v_i), \forall v_i \in V_f \]
12. Various

**Petri Nets (2-47)**

- bipartite graph consisting of places and transitions
- data and control represented by moving tokens

**Firing:** enabled if at least one token in every input place
Remove one from each input and put one to each output

**NutOS & Programming Practice (2-50)**

**Creating a thread**

```c
THREAD(my_thread, arg) {
    for (; ;) {
        // do something
    }
}
```

```c
int main(void) {
    if (0 == ButThreadCreate("My Thread", my_thread, 0, 192)) {
        // Creating the thread failed
    }
    for (; ;) {
        // do something
    }
}
```

**Sleep**

```c
THREAD(my_thread, arg) {
    for (; ;) {
        // do something
        ButSleep(1000);
    }
}
```

**Posting & waiting for events (2-57)**

```c
#include <sys/event.h>

HANDLE my_event;

THREAD(thread_A, arg) {
    for (; ;) {
        // some code
        ButEventWait(my_event, NOT_WAIT_INFINITE);
        // some code
    }
}

THREAD(thread_B, arg) {
    for (; ;) {
        // some code
        ButEventPost(my_event);
        // some code
    }
}
```