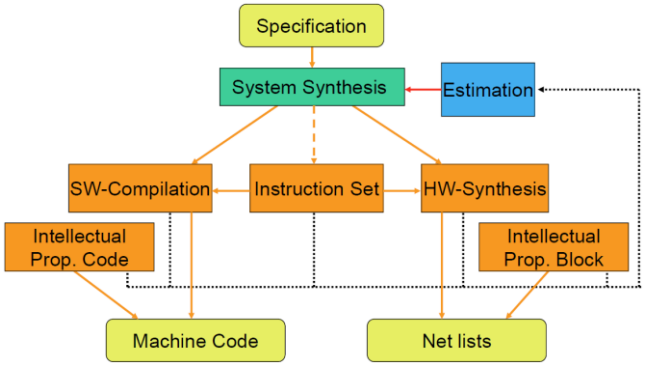
Hardware/Software Codesign

Summary Andreas Biri, D-ITET 27.01.18

1. Introduction

Embedded Systems are designed for specialized processes
- systems exist of dedicated, specialized hardware
- require design optimizations targeted at intended usage (performance, cost, power consumption, reliability)



Embedded systems (ES): information processing systems embedded into a larger product
- deliver enhanced functionality of existing system
- work in parallel & distributed target platforms
- require assured reliability, guarantees & safety (predictability & bounded execution times are critical)
- usage known at design-time, not programmable by user
- fixed run-time requirements (at lowest possible cost)

Often, such systems require real-time processing while still offering low power consumption for energy independence

Multiprocessor system-on-a-chip (MPSoC)
dedicated system, highly specialized with dedicated HW
- application characterized by variety of tasks

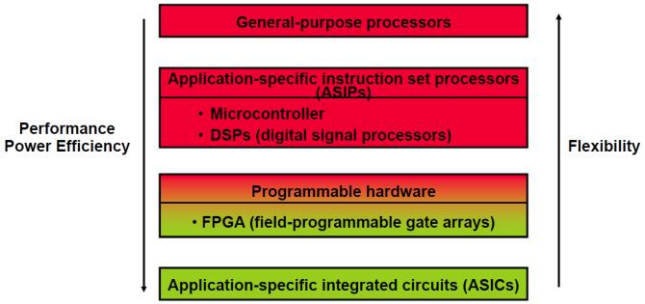
General Purpose Computer: broad class of applications
- programmable by end user, usage might vary over time

Design Challenges

Application complexity: want adaptability, but specialized
- large systems must often provide legacy compatibility
- mixture of event driven & data flow tasks

Target system complexity: design space is very large
- different technologies, processor types, designs
- use finished system-on-chips, distributed implementation

Constraints & design objectives
- cost, power consumption, timing constraints, size, predictability, processing power, temperature



Levels of Abstraction

Specification: "Computer requires FPGA, DSP, …"
- Model formally describes selected system properties
- Consists of data and associated methods

Synthesis: step from abstraction to real system
- connects levels of abstraction (*refinement*)
- from problem-level description to implementation

Modelling: connects implementation to problem-level
- estimation of lower layer properties to improve design

Structural view: abstract layout of components

Behavioural view: describes function of component

Physical view: effective hardware as seen on chip

Hardware/Software Mapping: partitioning of system to programmable components (software) & specialized HW
- can adapt implementation HW to match problem set

2. Specification & Models of Computation

Observer: subject changes state, observer is notified
- one-to-many dependency between subject & observers

Synchronized: causes processes to run sequentially
- solves race conditions, bad performance as not parallel
- can easily cause **deadlocks** if circular dependence

Models of Computation

Very specific systems for one description of model
- restricted language, restricted rules
- offer more possibilities to optimize (more pre-knowledge)
- *ease-of-use* & better analysis (can verify correctness)
- efficient usage, *high abstraction level*

Model of Computation: "What happens inside & how interact?"
- **Components** & execution model for computations
- **Communication** model for information exchange

Discrete Event model: associate even with (trigger) time
- search for next even to simulate, independent of realtime
- allows efficient simulation, as only compute actions
- *VHDL* (hardware description language): sensitivity lists

Finite state machines (FSM): abstract proc. representation
Differential equation: describe component mathematically

Shared memory: potential race conditions & deadlocks
- *Critical section:* must receive exclusive access to resource

Asynchronous message passing: *non-blocking*
- sender does not have to wait until delivered
- potential buffer overflow if receiver doesn't read enough

Synchronous message passing: *blocking*
- requires simultaneous actions by both end components
- automatically synchronizes devices
- *Communication sequential process (CSP):* rendez-vous based communication, bot indicate when ready for it

Specification requirements

Hierarchy: dependencies between components
- *Behavioural hierarchy:* states, processes, procedures
- *Structural hierarchy:* processors, racks, circuit boards

Timing behaviour: mostly requires to be tightly bound

State-oriented behaviour: required for reactive system

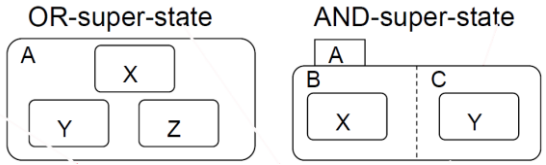
Dataflow-oriented behaviour: parts send data streams

Classical automata: input changes state & output (FSM)
- complex graphs are difficult to read & analyse by humans
- *Moore:* output only depending on state, not input
- *Mealy:* output depends on state and input

2.1 StateCharts (2-25)

Introduces hierarchies to increase readability:
- *Super-state:* can have internal substates
- *Basic state:* its super-state is called *ancestor state*

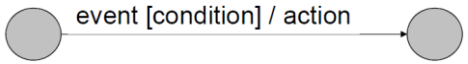
OR-super-state: can be in exactly one of the substates
AND-super-state: is in all of the immediate sub-states



Computation of state sets: traverse tree representation
- OR-super-states: addition of sub-states
- AND-super-states: multiplication of sub-states

Timers: indicated by special edges, traverse after timeout

Besides states, can use **variables** to keep information:
- *action:* variable changes as result of state transition (multiple actions executed simultaneously: $(a := b; b := a)$
- *condition:* dependences of state transitions



Simulation phase

All edge labels are evaluated in 3 different *phases*:

1. Effect of external changes & conditions evaluated
2. Set of transitions to be made is computed
3. Transitions become effective (simultaneously)

Can also consider internal events instantaneous
- external events only considered in *stable state*, i.e. when no more internal steps are conducted & status remains
- *state diagram* only represents stable states

StateChart solves some, but not all problems:
- hierarchy allows nesting of OR & AND states
- can auto-generate C code (but often inefficient)
- no object-orientation & structural hierarchy
- not useful for distributed applications (as have asynchronous events & executions)

Specification & Description Language (SDL): unambiguous specification of reactive & distributed systems
- allows asynchronous message passing, but simultaneous
- can still have undeterminism if race conditions occur (e.g. if we have multiple inputs for the same FIFO queue)

2.2 Data-Flow Models (2-55)

Try to make result independent of time, focus on data
- *processes* communicate through *FIFO buffers*
- one buffer per connection to avoid time dependence

All processes run simultaneously with imperative code
- processes can only communicate through buffers
- maps easily to parallel hardware / *block-diagram specs*

Kahn Process Network
- *read:* destructive & **blocking** (empty queue → busy-wait)
- *write:* non-blocking
- FIFO queues are of infinite size
- **determinate:** no random variables, always know which queue I will read next as blocking (cannot peak & decide)

Random: information known about system & input is not sufficient to determine its outputs (undeterministic)
Determinate: histories of channels depends only on input
- independent of timing, state, hardware; only function

Kahn process: monotonic mapping of input to output
- create output solely based on previous input

Adding non-determinacy can occur in multiple ways:
- allow processes to test for emptiness of input channels
- allow shared channels (read or write)
- allow shared data between processes (variables)

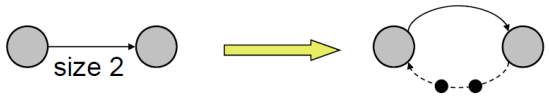
Scheduling Kahn networks

Responsibility of the system, not the programmer
- bounded memory (buffers) can overflow if not managed

Tom Parks algorithm: iteratively increase buffer sizes

1. Start with network with blocking writes
2. Use scheduling which doesn't stall if not all block
3. Run as long as no deadlocks occur
4. When deadlock, increase size of smallest buffer

Finite buffer sizes: limit buffers by introducing reverse
- if want to send, always need to first read from reverse
- blocks if already wrote too much, will wait for token



Kahn Process networks offer various advantages:
- scheduling algorithm does not affect functional behaviour
- matches stream-based processing & explicit parallelism
- easy mapping to distributed & multi-processor platforms
- fuzzy & difficult to implement with balanced rates

Synchronous Dataflow (SDF): allow compile-time schedule
- process reads/writes *fixed number of token* each time
- uses relative execution rates by solving linear equations
- requires rank $n - 1$ for n processes & sufficient init token
- period found when same number of initial tokens again

3. Mapping Application-Architecture

Allocation: select components

Binding: assign functions to components (run what where)

Scheduling: determine execution order (scheduling)

Partitioning: allocation & binding
Mapping: binding & scheduling

Synthesis: implementation of the given specifications
- uses an underlying model of computation

Data Flow graph (DFG): show operation & communication
- *nodes:* operations; *edges:* communication / data
- initial, abstract representation

Control Flow graph (CFG): node ≙ line of code
- includes loops and conditional statements

Architecture Specification: reflects structure & properties
- can be done at different abstraction levels

Mapping relates application & architecture specifications:
- binds processes to processors
- binds communication between processes to busses/paths
- specifies resource sharing disciplines & scheduling

DFG Application model (3-11): initial specification
- *functional* nodes V_P^f : tasks, procedure
- *communication* nodes V_P^c : data dependencies

Architecture model (3-12): describe physical hardware
- *functional* resources V_A^f : processor, RISC, DSP
- *bus* resources V_A^c : shared bus, PTP bus, fiber

Specification graph: maps *application* graph to *architecture*
- action of binding abstract functions to actual hardware
- must occur after allocation
- each data flow node must have an outgoing edge
- communication must actually connect correct HW

4. System Partitioning

Assign tasks to computing resources (& com to networks)

Optimal partitioning can be achieved using various ways:
- compare design alternatives (*design space exploration*)
- estimate with analysis, simulation, prototyping
  (if system parameters are unknown / not determinable)

Usually, one has conflicting design goals & constraints
- *costs:* cost of allocated components (should be minimal)
- *latency:* due to scheduling / resource sharing (parallelize)
- constraints give maximal parameters, try to find solution

Cost functions: *quantitative performance measurement*
- system cost, latency, power consumption, weight, size
- try to minimize function consisting of all variables
- *linear cost* function weights & sums individual costs

Partitioning: assign n objects to m locks such that
- all objects are assigned / mapped (uniquely / once)
- costs are minimized & all constraints are kept

Partitioning methods

Exact methods: get on optimal solution with minimal costs
- *enumeration:* iterate through all solutions & compare
- *integer linear program* (ILP)

Heuristic methods: get good solution with high probability
- *Constructive:* random mapping, hierarchical clustering
- *Iterative:* Kernighan-Lin algorithm, simulated annealing, evolutionary algorithms

4.1 Integer Linear program (ILP) (4-10)

An integer programming model requires two ingredients:
- *objective function:* linear cost expressions of int variables
- *constraints:* limit design space & optimization

$$C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in \mathbb{R}\ x_i \in \mathbb{N}$$

$$\forall j \in J : \sum_{x_i \in X} b_{i,j}\, x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R}$$

IP problem: minimize objective function under constraints

For partitioning, we can setup the following ILP
- $x_{i,k} = \{0,1\}$: determines whether object o_i in block p_k

$$\boldsymbol{x_{i,k} \in \{0,1\} \quad 1 \leq i \leq n, 1 \leq k \leq m}$$

$$\boldsymbol{\sum_{k=1}^{m} x_{i,k} = 1 \quad 1 \leq i \leq n}$$

$$\text{minimize} \quad \boldsymbol{\sum_{k=1}^{m} \sum_{i=1}^{n} x_{i,k} \cdot c_{i,k} \quad 1 \leq k \leq m, 1 \leq i \leq n}$$

Load balancing: maximal sum of all durations should be minimized (minimize T, where $T \geq processing\ time\ P_i$)

Additional constraints: can e.g. maximize number of objects in a single block

$$\boldsymbol{\sum_{i=1}^{n} x_{i,k} \leq h_k \quad 1 \leq k \leq m}$$

Maximizing the cost function: minimize negative function

ILPs are very popular for synthesis problems
- acceptable run-time with guaranteed quality
  (might be sub-optimal, as only search integer values
- scheduling can be integrated as well
- can add arbitrary constraints (however, hard to find)
- NP-complete (can take long time if too complex)
- good starting point for designing heuristical optimization

4.2 Heuristic methods (4-17)

Constructive methods

Try finding a good solution in a single computation

Random mapping: object is randomly assigned to block

Hierarchical clustering: stepwise group objects (i.e. assign to same blocks) by evaluating the closeness function
- always merge the two "closest" objects (maximal value)
- can stop after reaching desired level of clusters

Iterative methods

Start at one point and try to improve in steps

1. Start with an initial configuration
2. search *neighbourhood* (similar partitions) and select one as a candidate (slightly modified)
3. Evaluate *fitness function* of candidate
4. Stop when criterion is fulfilled / after some time

Hill climbing: always take the one with higher fitness
- if no more neighbours better, stop execution
- local optimum as a best result (depends on initialization)
- can start at various points to get good (& quick) estimate

KL: use information of previous runs to find global one
Simulated annealing: use complex acceptance rule (jump)
Evolutionary algorithms: complex strategy to add entropy

Kernighan-Lin algorithm (4-29)

From all possible pairs of objects, *virtually* regroup the best
- from the remaining objects, continue until all regrouped
- after $n/2$ turns, take lowest cost one & *actually* perform

External costs E_i : from node to nodes in *other* partition
Internal costs I_i : from node to nodes in *same* partition

Desirability to move: $D_i = E_i - I_i$

Gain: $g = D_x + D_y - 2 * c(x, y)$

Simulated annealing: vary (randomly), always take better-cost but also probability to take worse-cost neighbours
- *gradual cooling:* slowly decrease prob. of accepting worse

5. Multi-Criteria Optimization

Network processor: execute communication workload
- high-performance for network packet processing

For optimization, we require:
- *task model:* specification of the task structure
- *flow model:* different usage scenarios

The implementation defines architecture, task mapping & scheduling while considering objectives & constraints
- *objectives:* maximize performance, minimize costs
- *constraints:* memory, delay, costs, size (conflicting)
- results in a **performance model** of the system

Black-box optimization: can only give input, observe output and use objective function to optimize input

Constraints: only take feasibly solutions, add penalty

5.1 Multiobjective Optimizations (5-12)

Different objectives are often not comparable
- however, there are clearly inferior solutions

Can use classical single objective optimization methods
- simulated annealing, Kernighan-Lin, ILP
- decision making is done before optimization
- map all dimensions to one using some cost function

Decision space: feasible set of alternatives

Objective space: image of decision space using the objective function ("evaluated performance")

Pareto-dominated: if better or equal for all objectives
Pareto-optimal: not dominated by any other solution
Pareto-optimal front: set of pareto-optimal points

Population-based optimization costs: Pareto-optimal front
Use evolutionary algorithms to get a set of solutions
- decision making is done after the optimization
- function can then weight & map different image points

5.2 Multiobjective Evolutionary Algos (5-24)

Evaluate set of solutions simultaneously
- black-box optimization using *randomization* (no local min)
- *assumption:* better solutions are found near good ones

1. Choose a set of initial solutions (*parent set*)
2. *Mating selection:* select some out of parent set
3. *Variation:* use neighbourhood-operators to generate a new *children set*
4. Determine a union of *children & parent* set
5. *Environment selection:* eliminate bad solutions

Environmental selection

Criteria to choose which new solutions to take on:
- *Optimality:* take the ones close to (unknown) front
- *Diversity:* should cover a large part of objective space

Hypervolume indicator: should be maximized
- reference point: should be far away from optimal point (can strongly influence chosen set by weighting area)
- corresponds to region dominated by pareto-optimal front
- using dominated points will not increase the area
- all additional pareto-optimal solutions increase the area
- a better set (dominates other one) always has larger area

Choose the solution which increases hypervolume the least and throw it out of the parent set for the next round

Neighbourhood-operators (5-37)

Work on representations of solutions (e.g. integer vector)

Completeness: each solution has an encoding

Uniformity: all solutions are represented equally often (else biased to solution with many encodings)

Feasibility: each encoding maps to feasible solution (e.g. make it priority, not absolute definition)

Crossover: take 2 solutions & exchange properties

Mutation: randomly vary property of solution (reorder, flip, replace with different part)

6. System Simulation

6.1 System classification

System: combination of components to perform a function
not possible with the individual parts

Model: formal description of the system (*abstraction*)

State: contains all information necessary to determine the output together with the input for all future times

Discrete state models: countable number of states
- e.g. processors with registers

Continuous state model: actual analogue signals

Discrete time model: changes only at discrete times
Continuous time model: time advances the system

Events: tuple of a value v and a tag t
- $t = time$: timed event
- $t = sequence\ number$: untimed event

Time-driven simulation: time is partitioned into intervals
- simulation step is performed even if nothing happens

Event-driven simulation: discrete or continuous time base
- evaluation & state changes only at occurrences of events,

Discrete Event system (DES): event-driven system
- state evolution depends entirely on occurrence of discrete events over time (not by the evolution of time)
- *signals / streams:* ordered and/or timed events
- *processes:* functions which act on signals or streams

6.2 Discrete event simulation (6-16)

Modules describe the entire system & allow separation:
- *Behaviour:* described using logic and algebraic expression
- *State:* persistent variables inside these modules
- *Communication:* done through ports via signals
- *Synchronization:* done through events and signals

Event list: queue of events, processed in order
- organized as a priority queue (may include time)

Simulation time: represents current value of time
- during discrete event, clock advances to next event time

System modules: model subsystems of simulated system
- process events, manipulate event queue & system state
- *sensitivity list* indicates whether module is concerned

Zero duration virtual time interval: **delta-cycle (δ)**
- prevents the *cause* & *effect* events to coincide at the same time instance (if they occur instantly, outcome depends on ordering & race conditions occur)
- orders "simultaneous" events within simulation cycle

SystemC (6-23)

System-level modelling language to simulate concurrent executions in embedded systems (HW, communication)
- event-driven simulation kernel for discrete-event models

Processes are the basic units of functionality:
- *SC_THREADS:* called once, run forever (block if no input), can be suspended using $wait()$ / started with $notify()$
- *SC_METHOD:* event-triggered, require sensitivity list execute repeatedly when called without being suspended

Channel: contained for communication & synchronization
- can have state (FIFOs), implement one or more interfaces

Check that modules have sufficient initial tokens!

6.3 Simulation at high abstract levels

(Untimed) functional level: model functionality
- *C/C++, Matlab:* shared variables & messages

Transaction level: early SW development, timing
- *SystemC:* method calls to channels

Register transfer / Pin level: HW design & verification
- *Verilog, VHDL:* wires and registers

7. Design Space Exploration

Optimal design criteria

Mappings: all possible bindings of tasks to architecture

Request: operational cost of executing given task

Binding: subset of mappings so that every task is bounded to exactly one allocated resource (actual implementation

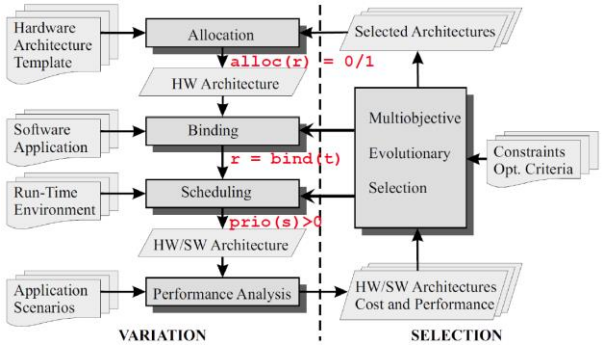
Design constraints

Delay constraints: maximal time a packet can be processed

Throughput maximization: maximize packets per second

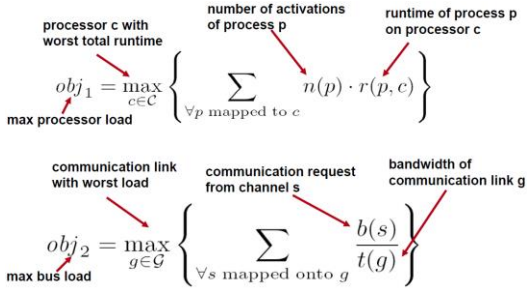
Cost minimization: implement only numbered resources

Conflicting usage scenarios: should be good for mixture



Simple analysis model

Optimize both to minimize maximal processor & bus load
- want to spread load over all CPUs, but not too much
- get numbers using *static parameters*, *functional simulation* & *instruction-set simulation* (using benchmarks)



8. Performance Estimation

High-level estimation: just look at the functional behaviour
- short estimation time, implementation details irrelevant
- limited accuracy, e.g. no information about timing

Low-level estimation: simulate all physical layers
- higher accuracy, deeper analysis possible
- long estimation time, need to define exactly

We use performance estimation to check:
- *Validation of non-functional aspects:* verification
- *Design space exploration:* allows optimization

Exploration: reconfigure system and evaluate performance

Performance metric: function giving a quantitative indication on system execution, should be representative
- time, power, temperature, area, cost, SNR, processing

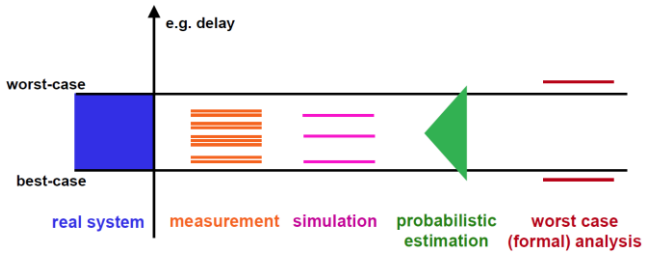
Evaluation difficulties

Non-determinism: computation (parallel), communication (interference), memory (shared resources), interactions

Cyclic timing dependencies: internal streams interact on computation & communication, influences characteristics

Uncertain environment: different scenarios (cached, pre-emption enabled), worst-case vs. best-case inputs

Various resource availability & demands: request depend on precise circumstances, where & when it is executed
- run-time of functions can vary depending on state



Estimation methods (8-19)

Measurements: use prototype to measure performance

Simulation: develop program which runs system model

Statistics: develop statistical abstraction & derive statistics

Formal analysis: mathematical abstraction to compute formulas which describe the system performance (WC)

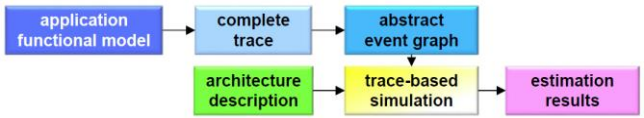
Analytic models: abstract system & derive characteristics
- performance measures are stochastic values (e.g. *avg*)
- can use for worst-case/best-case evaluation (*bracketing*)

Static analytic models: use algebraic equations & relations between components to describe properties
- fast & simple, but generally inaccurate modelling (scheduling, overhead, resource sharing neglected)

Dynamic analytic model: extend static models
- implement non-determinism in run-time & processing
- describe e.g. resource sharing (scheduling & arbitration)

Simulation: implement a model of the system (SW, HW)
- include precise allocation, mapping, scheduling
- combines *functional simulation* & *performance analysis*
- performance evaluation by running the entire program
- difficult to focus on part, but enables detailed debugging
- one run only evaluates for a single simulation scenario (specific input trace & initial system state)
- complex setup & extensive runtimes, but accurate

Trace-based simulation: separate functional & timing beh.
- trace only determined by functional application
- disregards timing (only focus on functional behaviour)
- faster than low-level simulation, as abstracting as events
- allows evaluation on multiple architectures using the same event graph on “virtual machines”



9. WCET Analysis

Hard Real-Time Systems: embedded controllers are expected to finish their tasks reliably within time bounds

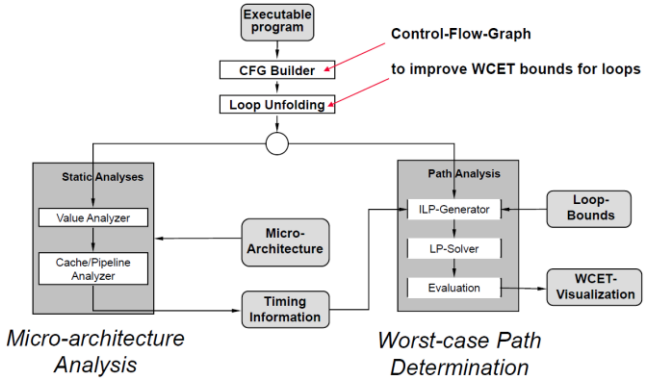
Worst-Case Execution Time (WCET): upper bound on the execution time of a task, should be kept minimal
- upper bound consists of all pessimistic consumptions
- can be approximated with exhaustive measurements

Usually, try to compute by analysing program structure
- modern processers exploit *parallelism*, therefore execution time not simply sum of single instructions
- *out-of-order execution* by leveraging independence
- caches, pipelines, branch prediction, speculation
- difference between BCET and WCET can be gigantic

Timing Accident: cause for increase of execution time
- execution is increased by *timing penalty*
- causes: cache miss, pipeline stalls, branch misprediction, bus collisions, DRAM memory refresh, TLB miss

Micro-architecture analysis: use abstract interpretation
- exclude as many timing accidents as possible
- determine WCET for basic blocks by analysing HW

Worst-case Path Determination: maps control flow graph to an ILP to determine upper bound & associated path



9.1 Program Path Analysis (9-18)

Determine sequence of instructions which is executed in the worst-case scenario (i.e. resulting in longest runtime)
- we know WCET for basic block from static analysis
- number of loops must be bounded

Basic block: sequence of instructions where control flow enters at beginning and exits at end without stopping in-between or branching (just linear sequence, SISO)

Determining the first instructions of basic blocks:
- The first instruction
- Targets of (un-)conditional jumps
- Instructions that follow (un-)conditional jumps

The WCET can then be calculated as the sum of the blocks:

$$WCET = \sum_{i=1}^{N} c_i \, x_i \;, \qquad \begin{matrix} c_i : WCET\ of\ block\ \ i \\ x_i : \#\ executions\ of\ i \end{matrix}$$

The number of executions of a block x_i is given by:
- **structural constraints:** given by *flow equations*
- **additional constraints:** extracted from program code (e.g. number of loop iterations, logical connections)

The entire ILP can then be written as:

$$WCET = \max \Big\{ \sum_{i=1}^{N} c_i \cdot x_i \;\Big|\; d_1 = 1 \wedge$$
$$\sum_{j \in in(B_i)} d_j = \sum_{k \in out(B_i)} d_k = x_i, i = 1 \ldots N \wedge$$
$$\text{additional constraints} \Big\}$$

program is executed once

structural constraints

9.2 Value Analysis (9-29)

Abstract Interpretation (AI): don't work on actual variables, but consider possible variable intervals (*abstract values*)
- can give exact WCET by considering all possible inputs
- supports correctness proofs

Value analysis is used to provide:
- Access information to data-cache/pipeline analysis
- Detection of infeasible paths
- Derivation of loop bounds

9.3 Caches (9-35)

Provide fast access to stored data without accessing main memory, as speed gap between CPU & memory is large

Assumes *local correlation between data access*:
- program will use similar data soon (many hits)
- program will reuse items (instructions, data)
- access patterns are evenly distributed across the cache

4-way set associative cache (9-39): store 4 tags / cache line

Least Recently Used (LRU): replace oldest block (ages)

We can distinguish two statically cache contents analyses:

Must analysis: *Worst-case*, "At which position at least?"
- each predicted cache hit reduces WCET (always hit)
- "union + maximal age": where is my worst position?

May analysis: Best-case, "At which position at most?"
- each predicted cache miss increases BCET (always miss)
- "union + minimal age": where is my best position?

Loop unrolling: improve analysis by limiting influence of state before the loop by executing it first as if
- more optimistic result for WCET, pessimistic for BCET

9.4 Pipelines (9-54)

Ideal case: finish 1 instruction per cycle

Instruction execution is split into several stages
- multiple instructions can be executed in parallel
- may execute instructions out-of-order

Pipeline hazards

Data hazards: operands not yet available
(data dependencies, cache miss)
(solve with pipeline stall & forwarding)

Resource hazards: consecutive instr. uses same resource

Control hazards: conditional branches (requires flush)

Instruction-cache hazards: instruction fetch causes miss

Cache analysis: prediction of cache hits (data / instr.)

Dependence analysis: analysis of data/control hazards

Resource reservation tables: analysis of resource hazards

Simulation

Processor: consider CP as a big state machine with initial state s, instruction stream b and trace t
Abstract pipeline: limit simulation to pipeline
- may lack information, e.g. about cache contents

Assuming *local* worst-case at every step leads to the *global* worst-case result (might be longer than in real-world)
- *timing anomalies* might however counteract this assumption (may be faster if delayed in beginning)
- can also join sets of states under this assumption (always keep most pessimistic look, as always safe)
- always assume cache misses where not excluded

10. Performance Analysis of Distributed Embedded Systems

Embedded system (ES)
- Computation, Communication, Resource interaction
- build a system from subsystems meeting requirements

10.1 Real-time calculus (10-11)

Abstract systems to calculate evaluation for all possible executions at once (entire behaviour in one analysis)

Min-plus
- used for interval arithmetic (abstract values)

$$\text{plus-times: } (S, \boxplus, \boxdot) = (\mathbf{R}, +, \times)$$
$$\text{min-plus: } (S, \boxplus, \boxdot) = (\mathbf{R} \cup \{+\infty\}, \inf, +)$$

Infimum: greatest element (not necessarily in set) which is less or equal to all other elements of the set

Metrics (10-17)

Data streams $R(t)$: number of events in $[0, t)$
Resource streams $C(t)$: available resources in $[0, t)$

Arrival curve $\alpha = [\alpha^l, \alpha^u]$
- max / min *arriving events* in *any* interval of length Δ

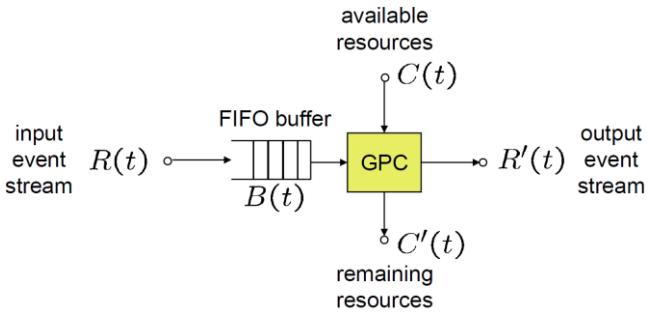
Service curve $\beta = [\beta^l, \beta^u]$
- max/min *available service* in *any* interval of length Δ

Common event pattern: specified by parameter triple
- p : period
- j : jitter
- d : minimum inter-arrival distance of events

Ex. *Periodic with Jitter*: (10-21)
Ex. *TDMA Resource*: (10-24)

Greedy Processing Component (GPC)

If tasks are available & resources ready, always use them
- assume *preemptable tasks* (can stop anytime if $C(t) = 0$)
- processes are only restricted by limited resources
- are processed one after the other in FIFO order



Computation: task instance $R(t)$, computing resource $C(t)$
Communication: data packet $R(t)$, bandwidth $C(t)$

$$\left.\begin{aligned} C(t) &= C'(t) + R'(t) \\ B(t) &= R(t) - R'(t) \end{aligned}\right\} \text{ Conservation Laws}$$

$$R'(t) = \inf_{0 \le u \le t} \{R(u) + C(t) - C(u)\}$$

u : last time the buffer was completely empty ($R' = R$)
- for $0 \le t' \le t$, we constantly use all available resources

$$R'(t) - R'(u) \le C(t) - C(u)$$
$$C'(t) = \sup_{0 \le u \le t} \{ C(u) - R(u) \}$$

Conservation law: $R'(t) \le R(t) \ \forall\, t$

Time domain: **cumulative functions**
Tine-*interval* domain: **variability curves**

$f \otimes g$ is called ***min-plus convolution***

$$(f \otimes g)(t) = \inf_{0 \le u \le t} \{f(t-u) + g(u)\}$$

$f \oslash g$ is called ***min-plus de-convolution***

$$(f \oslash g)(t) = \sup_{u \ge 0} \{f(t+u) - g(u)\}$$

Time-interval domain relations (10-32)

$$\alpha^l(t-s) \le R(t) - R(s) \le \alpha^u(t-s) \quad \forall s \le t$$
$$\beta^l(t-s) \le C(t) - C(s) \le \beta^u(t-s) \quad \forall s \le t$$

Using convolution, we can describe the relation as:

$$\alpha^u = R \oslash R; \quad \alpha^l = R \overline{\oslash} R; \quad \beta^u = C \oslash C; \quad \beta^l = C \overline{\oslash} C$$

The *output stream* of a component satisfies:

$$R'(t) \ge (R \otimes \beta^l)(t)$$

The *output upper arrival curve* of a component satisfies:

$$\alpha^{u\prime} = (\alpha^u \oslash \beta^l)$$

The *remaining lower service curve* of a component satisfies:

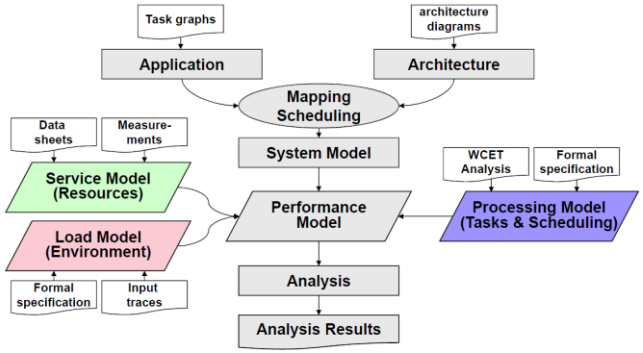
$$\beta^{l\prime}(\Delta) = \sup_{0 \le \lambda \le \Delta} (\beta^l(\lambda) - \alpha^u(\lambda))$$

Delay & Backlog (10-37):
- *backlog*: max. nr of components in queue / vertical diff.
- *delay:* max. time in queue / horizontal difference

$$B = \sup_{t \ge 0} \{R(t) - R'(t)\} \le \sup_{\lambda \ge 0} \left\{\alpha^u(\lambda) - \beta^l(\lambda)\right\}$$

$$D = \sup_{t \ge 0} \{\inf \{\tau \ge 0 : R(t) \le R'(t+\tau))\}\}$$
$$= \sup_{\Delta \ge 0} \left\{\inf \left\{\tau \ge 0 : \alpha^u(\Delta) \le \beta^l(\Delta + \tau)\right\}\right\}$$

10.2 Modular Performance Analysis (10-40)



Different scheduling mechanisms: (10-41)
- *Fixed priority/Rate monotonic*, EDF, *Round Robin*, TDMA

11. Various

Application Specific Instruction Set
- specialized, but still programmable (efficiently)
- HW with custom instruction set & operations

Translation Look-aside buffer (TLB): stores recent translation of *virtual* to *physical* memory

“Traffic shaper”: guarantee min. delay between tasks
- spreads bursts to minimize influence on other tasks